Search for Word Break

Nitin Gaddipati (ngaddipa) **Karandeep Johar (kjohar)**

Varshaa Naganathan (vnaganat)

1 Introduction

The word break problem is defined as follows: Given an input string that is not space-delimited and a dictionary of words, the goal is to space-separate the input string into a sequence of dictionary words.

In some languages such as Chinese, sentences are written without spaces between the words, and an important first step in language processing is segmenting sentences into words. Say we have a sentence which is a sequence of characters and a dictionary, which is the set of words (each word is also a sequence of characters). Given any such sentence, the goal is to split the sentence into words from the dictionary. For example, if D = {i,cat,dog,see,sleep,the}, then given the sentence "iseethecat", [i, see, the, cat] is a possible segmentation.

Common applications for word segmentation include parsing URLs and hastags.

A dynamic programming solution exists for this problem which runs in $O(MN^2)$ where M is the size of the dictionary and N is the size of the input string. We propose a search based solution using different heuristic functions derived from language models to find the best possible parse.

2 Baseline

As a baseline, we used the data set and approach provided by Grant Jenks[1]. The set of test sentences provided with the code base was used as unit tests for development of our search algorithms.

The baseline code provided expanded along every possible split of the input string and picked the parse with maximum score based on a bigram language model with backoff. There was no pruning of the search space.

3 Formulation as A^* Search

We can come up with our own utility function and construct a deterministic state space model for this task.

Our state consists of three parts:

[<segmented words>, <most-recently segmented word/prefix>, <unsegmented string/suffix>]

In the initial state, the entire input needs to be segmented. For example, if the input sequence is "themeterman",

sstart = [ϕ, "<s>", "themeterman"], where <s> tag just denotes the start symbol.

In this model, $actions$ will represent choosing the next word to segment. For any state s, $actions(s)$ is dividing the unsegmented part of the string.

actions(s) = {w: w is a prefix of the remaining input}

In our example,

actions(sstart) = {”t”, ”th”, ”the”,”them”,”theme”,”themet”,”themete”, ”themeter”, ”themeterm”,”themeterma”, ”themeterman”}

Taking an action (choosing the next word to segment) removes that word from the input sequence and makes it the prefix. The old prefix is added to the segmented words. For example,

next_state(sstart, ”the”) = [{”the”}, ”the”, ”meterman”]

next_state(sstart, ”them”) = [{”them”}, ”them”, ”eterman”]

next_state([{”the”}, ”me”, ”terman”], ”term”) = [{”the”, ”me”}, ”term”, ”an”]

Finally, we have reached the goal when no input remains to be sequenced.

Instead of just finding a single solution we can instead modify our states to keep track of the fluency of our current segment. We can negate this fluency because a high fluency should correspond to a low cost.

An elementary form of fluency score could be the product of the unigram probability of the segmented words in the state. Note this would mean that fluency of [”humpty”, ”dumpty”] would be exactly the same as the fluency score of [”dumpty”, ”humpty”]. This is the major flaw with this approach.

Until now our algorithm does not take into account the context of the word. Consider the segmentations of ”endear”. Clearly ”end” and ”ear” are common words, but they are very rarely written in that order. Much more common is the single word ”endear”, and an optimal model would take this into account.

A solution would be to use Bayes rule and break the joint probability of a set of words into product of conditional probabilities of word given its history. We then use the Markov assumption to truncate the history to size 1. We do this to due to memory and speed considerations.

3.1 Algorithmic correctness

We can prove the correctness of our search by induction[2]. Say we know the optimal segmentations after the first split. In our running example, say we know the fluency score of all the successor states of the initial state. Then we know that the most fluent segment would be the one which has the largest probability product. Because we look at all possible split pairs we know that our segment method would return the optimal segmentation. Since each unsegmented string contains a strictly smaller substring we know that our algorithm will terminate and is thus correct.

3.2 What constitutes a word?

The hardest part of word segmentation is word identification[2]. Initially we proposed to look at only those words contained in a static dictionary. We soon found a flaw with this approach. Our program failed to recognize strings such as ”bbc” as a word which almost all humans would categorize as a separate word. To counter this we assigned a very low probability for any sequence of alphabets that are not in the dictionary.

Say ”bbc” was not in our dictionary and we were given the string ”bbcamerica” to segment. We would return ”bbcamerica” as one string as the output. Ideally we should have returned ”bbc america”.

We need to incorporate information about the frequency of words based on their length. In particular, assume we have an unknown word. It is unlikely to be a word if its longer than, say, ten characters long. Improving this we assigned a penalty on the word on the basis of its length. The intuition behind this is to give a higher score to ”bbc america” as opposed to ”bbcamerica”.

3.3 Pruning

Say for an intermediate parse of the string ”wheninthecourseofhumanevents” we have found a best score of -100 (log probability of score). We have now currently segmented into [”wheninthe”,

”courseofhumanevents”] and now we have to segment courseofhumanevents. Say our current score is -101 because of the unlikely word of wheninthe. We can prune our search and decide not to go any further because we know that all score from this particular state can never exceed the best score. This is due to the fact that probability of an event occurring is between 0 and 1 and log of that quantity is always negative. Implicitly we are using an admissible heuristic of 0.0 here. We can instead use an upper bound which takes the maximum conditional probability of a bigram with the last segmented word as the prefix. This would be a much better estimate of score.

3.4 Memoization

To prevent redudant computation we save scores for a particular prefix and unsegmented string pair. Therefore, we will only expand once for [{”the”}, ”meter”, ”man”] and [{”th”, ”e”}, ”meter”, ”man”]

4 Using PCFG

4.1 Motivation

The idea behind using a probabilistic context free grammar (PCFG) is based on the observation that the structure of sentences is implicit within the bigram corpus. For example, the phrase ”the ball” can be expected to be seen in the corpus since its grammatically correct in the English language. However, the phrase ”ball the” will never appear in the corpus. Stating this example in a formal sense, we can definitively conclude that the Determiner, such as ”the”, will always precede a Noun, such as ”ball”. For this project, we used our own simple PCFG which is as follows:

```
ROOT -> S1 99
S1 -> NP VP 1
VP -> VerbT NP 1
NP -> Det Nbar 20
NP -> Proper 1
Nbar -> Noun 20
Nbar -> Nbar PP 1
PP -> Prep NP 1
```

Figure 1: Simple PCFG (screenshot from editor window)

In practice, it does not matter what PCFG is used as long as we can ascertain that the input strings can be parsed using it. Significant research has been done into generating complex PCFGs which are designed to model all of the English language with very high accuracy.

4.2 Assumptions

- Input strings are non-space separated sentences that can be parsed using our chomsky-normalized probabilistic context free grammar which includes a lexicon.
- The score of a prefix and suffix is determined using POS tagging and reachability if applicable.

4.3 POS Tagging

We use our own lexicon which maps terminals to their possible parts of speech tags. This is for testing purposes. In practice, we can easily switch to a high accuracy tagger such as the Stanford POS tagger which is capable of tagging any word.

If both prefix and suffix are not found in the lexicon, we penalize based on the square of the length of suffix. If only one or the other is found, we penalize only on the length of the word not found. If

both are found we move on to reachability. POS tagging essentially serves as a front which gets rid of all the splits that result in prefixes and suffixes that are non-sensical.

In the context of the "the ball" example discussed previously, "the" would be tagged as Det and "ball" would tagged as Noun.

4.4 Reachability

We define reachability as follows: Given two POS tags and a PCFG, what is the probability that the second tag (tag2) immediately succeeds the first tag (tag1)?

This problem can be broken up into two cases. The first case involves a forward search on every rule containing tag1 as the first terminal after the arrow. If the succeeding element is a terminal then we check to see if that terminal is tag2. If it is not, then that rule is inconclusive in terms of establishing whether tag2 can immediately succeed tag1. If the succeeding element is a nonterminal, then all the rules that begin with that nonterminal must be analyzed where their rules begin with a preceding terminal. If none of the rules containing the nonterminal have this property, then we can definitely state that the forward search was inconclusive. A simple example of the forward search is as follows:

In the context of the the ball example discussed previously, we can see from Figure 1 that the rule is NP $\rightarrow$ Det NBar is the only rule containing tag1 (Det) as the first terminal after the arrow. From here, we look for all rules beginning with NBar which are NBar $\rightarrow$ Noun and NBar $\rightarrow$ NBar PP. On a side note, careful care must be taken in order for this process to avoid getting stuck in cycle. For example, the rule NBar $\rightarrow$ NBar PP can easily cause never ending recursion if proper care is taken to avoid this. The algorithm first looks at the rule NBar $\rightarrow$ Noun and comes up with a possible match. In the context of the code, this is considered as a path (Det $\rightarrow$ NBar $\rightarrow$ Noun).

The second case involves a backward search on every rule containing tag2 as the sole terminal. Take for example, "...bag for..." where "bag" is tagged as a Noun and "for" is tagged as a preposition. The search begins with the rule NBar $\rightarrow$ Noun. All other rules beginning with NBar are analyzed which results in the rule NBar $\rightarrow$ NBar PP. Note that this search is slightly more complicated than the first case since we have to keep searching until we hit a terminal. There is a very high probability that we will see rules ending with two nonterminals as the grammar is Chomsky-normalized. When we evaluate PP, we see that PP only has the following rule PP $\rightarrow$ Prep NP where Prep is a terminal. Thus, we can definitely state that Prep is reachable from Noun according to the path Noun $\rightarrow$ NBar $\rightarrow$ PP $\rightarrow$ Prep.

Note that the weights on each of the rules traversed in both the searches is summed from which the PCFG scorer decides the best path to use. This is an interesting point since grammar structure is implied in the score calculation. Take for example, the string "forthis", the split "for this" will have greater score than the split "fort his" because there is a greater probability of the Preposition, "for", preceding the Pronoun, "this" than the Noun, "fort", preceding the Pronoun, "his".

4.5 Complexity Analysis

Even though using the PCFG Heuristic expands much less nodes than its bigram counterpart, the time spent in each node is more than that spent using the bigram heuristic. However, this overhead increase is only minimal in that the runtime for the PCFG calculation is entirely based on the list of rules which is much smaller in size compared to the list of bigrams.

Word to POS lexicon

This is the cost to obtain all possible POS tags for a given word. In the worst case, we may end up having to traverse the entire lexicon and still not come up with a possible mapping. Thus, this cost is $O(L)$ where L is the number of lexicon rules.

Reachability

This can be broken down into the following cases:

- tag1 is a preceding terminal $\rightarrow$ forward search on succeeding element
  In the worst case, for each rule containing tag1, we go one level deep since every rule ends in a non-terminal. Thus, in this case we traverse $O(2T_1)$ rules where T_1 is the number of

rules containing tag1. Note that the check for the existence of tag1 does not incur any cost because the pre-processing step, which runs in $O(G)$ where G is the number of rules in the grammar, optimizes the grammar for fast look ups using nested lists within dictionaries.

- tag1 is a lone terminal $\rightarrow$ backward search on preceding element
  In the worst case, we may have to go through the rules in order to find a terminal so this would simply be $O(G)$

Thus,

$$\begin{aligned} reachability\ cost &= preprocessing\ cost + forward\ search\ cost + backward\ search\ cost \\ &= O(G) + O(G) + O(2T_1) \\ &= O(G) \end{aligned}$$

Total Cost of PCFG Heuristic

The cost of PCFG scoring can now be broken up as follows:

$$\begin{aligned} PCFG\ scoring\ cost &= Word\ to\ POS\ lexicon\ cost + reachability\ cost \\ &= O(L) + O(G)\ where\ L\ is\ the\ size\ of\ the\ lexicon\ and\ G\ is \\ &\quad the\ number\ of\ grammar\ rules \end{aligned}$$

Note that, in practice, $L + G$ is much smaller than the size of a typical bigram corpus that is meant to model the English language with reasonably high accuracy. Thus, we can definitively conclude that, with the PCFG heuristic, the intra-node runtime overhead is a small sacrifice that we have to make in order to achieve phenomenal savings in the number of overall node expansions, which will be shown in the next section.

5 Results

In this section we discuss trends and results observed on implementing the two methods presented above. First, we present results based on the number of node expansions performed by each algorithm. For this, we use the set of test sentences from our baseline code for the A^* search method and a set of sentences generated from our PCFG for the PCFG method.

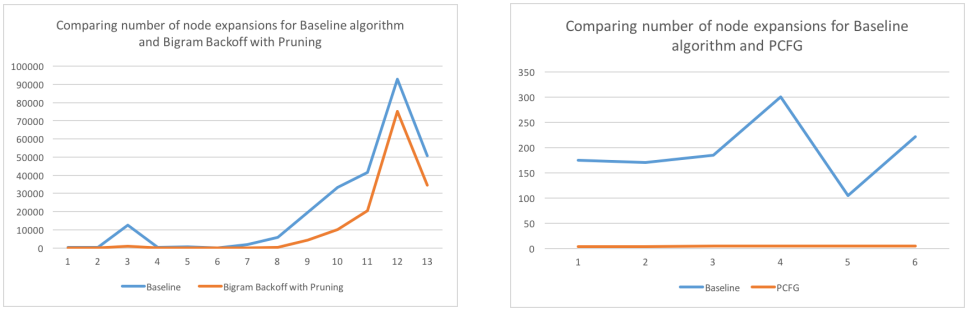

Figure 2: Number of node expansions performed by each algorithm

As can be seen from the figures above, both the A^* search method as well as the PCFG method gave lesser node expansions than the baseline. However, in the case of the PCFG method, we need to keep in mind the tradeoff between time spent per node and the number of nodes expanded on as discussed in the Runtime Analysis section.

The second set of metrics we used to measure performance was Precision and Recall. Precision is defined as - For the boundaries that were inserted, how many were correct while Recall is defined as - For all the correct boundaries, how many were found. These results are presented in the graphs below.

The first graph shows Precision and Recall results on a random Wikipedia article, cleaned to remove any characters other than alphanumerics. The PCFG method performs quite poorly on this data set.

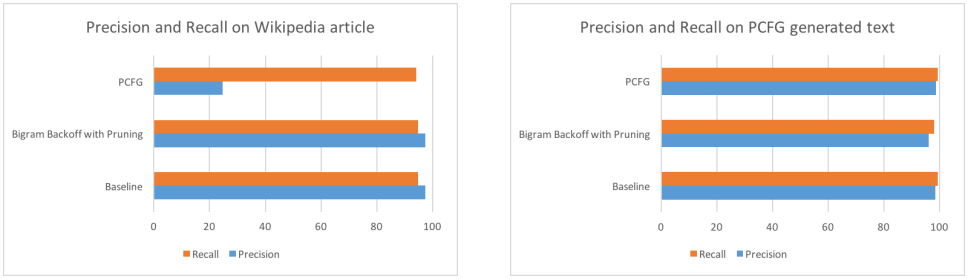

Figure 3: Precision and Recall for each algorithm on different data sets

This is to be expected as our PCFG is a toy grammar and does not model the language used in regular articles.

For the second graph, a corpus of 100 sentences generated by our PCFG was used. On this dataset, all three methods had comparable Precision and Recall values.

Runtime Analysis

The actual total run time on the Wikipedia article was found to be 46.6 s for the baseline method, 39.6 s for the A^* search method and 1 min 11s for the PCFG method. These runtimes reiterate the tradeoff involved between the A^* search method and the PCFG method between number of node expansions and the work done per node.

In conclusion, we believe that writing a PCFG that will model the data on which word segmentation tasks need to be performed could be a good investment.

6 Alternate Representation - Reinforcement Learning

We also attempted to formulate the Word Break problem using Reinforcement Learning. The prefix and suffix of the string after introduction of a space was considered as the state and the position in the suffix where a new space should be introduced was considered an action. At each state, the action to be taken was decided using an ϵ-greedy approach. The reward function was a large number minus the Bigram backoff function defined in the next section.

For example, consider the input string "itisspringinPittsburgh". The start state has prefix: "¡s¿" and suffix: "itisspringinPittsburgh". If the action selected is to introduce space at position 2, the new state becomes the following - prefix: "¡s¿ it", suffix: "isspringinPittsburgh". SARSA(λ) was used to determine the values of the Q function.

This state space definition is not generic enough to allow for extension of a policy learnt to strings yet unseen. Ideally, we would want to incorporate generic information about the language structure of a sentence, POS tags for example, as a feature vector in the state space description. Though our aim was to eventually find a policy which once learnt from a training corpus could determine the word break for a given test sentence, we initially wanted to see if we could emulate search for a specific string using a RL formulation. However, this formulation led to all strings converging to the case of a space after every letter. Hence, the string in the above example would converge to "i t i s s p r i n g i n P i t t s b u r g h".

Though this attempt was unsuccessful, we believe that a good RL formulation can solve the RL problem.

7 Future Work

Apart from coming up with better formulations to use Reinforcement Learning to solve Word Break, we would like to improve the scalability of the A^* approach proposed. One natural way to do this would be to parallelize the algorithm.

Different languages have different definitions of what a word is. We would like to explore whether the current formulation would work for languages other than English.

Since experimenting with PCFGs showed that having a grammar that modeled our data well is useful, we also believe that it would be worth the effort to build a PCFG that would model specific data sets, Wikipedia articles for example, better.

References

[1] Jenks, Grant. Python Word Segmentation.

[2] Word Segmentation, or Makingsenseofthis, Jeremy Kun

[3] Johnson, Mark, and Sharon Goldwater. "Improving nonparameteric Bayesian inference: experiments on unsupervised word segmentation with adaptor grammars." *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics. Association for Computational Linguistics*, 2009.

[4] Gambell, Timothy, and Charles Yang. "Word segmentation: Quick but not dirty." *Unpublished manuscript* (2006).

[5] Chen, Songjian, Yabo Xu, and Huiyou Chang. "A Simple and Effective Unsupervised Word Segmentation Approach." *AAAI*. 2011.

[6] CS221: Artificial Intelligence: Principles and Techniques, Fall 2012 edition, Stanford University