NoSpace: An Online Linear-Time Algorithm for Optimal Segmentation of Noisy Text Sentences

Nader Al-Naji

Abstract

Eliminating the need to type spaces when texting can dramatically increase typing speed. Despite this, however, there do not appear to be any schemes capable of fixing errors and segmenting (inserting missing spaces) efficiently and reliably. Thus, we present an online, linear-time algorithm that takes in unsegmented and/or noisy strings, such as "jellomufarling", and corrects/segments this input "optimally", outputting, for example, ⟨"hello","my","darling"⟩. Optimality is defined with respect to an arbitrary "scoring function" that maps lists of strings to the real numbers, and the optimal segmentation is the list of strings that maximizes this scoring function, where the domain is all possible segmentations of the input. After describing the algorithm, we measure its performance using a common scoring function and find that it achieves an accuracy of over 97% on noiseless input and 95% on noisy input. Finally, we discuss our positive experience implementing and using the algorithm on a Nexus 4 phone.

1. Introduction

Consider the following unsegmented string:

"Whatareyoudoingfordinnertonight?" (1)

A person can probably easily divide the unsegmented string into a list of strings that form a sentence, such as:

⟨"What", "are", "you", "doing", "for", "dinner", "tonight", "?"⟩ (2)

but how to go about doing so in an "optimal" and efficient way is not immediately obvious. If we suppose further that our goal is to read in *noisy* input as in the perturbed version of (1) below, correct errors in the input, and segment it optimally and efficiently:

"Shataeeyoidoongfordinmrryonigjt?" (3)

eg. to take as input (3) and output (2), then the problem becomes more difficult still.

But solving such a problem can be incredibly useful. To see this, simply consider the average word length in English, usually taken to be five characters. If we can save a space after every word, this means we can increase typing speed by almost 17%. And with the rapid growth of mobile devices, much communication has been moving to cell phones and tablets, increasing both the cost of each keystroke, and the error rate of input. Thus, an algorithm that can segment text optimally and efficiently, and that is robust to errors, is more useful today than ever before.

Even on today's latest mobile devices, however, one will be hard-pressed to find an autocorrection engine capable of fixing errors and segmenting efficiently. While much research into autocorrection technology is done privately, forcing us to rely on speculation as to what the state-of-the-art really is, the results of such research are public. In particular, we can firmly say that neither the iPhone autocorrection engine (on iOS6), nor the Android autocorrection engine (on Android 4.1 Jelly Bean) provide such optimal full-sentence

segmentation functionality. The reason for this lack of functionality is unclear, but it could very well be due to the inherent complexity of existing state-of-the-art segmentation algorithms.

The main contribution of this paper is an online, linear-time algorithm that will compute the "optimal" segmentation (to be defined) of noisy sentences, that is fast enough to be run in real time on a mobile device. We implement a proof-of-concept on an Android Nexus 4 phone.

In this paper we begin by formalizing our notion of optimality, treating segmentation as an optimization problem. We then begin to describe our segmentation algorithm in full detail by starting with a simple version of the algorithm and steadily expanding it to get to the full online algorithm. After we've described the algorithm, we analyze the running time and the space usage. Then, we discuss the algorithm's accuracy and performance on a de-segmented corpus of ebooks, in addition to discussing our first impressions of the algorithm as implemented on a Nexus 4 phone. Finally, we provide concluding remarks and a summary of prior work.

2. Formalizing Optimality

In this section we formalize the goal of our algorithm and what it means for a segmentation to be "optimal". We consider input without noise (so input such as (1) above being split into (2) above) until section 5.

Our algorithm will take as input a "scoring function" g that maps lists of strings to real numbers. Formally, let S denote the (infinite) set of all lists of strings (the domain of g):

$$S \equiv \{\langle w_1, ..., w_n \rangle \text{ such that } w_i \text{ is a valid string } \forall\ i \in \{1, ..., n\},\ n \in \mathbb{N}\} \tag{4}$$

where $\langle w_1, ..., w_n \rangle$ denotes a list of elements and w_i denotes the i_{th} element of this list. In words, S is simply the set of all possible lists of strings, and it will form the domain of our scoring function, g. Note that many of the lists in S will not correspond to valid English sentences but that's OK.

Having defined the domain of g, we now formally define g itself:

$$g : S \to \mathbb{R} \tag{5}$$

So g is simply a function or mapping that takes lists of strings into the real numbers, and our algorithm will take such a function as input. Note that g can be completely arbitrary but, in the case of segmentation of natural language, a higher value of g should be at least loosely correlated with a more intuitive segmentation. So, for example, $g(\langle "hello", "world" \rangle)$ should be greater than $g(\langle "hel", "lowo", "rld" \rangle)$.

Having defined g, we now aim to define "optimality". Given a (noiseless) input string w and a scoring function g, it will be the goal of our (simplified) algorithm to find the segmentation of w that maximizes g. So let S_w denote the (finite) set of all lists of strings that, when concatenated, yield w. Formally,

$$S_w \equiv \{\langle x_1, ..., x_n \rangle \in S \text{ such that } x_1 || ... || x_n = w\}. \tag{6}$$

where $x_1 || x_2$ denotes the concatenation of x_1 with x_2. So S_w is simply the set of all lists of strings that, when concatenated, yield w. As an example, $S_{"he"}$ would contain $\langle "h", "e" \rangle$ and $\langle "he" \rangle$. We say a list of strings l is a "segmentation" of w if and only if $l \in S_w$. Thus,

we want to find the segmentation of w that maximizes g. Formally, we simply want:

$$argmax_{x \in S_w} \; g(x) \tag{7}$$

Or, in words, we want to consider all possible segmentations of w and choose one that maximizes g. A segmentation $s \in S_w$ will be called "optimal", then, if and only if $g(s) \geq g(s')$ for all $s' \in S_w$.

So to summarize, this simplified version of our algorithm will take as input a string w and a function g that maps lists of strings to real numbers. The algorithm will then compute an "optimal" segmentation of w, or equivalently a segmentation of w that maximizes g.

3. Restrictions on on Scoring Function

Our algorithm needs to make three restrictions on the scoring function g in order to be able to run in linear time. Here we describe our three restrictions and then provide an example of an intuitive scoring function that satisfies these restrictions.

3.1. Assumption 1: Length Restriction

We impose a maximum length on strings in our output segmentation. That is, we insist that $g(\langle w_1,...,w_n \rangle) = -\infty$ if $|w_i| > k$ for any $i \in \{1,...,n\}$ where $|w_i|$ denotes the length of string w_i. In words, this restriction insists that sentences with words longer than k characters will have very, very low scores. This restriction makes sense, since most languages have a natural cap on word length. For English, a reasonable cap appears to be around 25 characters.

3.2. Assumption 2: Markov Property

With regard to our scoring function, g, we assume that:

$$\begin{aligned} g(\langle a_1,...,a_{n_1},y \rangle) &\geq g(\langle b_1,...,b_{n_2},y \rangle) \\ \rightarrow g(\langle a_1,...,a_{n_1},y,c_1,...,c_{n_3} \rangle) &\geq g(\langle b_1,...,b_{n_2},y,c_1,...,c_{n_3} \rangle) \end{aligned} \tag{8}$$

This second restriction is a little more mysterious but we will see exactly how it comes into play when analyzing the run-time of the segmentation algorithm in full detail. For now, it will be enough to show that a commonly-used scoring function in natural language processing, namely the log probability of a segmentation under a first-order Markov model (We will describe this scoring function below.), satisfies this constraint, and we will show this.

3.3. Assumption 3: Fast Update

We assume that given only $g(\langle x_1,...,x_m \rangle)$, x_m, and x_{m+1}, we can compute $g(\langle x_1,...,x_m,x_{m+1} \rangle)$ in $O(k)$ time where k is the maximum length of x_m and x_{m+1} by restriction (3.1). This essentially assumes that we can make incrimental updates to values of g quickly, without having to run over an entire list of strings more than once. This assumption isn't strictly necessary to obtain a polynomial-time algorithm but it can be shown that it does decrease the run-time of our algorithm by a factor of n where n is the number of characters in the string we are segmenting, so it is worth considering. Is it justified? One can certainly find many scoring functions that don't satisfy this property but for now it will be enough, again, to show that a commonly-used scoring function in natural language processing satisfies this property.

3.4. Justification

The above assumptions are all we need in order to develop an algorithm that finds an optimal segmentation in linear time. However, before we proceed to describe the algorithm in detail, we will first provide a common scoring function, the log probability under a first-order Markov Model, and show that it satisfies all the assumptions above. The fact that such a widely used scoring function satisfies these constraints will be our justification that limiting our scope *only* to functions that satisfy these constraints is reasonable.

Under a first-order Markov Model, we have:

$$\begin{aligned} g(\langle w_1,...,w_n\rangle) &= \log(P(\langle w_1,...,w_n\rangle)) \\ &= \log(P(w_n|w_{n-1})) + \log(P(w_{n-1}|w_{n-2})) + ... + \log(P(w_1)) \end{aligned} \quad (9)$$

where the probabilities above, such as $P(w_n|w_{n-1})$, are determined by crawling or "training" on a real text corpus and keeping track of the number of times the tuple $\langle w_n, w_{n-1}\rangle$ occurred relative to all other tuples in the corpus. Note that we use log probabilities to prevent underflow (imprecision that arises due to multiplying together many numbers that are less than one) during computations. Clearly this is a very natural scoring function to use but we now show that it also satisfies our three assumptions.

3.4.1. Length Restriction Satisfied

If we make k sufficiently large, then no strings in the training corpus with length greater than k will ever be encountered. Thus the probability of encountering a list of strings l where one or more of the strings in l have length greater than k will be zero, implying $g(l) = log(P(l)) = log(0) = -\infty$, which is consistent with the restriction.

3.4.2. Markov Property Satisfied

We simply manipulate expressions to show that this property holds. For conciseness below, let A denote $a_1,...,a_{n_1}$, let B denote $b_1,...,b_{n_2}$, and C denote $c_1,...,c_{n_3}$ with y being a single string.

$$\begin{aligned} g(\langle A,y\rangle) &\geq g(\langle B,y\rangle) & \\ \rightarrow \log(P(A,y)) &\geq \log(P(B,y)) & \text{definition of } g \\ \rightarrow P(A,y) &\geq P(B,y) & \text{probabilities positive, log monotonic} \\ \rightarrow P(y|A)P(A) &\geq P(y|B)P(B) & \text{definition of conditional probability} \\ \rightarrow P(y|A)P(A) - P(y|B)P(B) &\geq 0 & \\ \rightarrow P(C|y)(P(y|A)P(A) - P(y|B)P(B)) &\geq 0 & \text{OK because } P(C|y) \text{ is positive} \\ \rightarrow P(C|y)P(y|A)P(A) &\geq P(C|y)P(y|B)P(B) & \\ \rightarrow P(A,y,C) &\geq P(B,y,C) & \text{definition of conditional probability} \\ \rightarrow \log(P(A,y,C)) &\geq \log(P(B,y,C)) & \text{probabilities positive, log monotonic} \\ \rightarrow g(\langle A,y,C\rangle) &\geq g(\langle B,y,C\rangle) & \text{definition of } g \end{aligned} \quad (10)$$

Thus, the probabilities computed under a second-order Markov model obey this property.

3.4.3. Fast Update Satisfied

Here, we note the recursive nature of the scoring function:

$$\begin{aligned} \rightarrow g(w_1,...,w_n) &= \log(P(w_1,...,w_n)) = \log(P(w_n|w_{n-1})) + \log(P(w_1,...,w_{n-1})) \\ \rightarrow g(w_1,...,w_n) &= \log(P(w_n|w_{n-1})) + g(w_1,...,w_{n-1}) \end{aligned} \quad (11)$$

And thus, given w_n, w_{n-1}, and $g(w_1,...,w_{n-1})$, we can look up $P(w_n|w_{n-1})$ and compute the new value of g in $O(k)$ time.

4. Simplified Algorithm Description

In this section we describe a simplified version of the algorithm. This simplified version takes in a string w and a scoring function g with the aforementioned properties, just as the full algorithm will. It is simpler than the full algorithm, however, because it assumes the string w is "noiseless", namely that w does not contain any errors or mistakes. That is, w is as in (1) above. We provide a high-level description of the algorithm, some pseudocode, and then a further discussion to solidify our understanding. Note in all of what follows, lists and strings are zero-indexed, meaning the first element is always at index zero. Additionally, throughout, let k denote the maximum length of any string in our language under restriction (3.1) on page 3.

The algorithm works by keeping track of two things. The first is simply an index, *end*, into w that sweeps from left to right. That is, *end* takes on values starting at one and incrementing up to $|w|+1$ inclusive where $|x|$ denotes the number of characters in x. As an example, if w were "hello", then *end* would take on values between 1 inclusive and 6 inclusive. In addition to this index, *end*, we keep track of a data structure called the "frontier". The frontier is a three-dimensional list (of characters) data structure that we will use to keep track of the "best" segmentations during the algorithm's execution. Its exact use will be made clear in the algorithm's description below but before describing the algorithm in detail, we state two theorems, proven in the appendix, that will hopefully help the reader understand the frontier a little better. We will also use these theorems to help explain the algorithm itself. In what follows, let $w^{i:j}$ denote a substring of w starting at index i inclusive and ending at index j exclusive, as in $"hello"^{1:3} = "el"$:

Theorem 1: During the algorithm's execution, $frontier[i]$ will contain lists of strings such that if one were to concatenate all of the strings in any of these lists, one would get $w^{0:i}$ for all $i \in [max(0,end-k),end[$. That is, $frontier[i]$ contains *segmentations* of the substring $w^{0:i}$

As an example, if w were "hello", we could have frontier[1] = $\langle\langle "h"\rangle\rangle$ because "h" is a substring of w that starts at index 0 and ends at index 1 exclusive. We could have frontier[2] = $\langle\langle "he"\rangle,\langle "h","e"\rangle\rangle$ since both these lists of strings concatenate to "he", which is a substring of w starting at index zero and ending at index 2 exclusive. We could also have frontier[3] = $\langle\langle "he","l"\rangle,\langle "h","el"\rangle,\langle "hel"\rangle\rangle$ because all of these lists of strings concatenate to "hel", which is a substring of w starting at index zero and ending at index 3 exclusive. This theorem is proven in the appendix but the reader is advised to understand how the algorithm works before trying to undertand the proof.

Theorem 2: During the algorithm's execution, $frontier[i]$ always contains an *optimal* segmentation of $w^{0:i}$ for all $i \in [max(0,end-k),end[$. That is, $frontier[i]$ always contains $argmax_{x \in S_{w^{0:i}}} g(x)$ where $S_{w^{0:i}}$ is the set of all possible segmentations of $w^{0:i}$ for all $i \in [max(0,end-k),end[$.

Note that this theorem implies that if we want the optimal segmentation of w, we can simply return $argmax_{x \in frontier[|w|]} g(x)$ after the algorithm's execution, when $end = |w|$. This theorem is also proven in the appendix but the reader is again advised to understand how the algorithm works before trying to understand the proof.

Theorem 3: During the algorithm's execution, if $end - k > 0$, $frontier[i]$ will be completely empty for all $i < end - k$.

This theorem will help us analyze the algorithm's space usage.

Having noted these theorems and described our data structure (at least minimally), we now summarize the algorithm in words before providing pseudocode. We begin by initializing *end* to one. Then, to keep theorems 1 and 2 true, we add an empty list l to the frontier and add a list containing just the empty string to l. After this initialization, we have $frontier[0] = \langle\langle""\rangle\rangle$. Note that theorem 1 is true because $w^{0:0} = ""$, which is in frontier[0], and theorem 2 is true because the optimal segmentation of $w^{0:0} = ""$ is just "" trivially, which is also in frontier[0]. We now start the main loop of the algorithm. We loop over all integral values of $end \in [1,...,|w|]$ in increasing order. So while $end \leq |w|$:

- We append an empty list to the frontier.
- We then loop over all substrings of w starting at index $end - k$ inclusive and ending at index *end* exclusive (unless $end - k$ is less than 0, in which case we start at index 0 instead). To do this, we define an index $start = max(0, end - k)$ and increment *start* up to *end*. So while $start < end$:
  - Let $s = w^{start:end}$.
  - Recall that by theorem 1, because $start \in [max(0, end - k), end[$ by construction, $frontier[start]$ always contains segmentations of $w^{0:start}$ (lists of strings that concatenate to $w^{0:start}$). Now, because $s = w^{start:end}$, if we add the string s to the end of any of the lists of strings l in $frontier[start]$, we will have a segmentation of $w^{0:end}$, since $l||s$ will be a list of strings whose elements concatenate to $w^{0:end}$, where $||$ denotes appending string s to the end of list l. Understanding this, we will choose the list l in $frontier[start]$ such that $l||s$ *maximizes* g, so let $best\_prev\_list = argmax_{x \in frontier[start]} g(x||s)$. Now, *best_prev_list* is the list in $frontier[start]$ that maximizes $g(best\_prev\_list||s)$. Although we won't show this now, choosing *best_prev_list* in this way guarantees that $frontier[end]$ will eventually contain an *optimal* segmentation of $w^{0:end}$.
  - Recall that $best\_prev\_list||s$ is now a segmentation of $w^{0:end}$, or a list of strings that all concatenate to $w^{0:end}$. We now add $best\_prev\_list||s$ to $frontier[end]$. Note that this keeps us from ever violating theorem 1, because $frontier[end]$ still contains only segmentations of $w^{0:end}$.
  - Increment *start* by one.
- In order to remain memory efficient, we clear all of the elements of $frontier[end - k]$ if $end - k \geq 0$, since we won't need them anymore. So, after this step, if $end - k > 0$, $frontier[i]$ will be empty for all $i < end - k$ and theorem 3 will remain unviolated.
- Increment *end* by one.

By the end of this loop, we will have $end > |w|$. If we believe theorem 2, then, because $|w| \in [max(0, end - k), end[$, $frontier[|w|]$ will contain an optimal segmentation of $w^{0:|w|} = w$. But this is exactly what we were looking for! So we will simply return $best\_segmentation = argmax_{x \in frontier[|w|]} g(x)$, and theorem 2 will guarantee that this will indeed be the *optimal* segmentation of w, or the list of strings that concatenates to w and maximizes g.

Here we provide pseudocode for the algorithm described above. It takes in a noiseless string w, a scoring function g mapping lists of strings to real numbers, and a maximum string length k, all as described. It then returns a segmentation of w that maximizes g.

Note that we use := to denote assignment.

Require: w, a string
$g : S \to \mathbb{R}$, a scoring function that maps lists of strings to real numbers as described above.
k, the length of the longest allowable string in the list of strings to be returned.
Ensure: $argmax_{x \in S_w} g(x)$

$frontier := \langle\langle""\rangle\rangle$ ▷ Initialize $frontier[0] = \langle\langle""\rangle\rangle$ as mentioned.
for $end := 1 \to (|w| + 1)$ **do** ▷ For loops are left-inclusive, right-exclusive.
    $frontier.append(\langle\rangle)$ ▷ Add an empty list to the frontier as described.
    for $start := max(0, end - k) \to end$ **do**
        $s := w^{start:end}$
        $best\_prev\_list := argmax_{x \in frontier[start]} g(x||s)$
        $frontier[end].append(best\_prev\_list||s)$ ▷ Keep Theorem 1 from being violated.
    end for
    if $end - k \geq 0$ **then**
        $frontier[end - k].clear()$ ▷ Empty this list and keep Theorem 3 from being violated.
    end if
end for
return $argmax_{x \in frontier[|w|]} g(x)$ ▷ Optimal segmentation assuming Theorem 2 holds.

We thus conclude our discussion of this simplified version of the segmentation algorithm. While it may not be immediately obvious why it is the case that the algorithm produces the *optimal* segmentation or, alternatively, why it is that Theorem 2 holds, how the algorithm works, at least at a high level, should be clear at this point. If it is not, the reader is encouraged to review the description above as the information will be reused and built upon in later sections.

5. Full Algorithm Description

In the previous section, an algorithm was presented that computes the optimal segmentation in lieu of input noise. In this section, we will extend this simplified algorithm and present a modified algorithm that is capable of incorporating fairly complex noise models into the computation.

Noise models provide assumptions as to the type of noise that might be present in the input. If we were dealing with pre-segmented text, for example, such noise models could look as follows:

- Each character could be perturbed, with probability p, to a letter adjacent to it on the input device. As an example, assuming an American keyboard, "hello my dear" $\to$ "jellp mu desr".
- Each character could be completely missing from the input with probability p. As an example, "hello my dear" $\to$ "ello my dar".
- An extra character can appear at each position with probability p. As an example, "hello my dear" $\to$ "hjello my ddear".
- Characters could be inverted with probability p. As an example, "hello my dear" $\to$ "ehllo my daer".
- Any combination of the above perturbations.

What such noise models all have in common is a set of *perturbations* that could occur in the input and their likelihood of occurring. We will refer to such noise models as

"perturbation" models. Remarkably, incorporating such models into the segmentation computation is incredibly simple, requiring us to add just a couple of lines to our original simplified algorithm description on page 7. Recall in our original simplified algorithm, we add substrings of w to the frontier at each step (on line 5). The way we incorporate a perturbation model into our computation is instead of only adding *substrings* of w to the frontier at each step, we add both substrings *and* their perturbations, forcing the algorithm to consider the noiseless segmentations *in addition to* the noisy ones.

We now formally describe the full algorithm that is robust to noisy input. The inputs to the new algorithm are as follows:

- This algorithm takes an input string w just as before but, unlike before, the string w can contain errors or noise.
- The algorithm also takes in a scoring function as before, but now this scoring function has to be able to account for noise. That is, the scoring function g_w has to accurately account for the likelihood of errors occurring in w. As an example, $g_{"helpoworls"}(\langle "hello", "world" \rangle)$ would have to factor in the likelihood of "hello" $\rightarrow$ "helpo" and "world" $\rightarrow$ "worls", unlike before. From the perturbation models provided as examples, this would essentially mean incorporating the error probability p into the calculation of g. While it won't be discussed here, it can be shown that accounting for noise in g does not cause g to disobey the three restrictions discussed before under the perturbation models mentioned above. That is, if g is a scoring function that obeys restrictions (3.1), (3.2), and (3.3) (page 3), and g_w is a modified version of g that accounts for perterubations such as those mentioned above, then g_w will also obey restrictions (3.1), (3.2), and (3.3).
- Finally, a third function needs to be provided. This function, call it *perturbations*, needs to take in a string s and return a list of strings $perturbations(s)$ that contains all possible perturbations of s that this noise model wishes to consider. As an example, if we are only considering the omission of characters in our perturbation model, then $perturbations("helllo")$ yields $\langle "elllo", "hlllo", "hello", "helll" \rangle$. As another example, if we are only considering two-character inversion in our perturbation model, then $perturbations("ehllo")$ yields $\langle "hello", "elhlo", "ehllo", "ehlol" \rangle$. Note that in both of these cases, the string "hello" was only found as a possibility *after* incorporating our perturbation model into the computation.

Having described the inputs to the algorithm, we now state exactly what the algorithm computes before presenting pseudocode. Recall that the simplified, noiseless version of the algorithm computed $argmax_{x \in S_w} g(x)$ where S_w is the set of all possible segmentations of w (defined in equation (6)). Given a perturbation function, *perturbations*, let:

$$S'_w = perturbations(S_w) \tag{12}$$

denote the set of all possible perturbations of the strings in S_w. Here we are abusing notation and letting *perturbations* act on a set of strings, rather than just a single string, but the result should be clear. We are now considering all of the perturbed strings in S'_w instead of the original strings present in S_w. The goal of the full algorithm, given an input string w, a scoring function g_w, and a perturbation function *perturbations*, is simply to compute:

$$argmax_{x \in (S'_w + S_w)} g_w(x) \tag{13}$$

where $+$ denotes set addition. It is simply computing the optimal segmentation where the set of segmentations to consider has expanded to include both segmentations of w *and* perturbations of these segmentations.

We now present pseudocode for the full algorithm followed by a discussion of how it differs from the original simplified version.

Require: w, a string
g_w, a scoring function that maps lists of strings to real numbers, factoring in a perturbation model, as described above.
perturbations, a function that takes a string and returns a list of all perturbations of the string, according to a perturbation model.
k, the length of the longest allowable string in the list of strings to be returned.
Ensure: $argmax_{x \in (S_w + perturbations(S_w))} g(x)$

1: $frontier := \langle \langle "" \rangle \rangle$
2: **for** $end := 1 \rightarrow (|w| + 1)$ **do**
3: $\quad frontier.append(\langle\rangle)$
4: $\quad$ **for** $start := max(0, end - k) \rightarrow end$ **do**
5: $\quad\quad s := w^{start:end}$
6: $\quad\quad best\_prev\_list := argmax_{x \in frontier[start]} g_w(x||s)$
7: $\quad\quad frontier[end].append(best\_prev\_list||s)$
8: $\quad\quad$ **for** $s' \in perturbations(s)$ **do** ▷ Added this line.
9: $\quad\quad\quad best\_prev\_list := argmax_{x \in frontier[start]} g_w(x||s')$ ▷ Added this line.
10: $\quad\quad\quad frontier[end].append(best\_prev\_list||s')$ ▷ Added this line.
11: $\quad\quad$ **end for** ▷ Added this line.
12: $\quad$ **end for**
13: $\quad$ **if** $end - k \geq 0$ **then**
14: $\quad\quad frontier[end - k].clear()$
15: $\quad$ **end if**
16: **end for**
17: **return** $argmax_{x \in frontier[|w|]} g_w(x)$

Above, all we've done is add lines 8-11 to the original simplified algorithm. While it may not be immediately clear that this algorithm produces the optimal segmentation (namely, $argmax_{x \in (S_w + S'_w)} g_w(x)$), the intuition should be clear: after adding the substring s to the frontier (line 5), we simply add all of s's perturbations as well (line 8).

The keen reader may note that the Theorems 1 and 2 will not trivially hold for this modified algorithm. In particular, Theorem 1 will break because $frontier[i]$ will not necessarily contain only segmentations of $w^{0:i}$, and Theorem 2 will break because our notion of optimality has changed slightly. To remedy these issues, we provide two new theorems that expand on Theorems 1 and 2 to cover the modified algorithm:

Theorem 4: During the algorithm's execution, $frontier[i]$ will always contain segmentations of $w^{0:i}$ and segmentations of *perturbations* of $w^{0:i}$ for all $i \in [max(0, end - k), end[$. That is, $frontier[i]$ will contain lists of strings such that if one were to concatenate all of the strings in any of these lists, one would get either exactly $w^{0:i}$, or something in $perturbations(w^{0:i})$ for all $i \in [max(0, end - k), end[$.

Here all we've done is expanded the constraint on what the frontier contains to cover *perturbations* of substrings of w in addition to just pure substrings. We prove this theorem in the appendix but it should be pretty clear that this theorem holds after looking closely at the pseudocode for the full algorithm.

Theorem 5: During the algorithm's execution, $frontier[i]$ always contains an *optimal* segmentation of $w^{0:i}$ for all $i \in [max(0, end-k), end[$. That is, $frontier[i]$ always contains $argmax_{x \in (S'_{w^{0:i}} + S_{w^{0:i}})} g_w(x)$ where $S'_{w^{0:i}} + S_{w^{0:i}}$ is a set that includes all possible segmentations of $w^{0:i}$ *and* all possible *perturbations* of these segmentations for all $i \in [max(0, end-k), end[$.

It should be clear that all we've done here is changed our notion of optimality to encompass perturbations of substrings of w *in addition to* noiseless substrings. Taking this theorem into account, it should be clear why the algorithm returns $argmax_{x \in frontier[|w|]} g_w(x)$ on line 17. A proof that this theorem holds for the full algorithm is provided in the appendix.

Considering theorem 5, along with the fact that the algorithm returns $argmax_{x \in frontier[|w|]} g_w(x)$ on line 17, we conclude that the full algorithm does indeed compute the optimal segmentation.

6. Online Version of Algorithm

Now that we fully understand the full offline algorithm, we introduce an online version before discussing runtime. The online version of the algorithm we present and discuss here processes single characters at a time rather than taking in an entire string at once. This can be very useful, since it allows the algorithm to run *while a person is inputting text*, making use of time that would otherwise be wasted. This, as we will see when analyzing runtime, would potentially allow one to use much more complex error models without having to worry about increasing the running time too much. To be clear, the online version does the exact same amount of work as the offline version, and comes to the exact same answer; the only difference is that the online version can proceed incrementally, before knowing the full string w, while the offline version cannot. Namely, as we will also show when we discuss running time, if the goal is to segment a string w, known a priori, as fast as possible, it makes no difference whether one calls the offline algorithm on w, or calls the online version on each individual character of w.

We now present the online version of the algorithm. For the online version, we keep two global variables that persist before and between calls to our algorithm: a string w and a $frontier$, where the $frontier$ will be a data structure exactly as in the full algorithm before. Before making a call to the online algorithm for the first time, we initialize w to the empty string (different than before), and we initialize $frontier = \langle\langle\langle ""\rangle\rangle\rangle$ (exactly as in line 1 of the full algorithm before). After this initialization, we can begin processing characters one at a time as they come in. Because this online version of the algorithm is so similar to the full offline version, we provide pseudocode followed by discussion:

Require: The following are global variables that exist outside the scope of this function and are initialized before this function is ever called.

$w = ""$, the string we are trying to segment. This grows by one character on each call to the function.

$frontier = \langle\langle""\rangle\rangle$, the *frontier* that gets updated on each call to the function, initially containing only a single list with the empty string in it.

Require: The following are variables required as input on each call of the function.

c, a single character

g_w, a scoring function that maps lists of strings to real numbers, factoring in a perturbation model, as described above.

perturbations, a function that takes a string and returns a list of all perturbations of the string, according to a perturbation model.

k, the length of the longest allowable string in the list of strings to be returned.

Ensure: $argmax_{x \in (S_w + perturbations(S_w))} g_w(x)$, where w here is updated on each call.

1: $w.append(c)$ ▷ Different from full offline algorithm.
2: $end := |w|$ ▷ Different from full offline algorithm.
3: $frontier.append(\langle\rangle)$ ▷ Nothing different after this point.
4: **for** $start := max(0, end - k) \rightarrow end$ **do**
5: $s := w^{start:end}$
6: $best\_prev\_list := argmax_{x \in frontier[start]} g_w(x||s)$
7: $frontier[end].append(best\_prev\_list||s)$
8: **for** $s' \in perturbations(s)$ **do**
9: $best\_prev\_list := argmax_{x \in frontier[start]} g_w(x||s')$
10: $frontier[end].append(best\_prev\_list||s')$
11: **end for**
12: **end for**
13: **if** $end - k \geq 0$ **then**
14: $frontier[end - k].clear()$
15: **end if**
16: **return** $argmax_{x \in frontier[|w|]} g_w(x)$

As one can see, the only lines we've changed from the full offline algorithm are the first two; *everything else remains as it was.* The reason for this is that the original offline algorithm *doesn't use* the characters in w that are beyond the index *end*. Thus, feeding them in one at a time, adding them to the end of a global string, and using this shortened string instead of w doesn't change anything. This line of reasoning leads us to our next theorem:

Theorem 6: If one feeds in each character of a string w into the online algorithm in order, the running time, space complexity, and output of the online algorithm will be the same as that of the full offline algorithm when run on w.

This should be clear based on our previous reasoning.

We thus conclude that the online algorithm comes to the same answer as the offline algorithm presented before, namely the online algorithm computes the optimal segmentation if all the characters of a string w are fed into it sequentially. In what follows, we

proceed to analyze the time and space complexity of our algorithms. Thanks to Theorem 6, it will be sufficient to analyze the time and space complexity of the online algorithm and simply claim the the same bounds hold.

7. Running Time Analysis

We now discuss the running time of the online algorithm and the full offline algorithm. In what follows, we take k to be the maximum length of any string in our language by restriction (3.1). We also take p to be the maximum number of elements in $perturbations(s)$ for any string s. For example, our perturbation model could assume that a single character could be replaced by any of a maximum of c adjacent neighbors on a keyboard, as in $perturbations(my) = \{"my","ny","jy","ky","mt","mg","mh","mj","mk"\}$. In this case, we would have $p = c \cdot k$. At the end of the section, we conclude that the overall running time to segment a string w of length n is $O(p^2 \cdot k^3 \cdot n)$, proving that the algorithm is indeed linear in the size of the input string.

Theorem 7: During the execution of the online algorithm on page 11, frontier[i] always contains $O(p \cdot k)$ elements for all i, where k and p are as described above.

This theorem will be useful when analyzing the overall runtime of the algorithm so we provide a proof here.

Proof of Theorem 7:

Looking closely at the pseudocode of the online algorithm, we can see that we only add to the frontier on lines 7 and 10; in particular, we only ever add elements to $frontier[end]$. Further, note that the index end always changes on each call to the function, so when we consider how many elements $frontier[i]$ contains for some i, we need only consider how many elements were added to the frontier on the i_{th} call to the function, when $end = i$. Having taken note of this, in order to prove our theorem, we consider the maximum number of elements that can be added to $frontier[end]$ on any particular call. Starting at line 10, where we add an element to $frontier[end]$, and working outward, we see that line 8 causes $O(p)$ elements to be added to $frontier[end]$ by line 10. Then, we see that line 7 adds a single element to the $frontier[end]$, keeping us at a total of $O(p)$ elements added on each execution of the block of code on lines $5-11$. Finally, working outward once more, we see that the block on lines $5-11$ is executed a maximum of $O(end-(end-k)) = O(k)$ times by the for loop on line 4. Thus, we make $O(k \cdot p)$ total additions to $frontier[end]$ on any given execution of the online algorithm. Recalling that we *only* add elements to $frontier[i]$ on the i_{th} call to the online algorithm (when $end = i$), we thus conclude that $frontier[i]$ always contains a maximum of $O(p \cdot k)$ elements for all i.

Theorem 8: The online algorithm on page 11 does $O(p^2k^3)$ operations on every update where k and p are as defined above.

This theorem gives us the cost of performing a single call to the online algorithm. We prove it below using our result from Theorem 7.

Proof of Theorem 8:

In order to prove this theorem, we first need to figure out how much work is done by lines 6 and 9, where we use argmaxes to find segmentations that maximize g_w. These argmaxes involve looping over all of the elements in $frontier[end]$ and computing $g_w(x||s)$ where x is a list of strings and s is a single string. Looking at Theorem 7, we know $frontier[end]$ contains a maximum of $O(p \cdot k)$ elements, so for each of these lines

we are computing $g_w(x||s)$ a maximum of $O(p \cdot k)$ times. But how long does it take to compute $g_w(x||s)$? Well, our fast update restriction (3.3) (page 3) on g_w states that if we keep $g_w(x)$ around, then it should take $O(k)$ operations to compute $g_w(x||s)$. Of course, the algorithm as described on page 11 makes no mention of caching $g_w(x)$, but we can easily fix this by adding a dummy element to the beginning of all our lists l containing the updated value of $g_w(l)$. So, lines 6 and 9 perform a maximum of $O(p \cdot k^2)$ operations. We now start on line 9 and work our way outward to get the running time of the whole algorithm. Moving out, we see that line 8 causes line 9 to be executed a maximum of $O(p)$ times, bringing us to a total cost of $O(p^2 \cdot k^2)$ assuming line 10 is executed in constant time. Then, line 6 contributes another $O(p \cdot k^2)$, making the entire block of lines $5-11$ take $O(p^2 \cdot k^2)$ time, assuming lines 5, 7, and 10 are all constant-time. Finally, the for loop at line 4 causes this inner block to execute a maximum of $end - (end - k) = k$ times, bringing us to a final running time of $O(p^2 \cdot k^3)$.

Having proven that the running time of a single update is $O(p^2 \cdot k^3)$, we now extend this bound to the full offline version of the algorithm or, alternatively, to the online version executed incrementally on a whole string w. Taking the length of the input string $|w|$ to be n, the online algorithm will need to perform n updates, each performing $O(p^2 \cdot k^3)$ operations. Thus, the overall running time of the online algorithm, when asked to segment a whole string w, will be $O(p^2 \cdot k^3 \cdot n)$. Incidentally, by Theorem 6, this is also the running time of the full offline algorithm.

We thus conclude that the overall running time to segment a string w of length n is $O(p^2 \cdot k^3 \cdot n)$, proving that the algorithm is indeed linear in the size of the input string.

8. Space Analysis

In this section we analyze the worst-case space usage of the algorithm. This analysis is actually very simple given what we've proven in the previous section. At the end, we conclude that, given a string w of length n, the space complexity of the full offline algorithm and the online algorithm is $O(p \cdot k^2 \cdot n)$, where p and k are as defined in section 7.

In order to provide a tighter bound on the algorithm's space complexity, we first need to make a claim about what elements of the frontier are actually populated at any given time.

Theorem 9: At the beginning of iteration i of the online algorithm, the only lists in the frontier that will contain elements will be $frontier[i-k]$ up to $frontier[i-1]$; the rest will be empty.

We will use this theorem to provide an upper bound on the space complexity. An informal proof of its correctness is provided below.

Proof of Theorem 9:

At iteration i of the online algorithm (page 11), the index end will be equal to i according to line 2, since w will contain exactly i characters (one from each iteration). Further, at iteration i, we add elements to $frontier[end]$, and only to $frontier[end]$, where $end = i$. Taking this into account, we thus provide an inductive proof that the theorem holds at all times during the algorithm's execution where the induction is performed on end (the iteration) as it starts at 1 and increments up to $|w|$. First, as our base case, consider the case when $end \leq k$, where k denotes the maximum length of a string in our language by assumption (3.1) on page 3. At this point, previous iterations will have

populated $frontier[0]$ up to $frontier[k-1]$, since elements are added only to $frontier[end]$ at each iteration. Thus the theorem will trivially hold when $end \leq k$. Now, consider the case where $end = i$, with $i > k$. By the inductive hypothesis, we will assume that the lists in $frontier[i-k]$ up to $frontier[i-1]$ are populated and that the rest of the frontier is empty. On the next iteration, lines 4 to 12 will populate $frontier[end]$ with elements. Thus the lists in $frontier[i-k]$ up to $frontier[i]$ will be populated. However, because $end - k \geq 0$ by assumption, line 14 will then proceed to clear $frontier[end-k]$, leaving only $frontier[i-k+1]$ up to $frontier[i]$ populated. Thus, at the beginning of the next iteration, iteration $i+1$, the condition will still hold. Having shown that the base and the step hold, we conclude that Theorem 9 always holds during the algorithm's execution.

We now use Theorem 9 to provide an upper bound on the space complexity of our online algorithm. What Theorem 9 basically tells us is that no more than k lists in the frontier will actually contain elements. Thus, assuming that each nonempty list in the frontier contains $p \cdot k$ lists of strings, where p and k are as defined in section 7, the frontier will never contain more than $O(p \cdot k^2)$ lists of strings. Finally, because the number of characters in each list of strings will be, at max, the length of the input string (by Theorem 5 on page 10), we conclude that the complete space complexity of the algorithm is $O(p \cdot k^2 \cdot n)$ where n denotes the length of the input string w. By Theorem 6, this is also the space complexity of the full offline algorithm.

We thus conclude that the overall space complexity when segmenting a string w of length n is $O(p \cdot k^2 \cdot n)$, proving the space usage is also linear in the size of the input.

9. Assessing Segmentation Accuracy

In this section we discuss the accuray of NoSpace (here we take NoSpace to refer to the offline algorithm on page 9) using a first-order Markov Model as our scoring function, g. We compare NoSpace, which computes the optimal segmentation under the first-order Markov Model, against a greedy "best bigram" segmenter, which we describe in detail below.

9.1. NoSpace with First-Order Markov Model

To test NoSpace, we use the Google Ngrams corpus [4] to develop a reasonable scoring function. The Google NGram corpus provides us with millions of bigrams and their associated counts, which we use to model the log probability of certain sentences occurring in natural language. That is, the Google NGram corpus gives us a set of lines as follows:

- of the 2766332416
- in the 1628795264
- to the 1139249024
- ...

We then take these counts and turn them into log probabilities. The result is as follows:

- of the $\log(P(\text{"the"}|\text{"of"}))$
- in the $\log(P(\text{"the"}|\text{"in"}))$
- to the $\log(P(\text{"the"}|\text{"to"}))$
- ...

This gives us the probability of a string w_2 appearing after a string w_1 in natural language. To extend this to the probability of a whole sentence appearing in natural language, we simply use the product rule along with a first-order Markov assumption:

$$\begin{aligned} P(\langle start\rangle, w_1, w_2, ..., w_n, \langle end\rangle) &= P(\langle end\rangle | w_1, ..., w_n) P(w_n | w_1, ..., w_{n-1}) ... P(w_1 | \langle start\rangle) P(\langle start\rangle) \\ &= P(\langle end\rangle | w_n) P(w_n | w_{n-1}) ... P(w_1 | \langle start\rangle) P(\langle start\rangle) \end{aligned} \quad (14)$$

where $\langle start \rangle$ and $\langle end \rangle$ are dummy strings used to denote the start and end of a sentence respectively. We let our scoring function g, then, be simply the log of this probability and pass this in as a parameter to the algorithm on page 9.

Further, as our perturbation model, we assume that each character can be changed to an adjacent letter on an American keyboard independently with a 5% probability. To correct for this, we define a function $perturbations(s)$ that takes in a string and returns all possible one or two-character perturbations of s. We also modify our scoring function g to account for the probability of an error, making perturbed strings less likely than unperturbed strings. This allows the algorithm to correct for misspellings in the unsegmented text.

The (admittedly unoptimized) code that performs this computation in Java is provided in the appendix.

9.2. Greedy Bigram Segmenter

As a competitor to NoSpace, we implement a second segmentation algorithm that does not necessarily end up with the optimal segmentation.

This implementation proceeds from left to right through the string and, at any given point, simply chooses the substring w_2 with the highest probability according to a first-order Markov Model, given the previous string w_1, and adds it to the output list. As an example, given "hesnothere", this algorithm could consider all substrings starting at index 0, find that "he" is the most likely string to put at the beginning of a sentence, and add it to the output list. Then, it could consider all substrings starting at index 2, decide that "snot" is the most likely string to put after "he", and add it to the output list. Finally, it could consider all substrings starting at index 6, decide that "here" is the most likely string to put after "snot", and add it to the output list. Thus, at the end the algorithm would return $\langle "he", "snot", "here" \rangle$. In order to account for noise, we have the algorithm consider perturbations of the input in addition to the input itself in the same way we did with NoSpace.

It should now be clear why this algorithm's approach is "greedy" and why the algorithm may output segmentations that are sub-optimal. Note that this is not intended to be a particularly clever algorithm, but rather, it is meant to serve as a simple baseline against which we can compare NoSpace's performance. Code for this algorithm is also provided in the appendix.

9.3. Testing Methods

In order to test the accuracy of NoSpace and the Greedy Bigram Segmenter, we performed the following steps:

- We acquired a corpus of sufficient size. In our case, we used a collection of free public domain ebooks from the Gutenberg Project [5]. A total of 40,000 sentences worth of text were used.
- We split this corpus into sentences by delimiting on periods, exclamation marks, and question marks, putting one sentence on each line. We refer to the file containing this text as the "golden file".
- We went through the golden file and, with 5% probability, perturbed each character in the file to a character adjacent to it on the keyboard. We refer to the file containing this perturbed text as the "noisy golden file".
- We removed the spaces from the golden file and the noisy golden file. We refer to the files containing the de-spaced versions of the golden file and the noisy golden file as the "de-spaced golden file" and the "de-spaced noisy golden file".

- We ran NoSpace and the Greedy Bigram Segmenter on the de-spaced golden file and the de-spaced noisy golden file.
- As we segmented each line in the de-spaced files, we kept track of statistics such as how many times unnecessary spaces were added, how many times too few spaces were added, and how many times each algorithm printed an incorrect character.

And that's all there is to it. After running both algorithms on the de-spaced files and gathering statistics, we were able to get a good idea of not only NoSpace's accuracy, but also some intuition on how much ambiguity there is in the segmentation problem for English text. A summary of some of our statistics is provided in the table below: We now explain

Table 1: Noiseless Input Results

	Extra Spaces	Missing Spaces	Incorrect Characters	Total Characters	Total Spaces	Percentage Spaces Correct
NoSpace	11618	3244	0	3239103	571704	97.4%
Greedy	96253	77844	0	3239103	571704	69.55%

Table 2: Noisy Input Results (5% Noise)

	Extra Spaces	Missing Spaces	Incorrect Characters	Total Characters	Total Spaces	Percentage Spaces Correct	Percentage Correct Overall
NoSpace	20020	8076	31270	3239103	571704	95.09%	98.17%
Greedy	263145	148301	134419	3239103	571704	28.03%	83.14%

these tables in a little more detail. We measured each of these statistics by going through each sentence with two pointers. One into the golden file and one into the segmented file. Whenever there was a space in the segmented file but not in the golden file, we incremented "Extra Spaces" above. Whenever there was a space in the golden file but not in the segmented file, we incremented "Missing Spaces" above. Finally, whenever we encountered a character in the golden file that was different from a character in the segmented file and at the same position (but was not a space), we incremented "Incorrect Characters". Thus in the noiseless case we never had any incorrect characters for either algorithm. However, in the noisy case note that some errors weren't fixed by either algorithm. "Total Characters" is simply the total number of characters in the golden file and "Total Spaces" is the total number of spaces in the golden file. To compute "Percentage Spaces Correct" we simply compute $1 - (\text{"}ExtraSpaces\text{"} + \text{"}MissingSpaces\text{"})/\text{"}TotalSpaces\text{"}$ and to compute "Percentage Overall Correct", we compute $1 - (\text{"}ExtraSpaces\text{"} + \text{"}MissingSpaces\text{"} + \text{"}IncorrectCharacters\text{"})/TotalCharacters$.

9.4. Analysis of Results

Of course, there are many interesting things to note about the results but one particularly interesting point is that NoSpace tends to include too many spaces rather than too few. That is, it tends to "overspace" rather than "underspace". This is somewhat counterintuitive, since when using a first-order Markov Model, one would expect longer sentences to be somewhat discounted. This could be perhaps be an indication that our text corpus is "different" than the corpus on which the Google NGrams were extracted.

In terms of NoSpace's accuracy, it appears as though NoSpace does reasonably well

when it comes to segmenting noiseless text, suggesting there is not much ambiguity in English when it comes to spacing. That is, odd cases such as having to choose between phrases such as "aloud" and "a loud" are not too difficult when given sufficient context. In fact, close analysis of the mistakes made by NoSpace seem to suggest that most of the issues came from proper nouns in novels and plays that the Markov Model had never seen before. As an example, simply consider the following inputs from "Les Miserables":

- "hewasfirstseenatjauziersthenattuiles" $\rightarrow$ "he was first seen at ja uzi ers then at tuiles"
- "hewasmakinghiscircuittochastelar" $\rightarrow$ "he was making his circuit to chastel ar"
- "interpocula" $\rightarrow$ "interpo cula"
- "ithinkitwasescoublon" $\rightarrow$ "i think it was esco ubl on"
- "defeletezsignedhimselfa" $\rightarrow$ "def elete z signed himself a"

Unusual proper nouns, ubiquitous due to the fact that our test corpus consisted of text from old public domain ebooks, appeared to be the main cause of NoSpace's errors in the "Extra Spaces" column, suggesting again that the corpus on which we tested NoSpace was somewhat different than the one from which we extracted our NGrams. We note that this is a much more favorable cause of errors than ambiguity, and that ambiguity appeared to account for relatively few errors upon close examination of the results.

We thus conclude that NoSpace's performance is pretty good as it is, but that it appears as though it can be improved further by using a scoring function that is more tailored to the input that it will receive. For example, if the NGrams used were extracted from a user's emails and used as a scoring function to segment text messages typed by the same user, then we predict that NoSpace could do considerably better than these results suggest.

10. Implementation on Nexus 4

Theoretical results about the algorithm's efficiency are great, but at the end of the day the algorithm needs to be tested on the platform on which it is most useful. For this reason, we implemented NoSpace using a first-order Markov Model as described, along with several perturbation models, on a Nexus 4 phone. Even with almost no knowledge of mobile programming we were able to code up an implementation that works very well, confirm that NoSpace does indeed do a very good job of segmenting text, even in the face of fairly serious input noise, and confirm that it does indeed run fast enough to make its way into commercial autocorrection engines. Segmentations were generally instantaneous, even when running the offline algorithm on the whole input string every time. Further, although we have not yet implemented the online version of NoSpace, doing so would allow one to take advantage of extra time that the user spends thinking and typing, and can even allow NoSpace to provide real-time suggestions as one is entering text. We thus conclude that our first impressions of NoSpace as a mobile app are promising and suggest that it has the potential to be useful on today's handheld devices.

11. Concluding Remarks

We have presented an online, linear-time algorithm that segments text optimally while removing input noise and, what's more, we have provided convincing evidence to suggest that omitting spaces from input results in a minimal loss of information. Indeed, from our analysis in section 9.4, not only does it appear as though NoSpace makes very few errors overall, but also the errors it does make are likely due to a weak scoring function (something that can be fixed) rather than ambiguity when it comes to which segmentation is optimal. Further, the fact that NoSpace is an online algorithm means that it can process single characters at a time *as a user is typing*, and take advantage of

time that would otherwise be wasted in order to provide more sophisticated perturbation and error-correction models, among other things. When one considers the algorithm's efficiency in addition to its apparent accuracy, the results suggest that NoSpace has the potential to become a powerful tool that may one day make its way onto mobile devices and into the hands of the masses.

12. Prior Work

The Viterbi algorithm is an algorithm for finding the maximum likelihood sequence, given noisy observations, under a Hidden Markov Model. Forney [2] gives a good description, both of how it works and of its overall importance. While NoSpace doesn't explicitly require one to define a Hidden Markov Model, the computation of the maximum likelihood sequence in both NoSpace and Viterbi utilizes dynamic programming heavily and the two algorithms are intimately related. Gambell and Yang [1] discuss techniques for segmenting speech using statistical learning, although this appears to be overkill when the input is text. Finally, not all languages have natural delimiters like the space in English. Such languages include Japanese and Chinese. In these languages, sentences are delimited but words are not, and so a lot of research effort has been spent on trying to extract meaning from text written in such languages, such as work done by Sproat and Emerson [3]. These algorithms are also overkill for languages that have natural delimiters, since in delimited languages we have access to copious amounts of training data, whereas for non-delimited languages we do not. As an example, we can generate a fairly good n^{th}-order Markov Models for English text simply by crawling through a large corpus and keeping track of counts. For a language such as Chinese, however, it is not possible to generate such a model, since there are few corpuses with actual word-level delimiters on which to train.

References

[1] T. Gambell, C. Yang. Word Segmentation: Quick but not Dirty. 2005.

[2] D. Forney. The Viterbi Algorithm: A personal History. 2005.

[3] R. Sproat, T. Emerson. The First International Chinese Word Segmentation Bageoff. 2003

[4] Google Ngram Corpus: http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html

[5] The Gutenberg Project: http://www.gutenberg.org/

13. Appendix

Proof of Theorem 1:

The proof proceeds by strong induction on the value of the index variable *end* using the pseudocode on page 7.

Base: When $end = 0$, before the for loop on line 2, the theorem holds.

To see this, simply consider that after line 1, the only element contained in the frontier is $\langle ""\rangle$ in $frontier[0]$, which concatenates to "", which is equal to $w^{0:0}$. Thus, the theorem holds for the base case.

Step: Assuming the theorem holds when $end \leq i$, show it holds at the beginning of step $end = i + 1$.

We show that the work done at step i causes the theorem to hold at the beginning of step $i + 1$. When $end = i$, we only add to the frontier at line 7. In particular, we only add to $frontier[end] = frontier[i]$ so all we need to do is make sure the elements of the

list that we add to the frontier at this line concatenate to $w^{0:end} = w^{0:i}$. Well, the list that we add consists of a list from $frontier[start]$ (line 4) with $s = w^{start:end}$ (line 5) at the end. Because the theorem holds for $frontier[i]$ where $i \leq end$ (assumed), and because $start < end$ (line 4), the elements of any list in $frontier[start]$ concatenate to $w^{0:start}$. But $s = w^{start:end} = w^{start:i}$ by line 5. So taking an element from $frontier[start]$ and adding s to the end yields a list whose elements concatenate to $w^{0:end} = w^{0:i}$, which we add to $frontier[end] = frontier[i]$. Thus, the theorem holds when we reach step $end = i + 1$ assuming it holds before the step where $end = i$.

Having shown both the base and the step, we now conclude that the theorem holds at all points during the algorithm's execution.

Proof of Theorem 2:

To prove Theorem 2, we actually prove a stronger result first.

Lemma 1: $frontier[i]$ **contains lists of strings** $\langle s_1, s_2, ..., w^{j:i} \rangle$ **for all** $j \in [max(0, end - k), end[$.

This is just saying that $frontier[i]$ always contains lists of strings such that the last string in each list is a substring of $w^{0:i}$ ending at index i. More than that, it is saying that *all* substrings of length k or less appear as the last string in each list in $frontier[i]$. The proof is clear simply from observing that the for-loop on line 4 adds all such lists to the frontier.

Lemma 2: During the algorithm's execution, $frontier[i]$ **contains lists of strings** $l = \langle s_1, s_2, ..., s_j \rangle$ **such that** $s_1||s_2||...||s_j = w^{0:i}$ **and** l **is the *best* segmentation of** $w^{0:i}$ **ending in** s_j**. That is,** $g(\langle s_1, s_2, ..., s_j \rangle) \geq g(\langle s'_1, s'_2, ..., s_j \rangle)$ **where for all** s'_1**,** s'_2**,... such that** $s'_1||s'_2||...||s_j = w^{0:j}$**.**

The proof proceeds by strong induction on the value of the index variable *end* using the pseudocode on page 7. We also make use of restriction (3.2) on our scoring function g, defined on page 3.

Base: When $end = 0$, before the loop on line 2, the theorem holds.

There is only one segmentation of $w^{0:0} = ""$ and that is simply $\langle "" \rangle$, which we add to the frontier at line 1. This segmentation is the only one so it must be optimal and thus the lemma holds when $end = 0$.

Step: Assuming the lemma holds for any step where $end \leq i$, show it holds at the beginning of step $end = i + 1$.

By our assumption, at the beginning of step i of the algorithm, $frontier[j]$ contains optimal lists of strings ending in $w^{m:j}$ for all $m \in [max(0, j - k), j[$ and all $j \in [max(0, (i - 1) - k), (i - 1)[$. We want to show that after step i, $frontier[j]$ contains optimal lists of strings ending in $w^{m:j}$ for all $m \in [max(0, j - k), j[$ and all $j \in [max(0, i - k), i[$. Thus, what we need to show is that all of the strings added to $frontier[i]$ at step i of the algorithm ending in $w^{m:i}$ are optimal for $m \in [max(0, i - k), i[$.

At step i, we only add to the frontier at line 7. In particular, we only add to $frontier[i]$. Further, note that the for loop starting at line 4 considers adding all lists ending in $s_{start} = w^{start:i}$ for $start \in [max(0, i - k), i[$. Thus, without loss of generality we prove that the lemma holds for one particular s_{start} being added. At line 6, we consider

adding lists of string from $frontier[start]$ with s_{start} at the end to $frontier[i]$. The way we choose which list from $frontier[start]$ is using an argmax over all the elements. Without loss of generality, $best\_prev\_list_{best} = \langle s_1, s_2, ..., s_j \rangle$ be the list that we choose to append s_{start} to. By assumption, we know that $g(best\_prev\_list_{best}) = g(\langle s_1, s_2, ..., s_j \rangle) \geq g(\langle s'_1, s'_2, ..., s_j \rangle)$ for any $s'_1, s'_2, ...$ But, by restriction (3.2) on g on page 3, this implies that $g(\langle s_1, s_2, ..., s_j, s_{start} \rangle) \geq g(\langle s'_1, s'_2, ..., s_j, s_{start} \rangle)$. Further, since we took an argmax over all the s_j, we have: $g(\langle s_1, s_2, ..., s_j, s_{start} \rangle) \geq g(\langle s'_1, s'_2, ..., s'_j, s_{start} \rangle)$. Thus we have that $best\_prev\_list_{best} || s_{start}$ is the *optimal* segmentation of $w^{0:i}$ ending with s_{start}. Because the proof is identical for all of the s_{start}, we thus conclude that the lemma still holds at the beginning of step $end = i + 1$.

Having shown that both the base and the step hold, we thus conclude that Lemma 2 is true for at all times during the algorithm's execution.

Since Lemma 2 is clearly a *stronger* condition than Theorem 2, we thus conclude that Theorem 2 holds.

Proof of Theorem 3:

The proof of this theorem is almost identical to that of Theorem 9, so we direct the reader to the proof of Theorem 9 on page 13.

Proof of Theorem 4:

The proof of this theorem is almost identical to that of Theorem 1, so we direct the reader to the proof above.

Proof of Theorem 5:

The proof of this theorem is almost identical to that of Theorem 2, so we direct the reader to the proof above.

Implementation of the Full Segmentation Algorithm (page 9) in Java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class OptimalSegmenter {
	private static final double minProb = -1.0e100;
	private static final double errorProb = Math.log(.05);

	private static class Node
	{
		String str;
		public double updatedGVal;
		public Node previous;

		public Node(String s, double g, Node prev)
		{
			str = s;
			updatedGVal = g;
			previous = prev;
		}
	}

	// Read lines from a file containing unigrams. The lines should be
	   formatted
```

```
// exactly as follows: "w1\tcount\n" where w1 is a string and count is a
    double.
// filename is the location of the file and numUnigramsToRead is the
    number of
// lines to read from this file.
public static HashMap<String, Double> readUnigrams(
            String filename,
            int numUnigramsToRead) throws IOException
            {
      BufferedReader unigramReader = new BufferedReader(new
          FileReader(filename));
      HashMap<String, Double> unigrams = new HashMap<String, Double>();

      int numLinesReadIn = 0;
      String line;
      double totalCount = 0; // Used to convert frequency counts to
          probabilities at the end.

      // Read lines from the file into our map data structure.
      while ((line = unigramReader.readLine()) != null
                  && numLinesReadIn < numUnigramsToRead) {
            String[] stringValues = line.split("\t");

            if (stringValues.length != 2) {
                  unigramReader.close();
                  throw new IOException("Unigram input file should contain"
                              + "lines of the form: '%s\t%f\n'");
            }

            String w1 = new String(stringValues[0]);
            double count = Double.parseDouble(stringValues[1]);
            totalCount += count;

            if (unigrams.containsKey(w1)) {
                  unigrams.put(w1, unigrams.get(w1) + count);
            } else {
                  unigrams.put(w1, count);
            }

            numLinesReadIn++;
      }
      unigramReader.close();

      // Convert the frequency counts into normalized log probabilities.
      for (Map.Entry<String, Double> entry : unigrams.entrySet()) {
            unigrams.put(entry.getKey(), Math.log(entry.getValue()) -
                Math.log(totalCount));
      }

      return unigrams;
            }

// Read lines from a file containing bigrams. The lines should be formatted
// exactly as follows: "w1\tw2\tcount\n" where w1 is a string, w2 is a
    string and
// count is a double. filename is the location of the file and
    numBigramsToRead is
// the number of lines to read from this file.
public static HashMap<String, HashMap<String, Double>> readBigrams(
            String filename,
            int numBigramsToRead) throws IOException
            {
      BufferedReader bigramReader = new BufferedReader(new
          FileReader(filename));
      HashMap<String, HashMap<String, Double>> bigrams =
                  new HashMap<String, HashMap<String, Double>>();

      int numLinesReadIn = 0;
      String line;

      // Read lines from the file into our map data structure.
      while ((line = bigramReader.readLine()) != null
                  && numLinesReadIn < numBigramsToRead) {
            String[] stringValues = line.split("\t");

            if (stringValues.length != 3) {
                  bigramReader.close();
                  throw new IOException("Bigram input file should contain"
                              + "lines of the form: '%s\t%s\t%f\n'");
            }
```

```
            String w1 = new String(stringValues[0]);
            String w2 = new String(stringValues[1]);
            double count = Double.parseDouble(stringValues[2]);

            HashMap<String, Double> distribution = bigrams.get(w1);
            if (distribution != null) {
                  if (distribution.containsKey(w2)) {
                        distribution.put(w2, distribution.get(w2) + count);
                  } else {
                        distribution.put(w2, count);
                  }
            } else {
                  distribution = new HashMap<String, Double>();
                  distribution.put(w2, count);
                  bigrams.put(w1, distribution);
            }

            numLinesReadIn++;
      }
      bigramReader.close();

      // Convert the frequency counts into normalized log probabilities.
      for (Map.Entry<String, HashMap<String, Double>> entry :
          bigrams.entrySet()) {
            double totalCount = 0;
            HashMap<String, Double> value = entry.getValue();
            for (Map.Entry<String, Double> subEntry : value.entrySet()) {
                  totalCount += subEntry.getValue();
            }

            for (Map.Entry<String, Double> subEntry : value.entrySet()) {
                  value.put(subEntry.getKey(),
                      Math.log(subEntry.getValue())
                                  - Math.log(totalCount));
            }
      }

      return bigrams;
            }

// This is a scoring function based on a first-order Makov model.
// If it can't find a bigram probability, it returns the unigram
   probability
// instead. If it can't find either a bigram probability or a unigram
// probability, it returns a smoothed value minProb.
public static double g(
            String w1,
            String w2,
            double oldG,
            HashMap<String, Double> unigrams,
            HashMap<String, HashMap<String, Double>> bigrams)
{
      HashMap<String, Double> temp = bigrams.get(w1);
      if (w1.length() != 0 && temp != null) {
            Double val = temp.get(w2);
            if (val != null) {
                  return oldG + val.doubleValue();
            } else {
                  val = unigrams.get(w2);
                  if (val != null) {
                        return oldG + val.doubleValue();
                  } else {
                        return oldG + minProb;
                  }
            }
      } else {
            Double val = unigrams.get(w2);
            if (val != null) {
                  return oldG + val.doubleValue();
            } else {
                  return oldG + minProb;
            }
      }
}

// Returns best_prev_list || s as described in the paper.
public static Node argmaxNode(
            ArrayList<Node> previousNodes,
            String s,
            HashMap<String, Double> unigrams,
```

```
            HashMap<String, HashMap<String, Double>> bigrams)
{
      double maxG = Double.NEGATIVE_INFINITY;
      int bestXIndex = -1;
      for (int i = 0; i < previousNodes.size(); i++) {
            Node x = previousNodes.get(i);
            double gXS = g(x.str, s, x.updatedGVal, unigrams, bigrams);
            if (gXS > maxG) {
                  maxG = gXS;
                  bestXIndex = i;
            }
      }
      Node bestPrevNode = previousNodes.get(bestXIndex);

      return new Node(s, maxG, bestPrevNode);
}

// Returns argmax(frontier[|w|]) as described in the paper.
public static Node argmaxEnd(
            ArrayList<Node> nodes,
            HashMap<String, Double> unigrams,
            HashMap<String, HashMap<String, Double>> bigrams) {
      double maxG = Double.NEGATIVE_INFINITY;
      int bestNodeIndex = -1;
      for (int i = 0; i < nodes.size(); i++) {
            double currentG = g(nodes.get(i).str, "</s>",
                                    nodes.get(i).updatedGVal,
                                       unigrams, bigrams);
            if (currentG > maxG) {
                  maxG = currentG;
                  bestNodeIndex = i;
            }
      }

      return nodes.get(bestNodeIndex);
}

public static ArrayList<String> extractStringArray(Node n)
{
      ArrayList<String> ret = new ArrayList<String>();

      while(n != null) {
            ret.add(n.str.toString());
            n = n.previous;
      }

      Collections.reverse(ret);
      return ret;
}

// A pair class. We use this so that perturbations can have different
// effects on the scoring function.
private static class Pair<X, Y>
{
      public X first;
      public Y second;

      public Pair(X x, Y y)
      {
            first = x;
            second = y;
      }
}

// This function returns all perturbations of the input string. Right now
// we consider single and two-character perturbations.
public static String[] adjacencies = {"qwsxz", "vfghn", "xsdfv",
    "swerfvcx",
                                                      "wsdfr",
                                                          "cdertgbv",
                                                          "vfrtyhnb",
                                                          "bgtyujmn",
                                                      "ujklo", "yhnmkiu",
                                                          "jmloiu",
                                                          "kiop", "nhjk",
                                                      "bghjm", "iklp",
                                                          "ol", "asw",
                                                          "edfgt",
                                                      "zaqwedcx",
                                                          "rfghy",
```

```
                                                        "yhjki",
                                                        "cdfgb",
                                                  "qasde", "zasdc",
                                                        "tghju", "xsa"};
static char[] tempString = new char[100];
public static ArrayList<Pair<String, Double>> perturbations(String s)
{
      ArrayList<Pair<String, Double>> ret = new ArrayList<Pair<String,
          Double>>();
      s.getChars(0, s.length(), tempString, 0);

      // Single character perturbations.
      for (int i = 0; i < s.length(); i++) {
            char currentChar = s.charAt(i);
            if (!Character.isLetter(currentChar)) {
                  continue;
            }

            int charIndex = Character.toLowerCase(currentChar) - 'a';
            for (int j = 0; j < adjacencies[charIndex].length(); j++) {
                  if (Character.isLowerCase(currentChar)) {
                        //s.getChars(0, s.length(), tempString, 0);
                        tempString[i] = adjacencies[charIndex].charAt(j);
                  } else {
                        //s.getChars(0, s.length(), tempString, 0);
                        tempString[i] =
                              Character.toUpperCase(adjacencies[charIndex].charAt(j));
                  }

                  Pair<String, Double> perturbation =
                              new Pair<String, Double>(new
                                  String(tempString, 0, s.length()),
                                  errorProb);
                  ret.add(perturbation);

                  tempString[i] = s.charAt(i);
            }
      }

      // Two-character perturbations.
      for (int i = 0; i < s.length(); i++) {
            for (int j = i + 1; j < s.length(); j++) {
                  char c1 = s.charAt(i);
                  char c2 = s.charAt(j);

                  if (!Character.isLetter(c1) || !Character.isLetter(c2)) {
                        continue;
                  }

                  int index1 = Character.toLowerCase(c1) - 'a';
                  int index2 = Character.toLowerCase(c2) - 'a';

                  for (int k1 = 0; k1 < adjacencies[index1].length();
                      k1++) {
                        for (int k2 = 0; k2 <
                            adjacencies[index2].length(); k2++) {
                              //s.getChars(0, s.length(), tempString, 0);

                              if (Character.isLowerCase(c1)) {
                                    tempString[i] =
                                        adjacencies[index1].charAt(k1);
                              } else {
                                    tempString[i] =
                                          Character.toUpperCase(adjacencies[index1].charA
                              }

                              if (Character.isLetter(c2)) {
                                    tempString[j] =
                                        adjacencies[index2].charAt(k2);
                              } else {
                                    tempString[j] =
                                          Character.toUpperCase(adjacencies[index2].charA
                              }

                              Pair<String, Double> perturbation =
                                          new Pair<String, Double>(new
                                              String(tempString, 0,


                              ret.add(perturbation);
```

```
                                    tempString[i] = s.charAt(i);
                                    tempString[j] = s.charAt(j);
                              }
                        }
                  }
            }

            return ret;
      }

      // This is the heart of our class. This algorithm takes an input string
          and segments it
      // optimally. Everything in this function has been organized to resemble
          the algorithm
      // presented in the paper as closely as possible.
      public static ArrayList<String> segmentOptimal(
                  String input,
                  int maxLength,
                  HashMap<String, Double> unigrams,
                  HashMap<String, HashMap<String, Double>> bigrams)
                  {
            ArrayList<ArrayList<Node>> frontier = new
                ArrayList<ArrayList<Node>>();
            frontier.add(new ArrayList<Node>());
            frontier.get(0).add(new Node(new String("<s>"), 0, null));

            for (int end = 1; end <= input.length(); end++) {
                  frontier.add(new ArrayList<Node>());

                  for (int start = Math.max(0, end - maxLength); start < end;
                      start++) {
                        String s = input.substring(start, end);
                        Node bestPrevList = argmaxNode(frontier.get(start), s,
                            unigrams, bigrams);
                        frontier.get(end).add(bestPrevList);

                        for (Pair<String, Double> sPrime : perturbations(s)) {
                              if (!unigrams.containsKey(sPrime.first)) {
                                    continue;
                              }

                              bestPrevList = argmaxNode(frontier.get(start),
                                  sPrime.first, unigrams, bigrams);
                              bestPrevList.updatedGVal += sPrime.second;

                              frontier.get(end).add(bestPrevList);
                        }
                  }

                  if (end - maxLength >= 0) {
                        frontier.get(end - maxLength).clear();
                  }
            }

            Node optimalNode = argmaxEnd(frontier.get(input.length()), unigrams,
                bigrams);

            return extractStringArray(optimalNode);
                  }

      // An implementation of the "Greedy Bigram" segmentation algorithm.
      public static ArrayList<String> segmentGreedyBigram(
                  String input,
                  int maxLength,
                  HashMap<String, Double> unigrams,
                  HashMap<String, HashMap<String, Double>> bigrams)
                  {
            ArrayList<String> ret = new ArrayList<String>();
            ret.add("<s>");

            int pos = 0;
            while (pos < input.length()) {
                  double maxG = Double.NEGATIVE_INFINITY;
                  int bestPos = -1;
                  for (int i = pos + 1; i <= input.length(); i++) {
                        double currentG = g(ret.get(ret.size() - 1),
                                                input.substring(pos, i), 0,
                                                unigrams, bigrams)/(i -
                                                    pos);
                        if (currentG > maxG) {
```

```
                        maxG = currentG;
                        bestPos = i;
                    }
                }
                ret.add(input.substring(pos, bestPos));
                pos = bestPos;
            }

            return ret;
                }

    // Perturbs each character in a string to an adjacent character on an
       American keyboard
    // independently with probability .05.
    public static String addNoise(String line)
    {
            char[] temp = line.toCharArray();
            for (int i = 0; i < temp.length; i++) {
                if (Math.random() < .05 && Character.isLetter(temp[i])) {
                    temp[i] = adjacencies[Character.toLowerCase(temp[i]) -
                        'a'].charAt((int)(Math.random()
                                * adjacencies[Character.toLowerCase(temp[i])
                                    - 'a'].length()));
                }
            }

            return new String(temp);
    }

    public static void main(String[] args) throws IOException
    {
            // The files from which to get the unigram and bigram language model
               are specified
            // as arguments.
            String unigramFile = args[0];
            String bigramFile = args[1];

            // This corresponds to the length restriction (3.1) mentioned in the
               paper.
            int maxLength = Integer.parseInt(args[2]);

            // Reads in the language model from a text file.
            HashMap<String, Double> unigrams = readUnigrams(unigramFile, 500000);
            HashMap<String, HashMap<String, Double>> bigrams =
                readBigrams(bigramFile, 5000000);

            BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in));

            System.out.println("Enter an unsegmented string.");
            int numLinesRead = 0;

            String line;
            while ((line = br.readLine()) != null) {
                numLinesRead++;

                ArrayList<String> optSeg = segmentOptimal(line.toLowerCase(),
                    maxLength, unigrams, bigrams);
                System.out.print(optSeg.get(1));
                for (int i = 2; i < optSeg.size(); i++) {
                    if (optSeg.get(i).contains("'")) {
                        System.out.print(optSeg.get(i));
                    } else {
                        System.out.print(" " + optSeg.get(i));
                    }
                }
                System.out.println();
            }
    }
}
```