Computer Science 182 **Fall 2015**

Final Report: Text Segmentation

Cecilia Zhou & Haoqing Wang

1 Motivation

Suppose we are attempting to train a machine to understand human speech. Before we even begin to attempt to parse through the words spoken, we must determine what the words are. It may be possible to determine the series of letters spoken, but many people speak in a manner such that the spaces between their words cannot be reliably discerned. Thus, we are led to the problem of deciding where to place spaces in a string of text to identify the words.

This problem appears in other situations as well. For example, in several Asian languages such as Chinese, spaces are not placed in written text, except between punctuation. Just as with an unspaced English sentence, the understanding of these sentences begins with first determining what the words are. However, the actual problem of understanding these sentences lies outside of the scope of our project. The problem of determining what these sentences mean as well as the problem of selecting a sentence through attempting to understand it require knowledge of natural language processing beyond our present capabilities, so we did not attempt to address these areas. These problems are tackled in much of the literature we read, particularly addressing Chinese.

2 Problem Statement

Formally stated, we are solving the following problem: Given a sentence composed of words in English with its spaces removed, where do we insert spaces? This is in essence a two-part problem. First, we must view the insertion of spaces as a constraint satisfaction problem, where the goal is to place spaces in a manner such that each word between a pair of consecutive spaces is a valid word in the dictionary. We call a series of spaces satisfying these constraints a *valid spacing*.

In many cases, there will be multiple such valid spacings. Thus, this leads us to the second

component of our problem, which is to decide in the event of multiple valid spacings which spacing is the intended one. That is, which spacing is the most probable in the context of not only the dictionary itself but also in the context of logic and meaning of the language in which the words belong. This amounts to finding a method to assign probabilities to each valid spacing, and then either choosing the spacing with the highest probability or making a random choice using the probabilities we assign.

Note that there are several implicit problems here that we will choose to ignore. One key assumption we make here is that there will be one correct, intended segmentation of a sentence. This is true in the overwhelming majority of English sentences, which cannot be interpreted in multiple ways. However, in several Asian languages, native speakers themselves may segment a sentence in different ways, with each segmentation being valid and interpretable. Researchers have devised methods for dealing with this, such as requiring algorithms to match a certain proportion of segmentations made by a large set of native speakers (Wu and Fung 1994). To avoid these issues, we will implement and test our algorithms with unambiguous English sentences.

3 Algorithms

3.1 Finding Valid Spacings

We first consider the constraint satisfaction portion of our problem. Suppose the text to be a string S of length n. We create $n - 1$ variables representing the positions between each consecutive pair of characters. Each variable has a domain of binary values, equaling 0 if a space should not be inserted at that location and 1 if a space should be inserted. For our factors, we consider the spaces contained within any substring of S of length W, where W is the maximum allowable word length; we set $W = 20$ as a default. A factor is satisfied if between every consecutive pair of spaces, the letters in-between form a valid word in English.

In our implementation, we store our variables as a list of binary values of length $n - 1$ and our factors as a list of binary values of length $n - W + 1$. We have a class that stores the string and its length as well as a particular (potentially partial) assignment for the variables. The class also provides a method to check whether the current assignment is consistent. To check whether a particular substring matches a word in the English language, we are using the PyEnchant library, which is a spell-checking library based on

the C Enchant library. Upon running our code, we found that the inclusion of many short words in the PyEnchant library was problematic for our algorithms. Thus, we removed words that were one or two letters long other than the ones that commonly appear in English.

To solve the constraint satisfaction problem, we first implemented an uninformed search solution using depth-first search. As there are 2^n possible spacings, our algorithm takes $O(2^n)$ time. We do not implement forward-checking, as we do not believe it would benefit our algorithm.

In the interest of improving our time complexity, we then implemented a left-to-right dynamic programming solution. For each variable, we suppose there is a space at that location and find the possible locations for the previous space. That is, we find the possible words that end at that variable. Then, we store these locations for each variable in an array. As we suppose a maximum word length, the runtime for this step of the algorithm takes $O(n)$ time. We can then find all valid spacings by looking at the possible previous spaces of the final position, and then walk backwards from there using a depth-first search until we reach the beginning of the text.

3.2 Choosing the Correct Spacing

After obtaining the list of possible segmentations from one of our algorithms, we now need to select the one that we believe is most likely to be the correct segmentation. As previously discussed, this entails finding a method of assigning probabilities to each segmentation.

We began by attempting a relatively naive approach: We determine the frequency of each word in the English language. Then, for each segmentation, we find the product of the frequencies of the words that appear. We consider the segmentation with the highest probability to be the best segmentation. To gather data about frequencies, we pre-process a sample text and store how often each word in the text shows up in a dictionary with words as keys and frequencies are values. Every word not appearing in the text is defaulted to be the same frequency as the word appearing least frequently. There is a noticeable flaw here in that segmentations with more words are much lower in probability than ones with fewer words, but we believe this is reasonable as the segmentations with fewer words have longer words, which are unlikely to occur by chance.

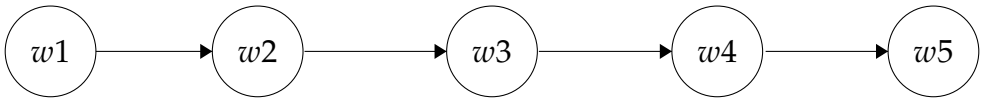
In our implementation, we obtain frequencies by examining a text and computing the frequency of each word. For words not in the text, we assign some low, base frequency. The default text we use is Ernest Hemingway's "A Farewell to Arms" as it is a fairly long text with relatively standard language. Upon running the above on a paragraph of text of approximately 40 words, we find that this actually takes a long time, since we have to find all possible segmentations and then process each one individually.

One method to improve runtime is to only remember the possible previous space that creates the word that appears most often, taking a greedy aproach instead. In this manner, we avoid finding all possible segmentations in the first place and having to iterate through all of them afterwards as well. Instead, we just compare the frequencies of the possible words ending at a given position and greedily selecting the most likely one.

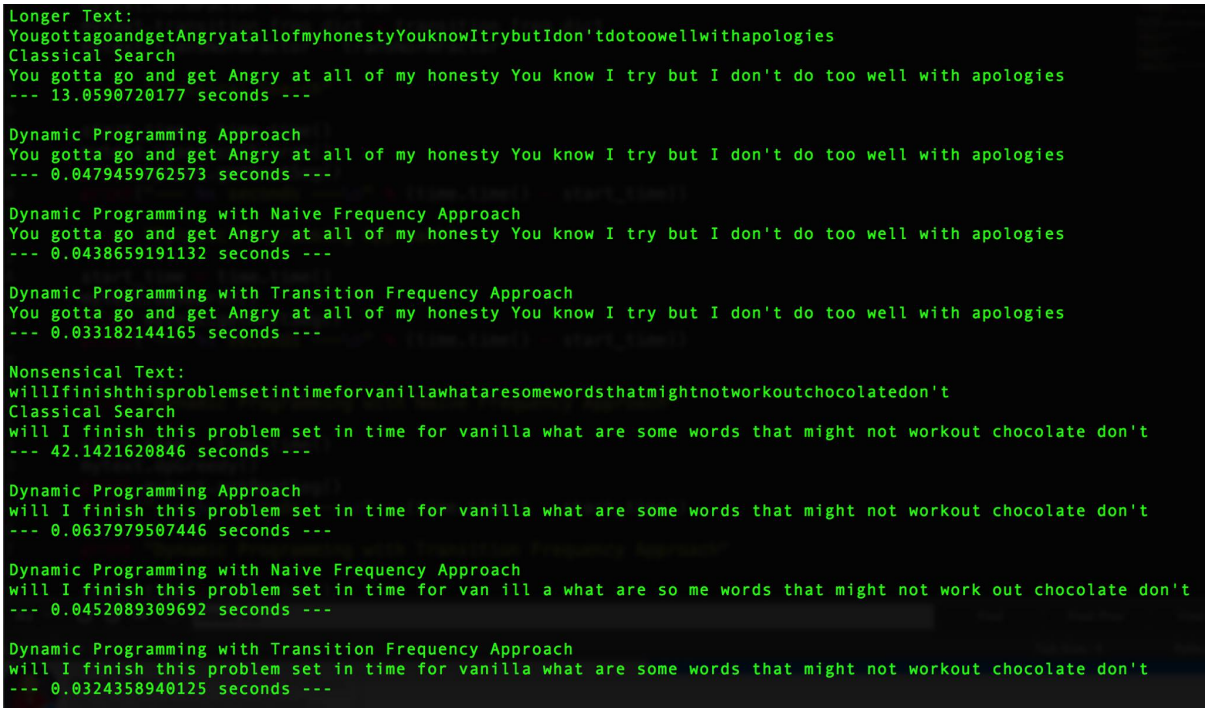
Here, in the greedy algorithm, we lose the preference of long words over shorter ones, as generally shorter words will have greater frequency than longer words. For example, the text "vanilla" is kept as a single word by our first algorithm but split into "van ill a" by our greedy algorithm. We would like to find an alternative method that considers the likelihood of words appearing together (and making sense) rather than merely the likelihood of the words themselves.

To account for the likelihood of combinations of words, we attempt a method in which we examine the probability of a transition: That is, for words that appear as AB in S, we estimate the frequency of the transition substring AB in the text, rather than the frequencies of A and B. Again, we assign some low frequency to transitions that do not actually occur. Here, we could use a standard approach, which was to multiply these frequencies after computing all segmentations, or a greedy approach, which took the next word that was most likely to appear in the dynamic programming step. We choose to take the greedy approach for the purpose of improving runtime.

This is a Markovian model, as we only consider the probability of the word based on the previous word, and we ignore the dependencies of the word on words before the most recent one. If we have a five word sentence, we can model this as shown below.



4 Testing and Observations



The above Python output displays the difference between our algorithms running on two different text sequences. The first text segment, labelled "Longer Text," consists of text from another writing by Ernest Hemingway, "The Old Man and the Sea." It is reasonable to assume that our algorithms will perform relatively well with this example, which is indeed the case. However, this example illustrates the drastic difference between the runtimes of our classical search solution using depth-first search and our dynamic programming solution. Additionally, we see that considering frequencies greedily gives a slight improvement over the basic dynamic programming solution.

The second example is more interesting as we provided a text example that was relatively nonsensical, although the transitions between words were mostly reasonable. However, contrasting the output from the dynamic programming algorithm considering word frequency with the output from the dynamic programming algorithm considering transitions between words confirms our hypotheses regarding the two algorithms. First, we see the effect of the algorithm on the word "vanilla," which is split in the naive frequency approach that favors shorter words. This also occurs to the words "some" and "workout," for example. It is interesting to note that while we would prefer "vanilla" and "some" to remain as one word, we would not do the same for "work out." However, we may be

able to attribute this error due to the fact that the phrase “work out” may not appear at all in the work we reference for frequencies, leading the algorithm to avoid that possibility.

We also tested our accuracy using a similar text, ”The Old Man and the Sea”. We consider this text similar because it is also written by Ernest Hemingway, and we use his ”A Farewell to Arms” as our base text to get frequencies. Using individual frequencies of words, we get a 63.04% accuracy rate, where a sentence is considered correct if all the words match the original sentence, and we divide correct sentences by total sentences. Using transition frequencies, we get a 74.74% accuracy rate. The entire process, including both algorithms, takes about 36.05 seconds to process for about 800 lines of text in the original document.

We then test our algorithm on the same text as the base text, ”A Farewell to Arms” and see that using individual frequencies of words, we get a 71.28% accuracy rate. Using transition frequencies, we get a 81.19% accuracy rate. The entire process, including both algorithms, takes about 229 seconds to process for about 15000 lines of text in the original document.

5 Discussion

Upon examining the performance of our system, we see that there are several areas in which we could find improvements. First, the implementation of the algorithms could be improved. For example, for word frequencies, we could have used a larger dictionary. In particular for the transitions case, the size of the present dictionary results in our algorithm rewarding the presence of any transitions present, assigning most transitions the default probability for transitions that do not appear in our dictionary as most transitions we encounter will most likely not appear in our dictionary. One method to create a much larger dictionary would be to implement a crawler over social media to find these frequencies.

Beyond these implementation details, resulting improvements lie with the algorithm itself. We would have liked to have devised a method of selecting the most probable segmentation given the list of valid segmentations in a more rigorous manner. However, we were unable to find one that was reasonable to implement given the resources and knowledge we had. Such a method would like outside the scope of our project.

Another interesting idea that we were unable to tackle was to perform text segmentation without the use of a dictionary. This was discussed in some of the literature we examined, with the use of statistical frequencies of letters in relation to each other used to create the segmentation instead (Van Anken, 2011). This would be an interesting alternative that would change the methods we used entirely, also requiring a large body of supporting text to derive the "true" relationships between letters. Ultimately, we found that the use of the dictionary was simply more practical, requiring a much smaller text to use as our constraint satisfaction problem was solved based on the dictionary alone.

Appendix 1: Example

We will use "thequickbrownfoxjumpsoverthelazydog" as our example text that should have a valid segmentation, and "thequickbrownfoxjumpsovertheazydog" as our example text with no valid segmentation. See the code in example.py.

First we create an instance of our NoSpaceText class with the text and give the user an option to parse a base text by calling *helpers.parseBaseText*, and if they choose to not give a text, we use "A Farewell to Arms" by Ernest Hemingway by default. We then gather data about our base text. We use *helpers.getFreq(basetext)* to get naive frequencies of individual words in the base text, and *helpers.getTransitionFreq(basetext)* to get transition frequencies of words in the base text. Here each pair of words has a frequency rather than each individual word having its own frequency.

All of the following functions are methods in the NoSpaceText class in models.py.
We then use classical search (*classicalSearch()*) to find all possible segmentations, and naive probabilities to get the best segmentation. We then use dynamic programming approach to get all possible segmentations (*dpSearch()*) and again naive frequencies to get the best segmentation. Then we use two different greedy approaches: the first just uses naive frequencies (*dpGreedy()*) and picks the word with the highest frequency as we find possible previous spaces, and the second uses transition frequencies(*dpGreedy(transFreq = True)*) and picks the word with highest transition frequency. All of the methods return a list of possible segmentations, where the latter two just return a list of size one with the one that they find to be the most probable. We then call (*getBestSeg()*) that takes those possible segmentations and does the multiplication of naive frequencies.

Finally, we run *dpGreedy(transFreq = True)* on the text we mentioned above that shouldn't

return any valid segmentations, and we find it indeed doesn't and returns *None*.

Appendix 2: Using Our System

Run *python example.py* in the terminal to see the results of the example we run through in Appendix 1.

Customize your dictionary by modifying the file "twoletter.txt", which currently holds two-letter words allowed in scrabble but are generally not commonly used words. Add words (not necessarily length 2) to ignore in "twoletter.txt", and run *helpers.modifyDictionary()* if you want to remove them from your PyEnchant dictionary. The words removed and added in *modifyDictionary()* are what we use.

We also provide three other files, *testSameText.py, testSimilarText.py, testCompareAlgs.py*.

The first runs the dynamic programming algorithm using transition frequencies on the same text as the base text, assuming that the user chooses to use the default base text. The second runs that algorithm on a similar text "The Old Man and the Sea", written by Ernest Hemingway as is the default base text. The third compares the four different algorithms' run-times and accuracy for three different texts. The first text is the same as in example.py, the second is an excerpt from the song "Sorry" by Justin Bieber, and the third is just a nonsensical sentence that illuminates some interesting behavior of our algorithms discussed earlier.

If one wants to run our algorithms on different texts, just run *python testYourOwnText.py* in the the terminal, and input the text file of your choice when prompted and then input the base text file of your choice when prompted as well.

Appendix 3: Collaborators

Haoqing Wang:

- implemented structure for constraint satisfaction problem
- implemented classical search
- manual testing with different phrases to check basic functionality

Cecilia Zhou:

- implemented dynamic programming approach

- implemented greedy dynamic programming approaches using naive frequencies and transition frequencies
- implemented parsing code for base texts and input texts

Together:

- reading literature and brainstorming algorithms
- debugging
- writing tests

References

W.J. Teahan, R. McNab, Y. Wen, and I. Witten, "A Compression-based Algorithm for Chinese Word Segmentation" in Computational Linguistics (Volume 26 Issue 3), 2000.

J. Van Aken, "A Statistical Learning Algorithm for Word Segmentation" in arXiv, 2011.

D. Palmer, "A Trainable Rule-Based Algorithm for Word Segmentation" in ACL '98 Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics, 1998.

R. Sproat, W. Gale, C. Shih, and N. Chang, "A Stochastic Finite-State Word-Segmentation Algorithm for Chinese" in Computational Linguistics (Volume 22 Issue 3), 1996.

D. Wu and P. Fung, "Improving Chinese Tokenization with Linguistic Filters on Statistical Lexical Acquisition" in Fourth Conference on Applied Natural Language Processing (ANLP-94), Stuttgart, Germany, p. 180-181, 1994.