Probabilistic Language-Agnostic Word Segmentation

Alex King
Tufts University
May 2016

Abstract

Accurate segmentation of text without spaces is an important task in natural language processing, particularly for languages written without spaces such as Chinese. We exhibit a dynamic programming algorithm to efficiently compute multiple segmentations of text, storing the best sub-segmentation at each step. We measure multiple metrics of assessing segmentation quality, finding that a unigram language model with Laplace smoothing slightly outperforms a bigram language model with linear interpolation smoothing. Using an English corpus of ~4 million tokens, we achieve an F-score of 99.25, segmenting over 90% of sentences perfectly, with French and Turkish models performing nearly as well. This performance is significantly stronger than the baseline maximum matching approach.

Introduction and Background

In natural language processing, word segmentation is the process of breaking up a string of characters representing a phrase or sentence (*"thetabledownthere"*) into its proper constituent words or tokens (*"the table down there"*). For many eastern languages such as Chinese, which are written without spaces, efficient and accurate word segmentation is paramount for correct processing later in the pipeline. Word segmentation is not traditionally necessary in processing space-delimited languages like English; however, the process is becoming more applicable to all languages, with smartphone keyboard autocorrect becoming tolerant to strings of words typed without spaces. Word segmentation is generally easy for humans to do by eye, but it poses multiple challenges for computers. Exhaustive segmentation is extremely memory and time inefficient, as there are 2^{n-1} segmentations for any given string of n characters. In this paper, we exhibit a word segmentation algorithm that takes advantage of a probabilistic language model and dynamic programming to efficiently compute the most likely segmentation of a sentence.

Datasets Utilized

Supervised word segmentation performance is extremely sensitive to the data provided. To that end, we sought to use large segmented sentence-based corpora so that n-gram models could be generated as necessary. Unfortunately, it was challenging to find accessible large segmented corpora of unsegmented languages such as Chinese and Japanese. The largest segmented Chinese corpus easily accessed was roughly 10,000 sentences long, which is not large enough to build a strong probabilistic model of a language. Instead, this project focused on building a *language-agnostic* framework that could build a model for any language when given a properly segmented corpus. The *Leipzig Corpora Collection*[1] was found to be the best resource for this, offering sentence-based corpora scraped from news websites and Wikipedia in over a dozen languages.

English was used as a test language, utilizing a corpus of roughly 300,000 sentences. Sentences were tokenized using the Natural Language Toolkit (NLTK) `word_tokenizer` function. French and Turkish were informally tested afterwards, achieving very similar results to English. Sentences directly from the corpus acted as "ground truth" segmented sentences. Sentences

[1] "Leipzig Corpora Collection Download Page." *Wortschatz*. Leipzig University, n.d. Web. 05 May 2016.

had spaces removed before being passed to the segmentation algorithm. The corpus comes alphabetized, but it was shuffled before being partitioned into training and test segments.

Here is one example sentence from the English corpus: *"Mention must be made of bass-baritone Nicolas Testé's superbly robust singing as Raimondo, here portrayed as a spiritual advisor to Enrico Ashton."* This sentence exhibits several proper nouns that will be cataloged by our language model.

Baseline Algorithms

There are two straightforward mechanisms to segment text that share considerable algorithmic similarity, but differ greatly in performance. The *maximum matching* or *maxmatch* algorithm reads a string of unsegmented text with a pointer beginning at the end of the string, checking if the current string is a word in the language's dictionary. If so, a space is inserted and the process is repeated. If not, the pointer is moved to the left by one and the process is repeated. If no word is found, a single character token is created. In practice, this algorithm will always find the longest word that can be made before inserting a space, hence the name. Our example sentence *"thetabledownthere"* would be segmented as *"theta bled own there"*.

Minimum matching or *minmatch* works similarly, but instead of starting the pointer at the end of the string, it is started at the beginning. This always finds the shortest word that can be made before inserting a space. Maxmatch is a worthwhile baseline algorithm that achieves moderately high precision and recall scores for words. Minmatch, though very similar to maxmatch, does not fare nearly as well. This is understandable given that many prefixes of longer words are also words themselves; our example sentence would be segmented as *"the tab led ow n the re"*. For this project, maxmatch was used as the baseline segmentation algorithm.

Our Method

Maxmatch's beauty is in its convenience and speed: given a dictionary of a language, maxmatch can segment text in linear time with respect to the length of the input. It does not rely on a precomputed model, so it can be implemented and integrated quickly. However, its main flaw is its greed: it always picks the longest word at each opportunity, even when the longest word is extremely uncommon, throwing off the next word's segmentation as well. The *"theta bled own there"* example is one such example in English, where "theta" is picked as the first word of the sentence, even though it is many, many orders of magnitude less common than the most common word in English, "the". It also faces a certain conundrum in its definition of a dictionary: if it uses a dictionary defined by a real-world corpus, it may mistakenly recognize longer words than it should, because those words appear as slang or casual compound words in the corpus. Conversely, if it uses an official language dictionary, it will not fare nearly as well at recognizing uncommon proper nouns, affecting overall accuracy.

An ideal algorithm compares multiple segmentations of the same text and picks the one that is considered the best. However, this is not trivially accomplished; a given string of *n* characters will have 2^{n-1} possible segmentations, so runtime can grow to be $O(2^n)$ if implemented naively. Dynamic programming can improve on this. If we can define the segmentation problem such that an optimal solution is composed of optimal solutions to subsets of the original problem, we can store the solutions to the subsets for use later in the algorithm. For this project, we used the following recursive definition of the best segmentation of string *s*, derived from a recursive algorithm described by Jeremy Kun[2]:

$$\text{seg}(s_0^n) = \underset{0<x\leq n}{\text{argmax}}(\text{quality}(s_0^x, \text{seg}(s_x^n)))$$

This definition states that the best segmentation of a string of length *n* will be the highest quality segmentation among *n* choices. For example, for the short English string *"weare"*, the best segmentation will be the highest quality segmentation of the following:

- "w" | *seg*("eare")
- "we" | *seg*("are")
- "wea" | *seg*("re")
- "wear" | *seg*("e")
- "weare"

This definition relies on recursively calling the function on substrings of the original. This is where the dynamic programming algorithm helps: by iteratively calculating the best segmentation of increasingly long substrings of *s* and storing each one, the algorithm becomes polynomial in runtime rather than exponential. In the case of *"weare"*, the following steps would be taken to segment

[2] Kun, Jeremy. "Word Segmentation, or Makingsenseofthis." *Math ∩ Programming*. N.p., 15 Jan. 2012. Web. 04 May 2016.

the entire string. The bold selection indicates the highest scoring segmentation, the measurement of which will be discussed below.

1. *seg*("e") = **"e"**
2. *seg*("re") = **"r | e"**, "re"
3. *seg*("are") = "a | r | e", "ar | e", **"are"**
4. *seg*("eare") = **"e | are"**, "ea | r | e", "ear | e", "eare"
5. *seg*("weare") = "w | e | are", **"we | are"**, "wea | r | e", "wear | e", "weare"

The algorithm in Python is not much harder to read than pseudocode, so the Python code is exhibited below. Though the inspiration for this algorithm comes from Jeremy Kun's recursive segmentation algorithm, the dynamic programming algorithm was implemented, tested and tweaked independently.

```
def segment_string(string):

    length = len(string)
    table = [0 for i in range(length)]
    for row in range(length - 1, -1, -1):

        segment = string[row:length]

        segs =
            [[segment[:i+1]] +
             table[i + row + 1]
             for i in range(len(segment) - 1)]

        segs.append([segment])

        scores =
            [quality(sent)
             for sent in segs]

        table[row] =
            segs[scores.index(max(scores))]

    return table[0]
```

This algorithm allows us to efficiently compute many different segmentations of a string, which offers promise over maxmatch. However, its performance is completely dictated by the performance of the *quality* function, the function that tells us how good or how likely a given segmentation is. To measure quality, we compared the performance of two different probabilistic language model structures: one smoothed with Laplace smoothing, and another smoothed with linear interpolation.

Quality Method 1: Laplace-Smoothed Language Model

An n-gram language model stores probabilities of sequences of *n* words appearing in the language. Laplace smoothing adds one (or a smaller constant) occurrence for every n-gram in a language model, such that n-grams not appearing in the corpus will not have a probability of zero, but instead, a very small one. Some sort of smoothing for unseen tokens is crucial, because without it, the quality function cannot easily reconcile an unseen token: if it chooses to include the zero probability, the probability of an entire segmentation immediately goes to zero. Conversely, if the probability is ignored, there is no penalty for including a token that may very well be uncommon, leading to poor segmentation behavior.

Unfortunately, using n-grams with $n > 1$ is not especially practical for Laplace smoothing, as bigrams are extremely sparse. Computing quality based on smoothed bigrams exhibited very poor performance, because many bigrams were never observed, so all quality scores were very low. Because of this, we instead chose to use Laplace smoothing on a *unigram* model, such that quality was defined by the sum of log probabilities of each unigram in the segmentation.

In tweaking the model, it was clear that the probability of unknown tokens had to be reduced in two ways. First, based on the size of the corpus and the size of the languages they represented, counting each unseen token as occurring *once* in the model was far too high of a probability to achieve good results. Empirically, we were able to change this count of 1 to a very small count of 10^{-40} and achieve better results. Second, the quality of a very long unknown word needed to be penalized, otherwise a long string of unknown characters would often be considered better than a string of short, segmented characters with probabilities compounded together. Jeremy Kun's article recommended penalizing unknown tokens by their length, and we found that penalizing all unknown tokens by dividing their (pre-log) probability by $10^{\text{len(s)}}$ achieved the desired effect.

Quality Method 2: Linear Interpolation-Smoothed Language Model

We also evaluated quality using a bigram model smoothed with linear interpolation. This method allows us to take advantage of the richer data inherent in a bigram model, but it fares better at smoothing than Laplace smoothing, because any unseen bigram is instead judged by the probability of its second unigram. We predicted that this method would lead to stronger segmentation accuracy, because the bigram model would be better at discerning segmentation between neighboring words. Lambda values for interpolation were calculated using the deleted interpolation algorithm using a held out corpus. The lambda values gave roughly 70% importance to bigram probability and 30% importance to second-unigram probability. Segmentation quality was defined by the sum of log probabilities for each bigram in the segmentation.

Algorithmic Complexity

Building a probabilistic model takes $O(|C|)$ time, where $|C|$ is the corpus size in words. The segmentation algorithm itself, on a string of length n, computes roughly n segmentations of length n n times, where each segmentation must have a quality calculated, taking $O(n)$ time. Therefore, the probabilistic dynamic programming algorithm runs in $O(n^3)$ time. Though this is considerably worse than maxmatch, sentence breaks are our friend. Sentence length does not typically grow without bound, so $O(n^3)$ performance on sentences of reasonable length is plenty fast in practice.

Evaluation

To assess the quality of a segmented sentence, full sentence segmentation accuracy was computed, along with precision, recall, and F-score of individual words in the sentence. Because it is very possible for long sentences to have a single incorrectly segmented word or word pair, full sentence accuracy is a much harder statistic to bring up in value. However, precision, recall, and F-score represent a fairer evaluation of the quality of segmentation, representing roughly what percentage of words are correctly segmented overall. Definitions of precision and recall with respect to segmented sentences were taken from *"Bigram Chinese Word Segmentation by Viterbi Algorithm"* by Dan Lieu et al.[3]

Calculating precision and recall of a segmented sentence based on ground truth data was a fun algorithm to build in itself, and thankfully not unreasonably difficult or inefficient. It's necessary to measure how many words in the ground truth sentence are correctly identified in the segmented sentence, but it isn't as easy as exhaustively searching for presence; order and relative position matter. Consider the following sentences:

Ground truth: *The table down there*

Segmented 1: *Theta bled own there*

Segmented 2: *The tab led ow n the re*

Comparing either of the machine-segmented sentences to the ground truth sentence requires walking each sentence in parallel, beginning at the first word, and seeing if the most recently segmented words are the same, and that the pointer for each sentence is at the same overall position in each string. This algorithm was devised and implemented independently and is documented in the `WordsCorrect` function in the segmentation module.

Results

Results are summarized in the table below with the best performances bolded. All measurements were taken with a training corpus of 290,000 sentences and a test set of 5,000 sentences. M denotes the maxmatch algorithm, LI denotes the linear interpolation-smoothed model, and LP denotes the Laplace-smoothed model.

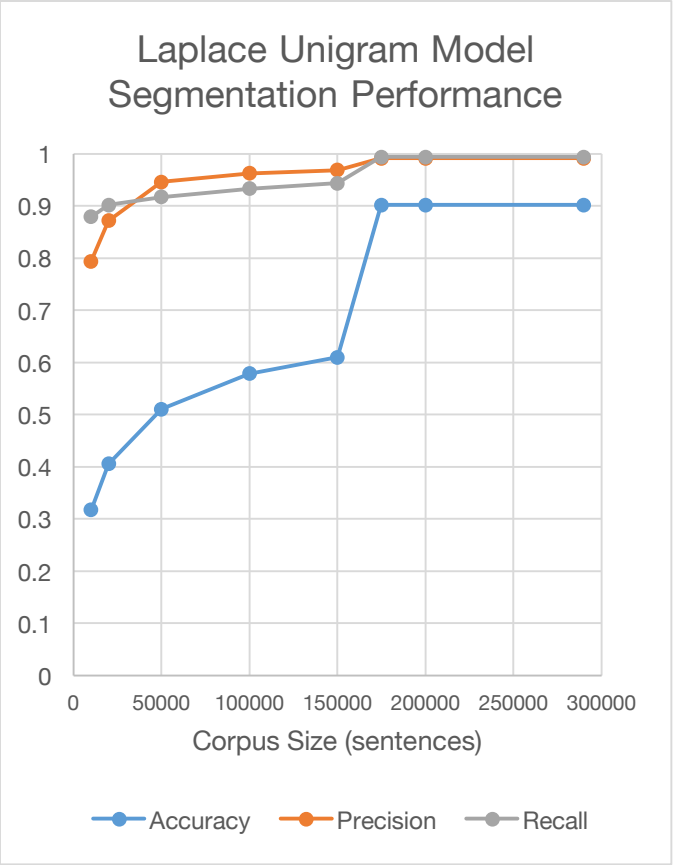
	Accuracy	Precision	Recall	F-Score	Time (s)
M	18.68%	83.22%	78.54%	80.81	**3.99**
LI	84.98%	98.56%	98.94%	98.74	110.51
LP	**90.16%**	**99.13%**	**99.37%**	**99.25**	462.43

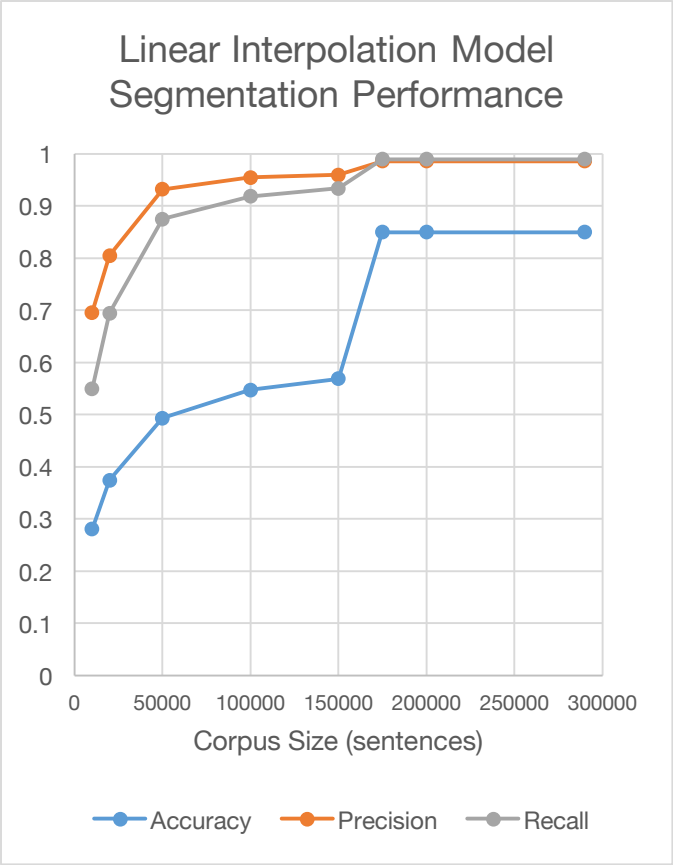
Maxmatch exhibited significantly poorer sentence accuracy (18.68%) than the probabilistic methods, but its precision and recall scores were acceptable, with an F-score of over 80. This demonstrates that accuracy improves in a delayed fashion relative to precision and recall. Maxmatch's main advantage is that it runs over 25 times faster than the linear interpolation method and over 100 times faster than the Laplace method. This was expected; it is not only a linear time algorithm, but it avoids the numerical operations inherent in probabilistic comparison. This also would explain why Laplace smoothing is considerably slower than linear interpolation; Laplace smoothing, with its penalization of long length, computes more numbers with time-intensive exponent and division operations.

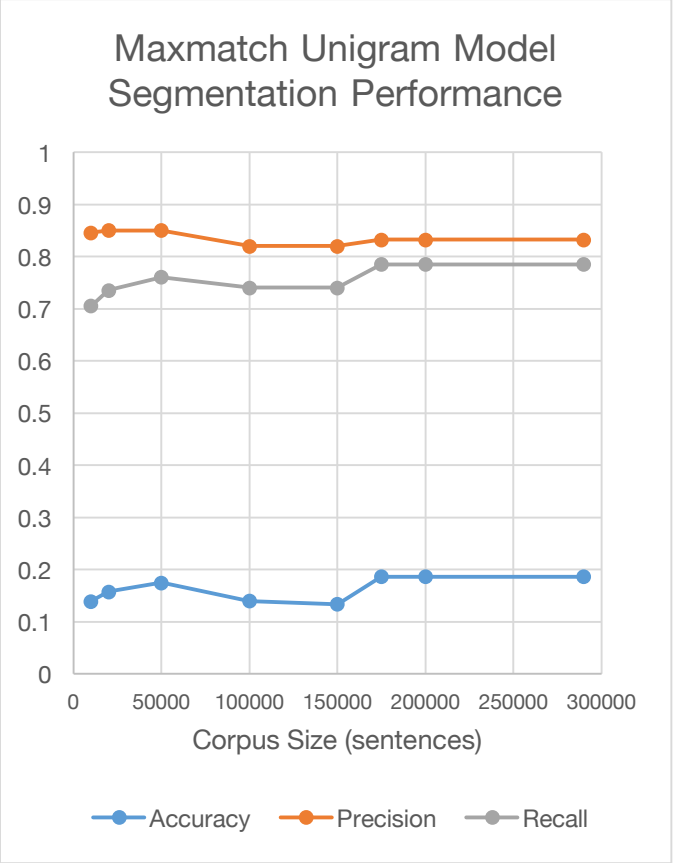
Laplace smoothing and linear interpolation both demonstrated very similar performance curves, with Laplace slightly edging out linear interpolation in F-score, and achieving roughly 5% better sentence segmentation accuracy. This is somewhat surprising, given that a bigram model would be expected to hold richer information about the most likely segmentation of neighboring words.

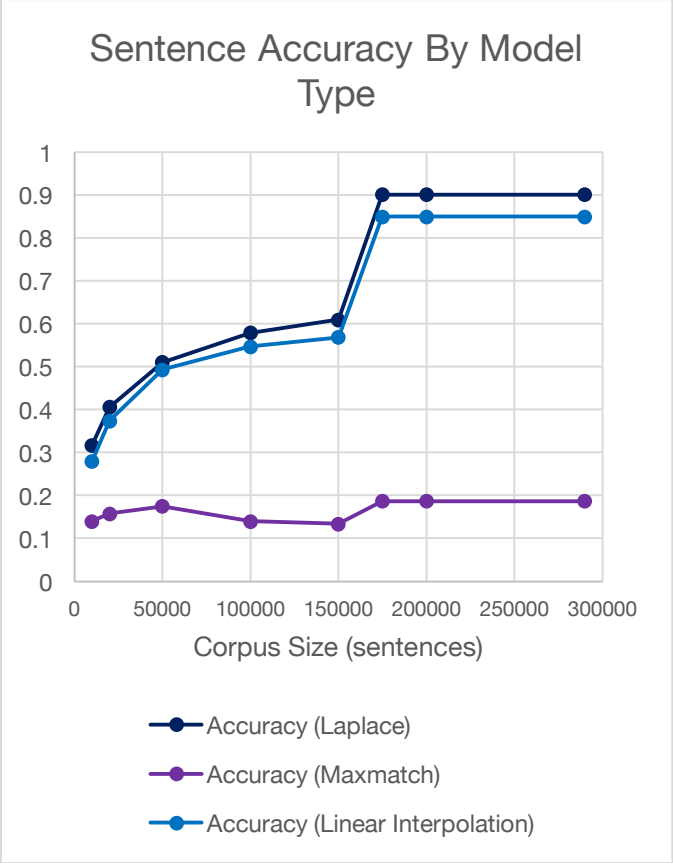
On the following page are several graphs demonstrating the segmentation performance of each algorithm depending on the size of the corpus. Unsurprisingly, as the corpus grew in size, probabilistic segmentation accuracy improved. Maxmatch saw less variance in performance, as it is not probabilistic but rather dictionary presence-based.

[3] Liu, Dan, Weiguo Fang, Hong Zhou, and Yan Li. "Bigram Chinese Word Segmentation by Viterbi Algorithm." *2009 Sixth International Conference on Fuzzy Systems and Knowledge Discovery* (2009): n. pag. Web. 4 May 2016.


Laplace Unigram Model Segmentation Performance
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
50000
100000
150000
200000
250000
300000
Corpus Size (sentences)
Accuracy
Precision
Recall


Linear Interpolation Model Segmentation Performance
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
50000
100000
150000
200000
250000
300000
Corpus Size (sentences)
Accuracy
Precision
Recall


Maxmatch Unigram Model Segmentation Performance
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
50000
100000
150000
200000
250000
300000
Corpus Size (sentences)
Accuracy
Precision
Recall


Sentence Accuracy By Model Type
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
50000
100000
150000
200000
250000
300000
Corpus Size (sentences)
Accuracy (Laplace)
Accuracy (Maxmatch)
Accuracy (Linear Interpolation)

It should be noted that the peculiar jump in accuracy for the two probabilistic models between 150,000 and 175,000 sentences, and the flat performance between there and 300,000 sentences, likely has to do with how the body of data was partitioned for testing, which indicates an oversight in experimental design. One unique random shuffling was selected and utilized for all testing, but this particular shuffling ended up being somewhat uneven. Ideally, after the corpus was initially shuffled, each sub-corpus of various sizes would have been a random sampling of the entire corpus, rather than a simple partition. Even then, a trend is clearly observable, just not as smooth as one would like.

Though our best-performing Laplace model segments around 99% of all words correctly, it makes certain mistakes that reveal that it has a relatively limited underlying model of the English language:

Ground truth: *If nobody can, it **may be** on the property*

Laplace: *If nobody can, it **maybe** on the property*

The Laplace model finds that "maybe" is a more likely occurrence than "may be". This is not a wrong assumption in terms of pure probability; it simply does not have added information about syntax to make a more informed decision. This is a mistake that maxmatch would make as well. A higher-order n-gram model may achieve better results.

Other times, the Laplace model performs quite well, making a single mistake that is not critical to sentence meaning:

Ground truth: *Two graduating university students offer tips to **first-year** students*

Laplace: *Two graduating university students offer tips to **first - year** students*

Here, the only discrepancy is the hyphenation of "first-year" in the ground truth sentence. This is a pattern that could easily be patched up with an additional layer of rule-based post processing after the segmentation algorithm does its initial pass.

Overall, the Laplace model exhibited slightly stronger performance on a large corpus, but achieved much better precision and recall on a small corpus, which indicates that a bigram model should only be used if a large corpus can be found. In practice, the linear interpolation model performed almost as well as the Laplace model and ran roughly four times faster, so it would appear to be the most practical choice of the algorithms exhibited.

Conclusion

In this project, we demonstrated an effective algorithm for word segmentation that allows for plug-and-play assessment of segmentation quality. Two measurements of quality were assessed, with a Laplace-smoothed unigram model faring the best, indicating that novel probabilistic smoothing may be as or more important than n-gram data. Most impressively, this level of performance is achieved without making any language-specific assumptions, instead only assuming that as unknown words become especially long, their probabilities of appearing fall. Tested on a French corpus of the same size, the Laplace model saw an F-score of ~99 and sentence accuracy of ~87%. Tested on Turkish, the model saw an F-score of ~97, and a sentence accuracy of ~76%.

Segmentation accuracy could be improved by creating stronger representations of phrase quality. Such representations could include higher-order n-gram models with more complex forms of smoothing, as well as a model that includes data such as part-of-speech n-gram likelihood on top of word likelihood.

We also would be curious to build a semi-supervised algorithm on top of this functionality. Whenever a segmentation is incorrect, there is rich information available that shouldn't be thrown away, but instead fed back into the model to improve segmentation accuracy. For example, an earlier version of the algorithm was often mis-segmenting "Your" as "Y | our". Such a mistake should be recognized and penalized, such that the likelihood of a future segmentation of "Y | our" decreases. We imagine a rule-based transformation could take place *after* the segmentation algorithm runs. This would require finding one-to-many and many-to-one mappings of words between the two sentences to construct rules, and scanning sentences for strings of words to apply rules. This could be accomplished in no worse than $O(n^2)$ time.

References

"Leipzig Corpora Collection Download Page." *Wortschatz*. Leipzig University, n.d. Web. 05 May 2016.

Kun, Jeremy. "Word Segmentation, or Makingsenseofthis." *Math ∩ Programming*. N.p., 15 Jan. 2012. Web. 04 May 2016.

Liu, Dan, Weiguo Fang, Hong Zhou, and Yan Li. "Bigram Chinese Word Segmentation by Viterbi Algorithm." *2009 Sixth International Conference on Fuzzy Systems and Knowledge Discovery* (2009): n. pag. Web. 04 May 2016.