Text Segmentation on a Probabilistic Unigram Model using Viterbi Algorithm

Aditya Batheja and Sidharth Thapar

College Of Computer and Info Sciences, Northeastern University, Boston, MA 02215

batheja.a@husky.neu.edu | thapar.si@husky.neu.edu

Abstract

This study aims to extract hidden words from continuous text (obtained by removing spaces from original text) by detecting word boundaries and performing text segmentation. Viterbi Algorithm is used on a Unigram model of word probabilities to retrieve the original text by inserting spaces at correct locations with the aim of maximizing the cosine similarity of the resultant text and the original (formatted) text. The analysis of the results produced by comparison confirms the efficiency of this approach to solve the problem.

Introduction

“Ifyoucanreadthis,congratulationsyoujustsolvedatextsegementationproblem!.” This may seem intuitive to humans, but represents a problem in Natural Language Processing. Text segmentation is of utility in speech to text conversion and languages like Arabic, Chinese and Japanese which unlike English do not use blankspaces as explicit delimiters between words. Text segmentation has also been found useful in handwriting recognition where the writer’s language and style change the spaces between words. This study aims to provide a solution for Text Segmentation problems using Viterbi algorithm. This solution relies on Zipf’s law to build the word sequence from a frequency dictionary of the language.

Problem Statement : Given a block of English sentences with the spaces removed, detect the word boundaries and insert spaces while maintaining punctuations. This is also known as the ‘word boundary problem’ in continuous speech.

The Viterbi Algorithm is used widely in Natural Language Processing. For Example, in speech-to-text conversion the acoustic signal is treated as the observed sequence of events, and a string of text is considered to be the "hidden cause" of the acoustic signal. The Viterbi algorithm finds the most likely string of text for the given acoustic signal.

Viterbi Algorithm:

"Viterbi algorithm" has become a standard term for the application of dynamic programming algorithms to maximization problems. The viterbi algorithm is used to return the most likely sequence of states from a list of hidden states.

This makes Viterbi an ideal algorithm for this problem, where words are regarded as the hidden states. The algorithm works by multiplying probabilities of word sequences and choosing the most likely sequence. A unigram model is used at this stage. This means that the probability of one word occurring are regarded as independent of the previous word. A frequency dictionary of the English language is used for the probabilities of all words. For each sentence the viterbi algorithm is used to find out the most likely sequence of words in the list of characters that it is.

Example problem :

Given a stream of characters with spaces removed from in between words :

ThisisanExampleProblem.

The proposed word segmentation algorithm should reintroduce spaces between words and return the following sentence :

This is an Example Problem.

Proposed Solution

Viterbi Algorithm in general, finds the most likely sequence of hidden states given a sequence of observations. It has been successfully used to solve various problems in speech recognition, speech synthesis, computational linguistics, etc.

In this paper, we have created a Probabilistic Unigram

language model of single-word frequencies using a frequency dictionary. The frequency dictionary is built by counting the word frequencies in the complete text of Pride and Prejudice. Building a frequency dictionary in this way ensures that the words in the test samples are covered. Also, Zipf's law ensures that the word frequencies for any literary work are similar for a given language. Hence, minimizing the chances of these calculated probabilities being incorrect. Eg. The word 'the' is still shown as the most used word by this method. Word boundaries are detected by performing segmentation with maximum total score i.e. product of single-word probabilities. The cost of computation for this approach is quiet low, and still it manages to produce great accuracy. This dynamic programming algorithm computes in O(n) asymptotic time complexity and does not backtrack.

Project started off with removing spaces(preserving the punctuations) from a chapter of "Pride and Prejudice" ebook provided by project Gutenberg. The proposed Viterbi approach was then applied to each line in the ebook and hidden words were segmented. Next step was to insert the spaces back into the text and storing it in an output file. The output file was then compared with the original text for cosine similarities between the words in the original text and the extracted hidden words from the output file. The algorithm's approach can be illustrated through the given pseudocode:

Pseudocode of Viterbi Algorithm :

Input to the algorithm is a string- input_string

```
1. Define max_dict_word_length
2. Initialize List<float> probScores -> [1.0]
3. Initialize List<int> lastWordStartIndex -> [0]
4. for i = 1 -> len(input_string)
     probScore <- max_score_combi(substringTo_i)
     lastWordStartIndex <-
                 (LWSI_for_max_score_combi)
5. List <String> words = new List<String>
6. size = length_of_input_string
7. while size > 0
     words <- subString (lastWordStartIndex(size)
                                 to size)
     size = lastWordStartIndex(size)
8. return reverseList(words)
```

Pseudocode for Viterbi Approach

The algorithm tries to detect word boundaries while iterating through each character in the input string:

1. Two lists namely, probScores and lastWordStartIndex are maintained.
2. Score for a boundary combination is calculated by multiplying single-word probabilities.
3. Each possible word boundary combination from the preceding_characters to the current_index (in the input_string) is evaluated to a score. Here, a preceding character can be at a maximum <longest_word_length> indices away from current_index.
4. The score for the combination with maximum score (for a particular index) is stored in the probScores list. Whereas, the starting index of the last word in this combination is stored in lastWordStartIndex list.
5. This continues till the last index of the input string.
6. After the lists are ready, next step is to extract the words from the input string following these boundaries.
7. An empty list (say- words[]) is created to store the extracted words.
8. We will extract each word starting from the last word.
9. Starting index of the last word in the string is checked from the lastWordStartIndex list. This word is stored in the words list.
10. Similarly, all the other words are extracted from the input_string and are appended to the words list
11. Once all words are extracted, reverse of the words list is returned.
12. This returned list contains the correct order of extracted words from the input string.

Experiments/Results/Conclusion

Here we confer the experimental results obtained by applying the proposed approach. The suitability of the proposed approach can be established from the comparison of the resulting text file. For all the experiments Python environment was used.

Cosine Similarity:

The input text file- "PrideAndPrejudiceChapter3.txt" and the resulting file from the experiment were compared for Cosine similarity-

Resulting Cosine Similarity = 0.929 (*approximately*)

where, the size of input file was = 9544 characters(166 lines)

Cosine similarity creates a vector of word frequencies for each document. Similarity in the two documents was then measured by calculating the cosine of the angle between the two vectors generated. 7% of the words were not segmented correctly because the proposed algorithm always gives a higher priority to a longer word in comparison to two small words. Let us take an example-

Suppose the given string is - “Letusmeetafternoon”

And, the expected output is “Let us meet after noon”

The proposed algorithm would give an output as-
“Let us meet afternoon”.

From the above example we can see that combination of small words can get ignored while looking for longer legal words. This drawback accounts for the 7% error. To improve the accuracy we can use a bigram model (markov chain) which includes transition probabilities -
Prob(words | previous_word)

This slight improvement in accuracy comes at the cost of much greater computational cost. Unsupervised learning is also an alternative approach to guarantee better accuracy.

Spaces Count Error Percentage:
This experiment measured the percentage difference in the occurrence of spaces between the two text documents. This experiment when performed on PrideAndPrejudiceChapter3.txt gave the following result:

Spaces Percentage Error = 13.95%

Taking the challenge of preserving the punctuations has resulted in 13% error. In natural language, each punctuation mark abides to specific spacing rules. Double quotes <“> are not followed by a space whereas, a comma<,> is always followed by a space. Such rules are difficult to accommodate in the frequency dictionary. Even after encountering this error the cosine similarity is not much affected as the words are preserved and extracted correctly.

Future Work

There are few improvement scope to make this approach even more efficient:

Use of a Bigram model:

In the proposed solution a Unigram model is used. By using a bigram model results can be improved. This will make sure that context is maintained in cases where multiple sequences are possible for the same character sequence. The trade-off here is that using a Bigram model will increase the time complexity of the solution.

Avoiding the underflow problem in joint probability calculations :

The probability of occurrence is extremely small for most of the words in the dictionary. This means that when we have a sequence of several words, the probability of all of them occurring is-

$$P(a,\ b,\ c) = P(a) * P(b) * P(c)$$

This is an even smaller number. If the length of a sentence is large, there is a huge chance of encountering the underflow problem. Due to a lack of available bits to handle such small numbers, their probabilities can go down to zero. This makes word sequences that have very different likelihoods appear equally likely, since probabilities of almost all word sequences are very small.

An effective solution would be to find the maximum of the sum of the probabilities’ logs.

Using smoothing to Prevent non-corpus words from disturbing the rest of the sentence.
It is not feasible to include entire dictionaries into the program. Encountering a non-corpus word can blow up the entire sentence. Smoothing can be used to handle this problem. Whenever, a new word that is not in the dictionary is seen, an extremely small probability is assigned to it. This prevents the algorithm from breaking down. By using smoothing, such cases are handled effectively making the solution more suitable for the real world.

References

Bird, Steven, Ewan Klein, and Edward Loper (2009), Natural Language Processing with Python, O'Reilly Media

Python Software Foundation. Python Language Reference, version 2.7. Available at http://www.python.org

Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

Austen, MacKaye. (2008). Pride and Prejudice. Urbana, Illinois: Project Gutenberg. Retrieved April 1, 2017, from www.gutenberg.org/ebooks/37431

David M. W. Powers. 1998. Applications and explanations of Zipf's law. In Proceedings of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning (NeMLaP3/CoNLL '98). Association for Computational Linguistics, Stroudsburg, PA, USA, 151-160.

http://www.phontron.com/slides/nlp-programming-en-01-unigramlm.pdf