SnipIt - A real time Dynamic Programing approach to Snippet Generation for HTML Search Engines

Ian Carr[*]
Pomona College Department of Computer Science
185 E. Sixth St.
Claremont, CA 91711
itc02006@mymail.pomona.edu

Lucy Vasserman[†]
Pomona College Department of Computer Science
185 E. Sixth St.
Claremont, CA 91711
lhv02006@mymail.pomona.edu

ABSTRACT

Returning a variety of search results is a key focus in the field of Information Retrieval. However, due to lexical ambiguity, results for a single search term spanning many categories can be confusing for a user. A simple but powerful way to help a user find the best documents for their information needs is to return, along with an ordered list of documents, short excerpts that help to differentiate and refine the documents' meanings. We prepose a simple and fast algorithm to create these so-called "snippets" that show query terms in the context of the document as a means to further enhance the user experience of an HTML based Information Retrieval system.

1. INTRODUCTION

It has been found that returning short summaries or snippets with search terms can greatly improve the user experience of an IR system. Summaries give users an idea of what content a document contains, allowing them to pick and choose documents that better fit their information needs. However, snippet systems differ from other IR tasks in their demands on speed and execution time.

The majority of IR systems do a great amount of work to insure that query results can be returned quickly. Techniques such as postings lists, word counts, and other query-independent[1] techniques can all be done at index time. This allows for the majority of web search engines to return results on the order of hundreds of milliseconds.

However, snippets are, by definition, very query-specific. Because of this, great pains are required to insure that a snippet is not only useful to a user, but can be generated for multiple documents at search-time. This limits the ability for snippet generators to be able to take other documents or even the whole of a single document into account when generating a snippet. This paper will not discuss the difficulties of storing full-text documents in memory (another very worthy area of research), but rather a simple heuristic based snippet generator that works to generate query-in-context snippets in tens of milliseconds.

[*]The Handsome One
[†]The One who can Dance
[1]common to many query terms or words

2. THE ALGORITHM

In approaching the problem of creating a relevant and useful snippet per document at search-time, we chose to break the problem down. Our algorithm first assigns each individual word a score, as determined by a linear combination of several feature functions. Then, we use a dynamic programing algorithm to return the best scoring set of words that fit within a definable window size. In the following sections we will discuss these features and the methodologies behind them. Here, we discuss the base Dynamic Programing algorithm, and then various efficiency improvements made to decrease run time by a factor of 10. Finally we will analyze our system against the snippets generated by the web search engine Yahoo.com, as a flawed but still useful metric of our system's performance.

2.1 Word Features

One of the main uses for snippets is to allow a user to select those documents from a ranked list that best suit their information need. Snippets compensate for the inherent problems with IR systems, as if a system were perfect a user would have no need for snippets - the first document would always perfectly satisfy their need. Therefore a good snippet should help users differentiate the information content with respect to the query terms of one document from that of other returned documents.

However, due to the time and space requirements of an IR system, it is often not feasible to have documents aware of one another in the snippet generating process. Another reason to not rely on access to other documents during snippet generation is the need for many larger IR systems to return query results from a distributed system of computers, making cross document snippet generation impossible or very difficult. An ideal system returns snippets that help to differentiate a document from other documents returned, without not requiring access or knowledge of those other documents.

The following discusses the feature functions we used to give each word a score that aids our system in finding a maximal window of words from which to form our snippet.

2.1.1 Query Term

One of the most basic ways to find a good snippet is to locate a set of adjacent words that contain many query terms. By returning snippets that contain query terms we are able to show them in context. While there may be a better set of words to help differentiate two documents from one another, we make the assumption that a user will be more concerned with the differences between two documents in terms of query term context than other contexts. While other metrics, such as salient terms or keywords, may do a better job at telling two documents apart in general, we are trying to tell them apart in the particular area of their use of query terms. Therefore, we give a high score to words that match the query terms.

2.1.2 Proximity to Query Term

One of the important parts of giving a query term in context is that context itself. By positively weighting words that appear immediately adjacent to a query term we are able to encourage snippets that contain a query term in the middle of a sentence. It is important that we try to return snippets with query terms roughly in their center, as the context of a word is important. The main justification for this is to prevent snippets from starting with or ending with a query term if they are in the middle of a sentence. This slight nudge was all our particular algorithm needed to keep query terms roughly centered in returned snippets.

2.1.3 Punctuation

In order to provide snippets that contained more context than just a mass of query terms, we chose to negatively weight certain punctuation to avoid straddling sentence breaks unless well justified. This means that the size of a returned snippet is dynamic, and while it can be as long as a provided window size, we may return a shorter snippet to prevent unnecessarily spilling into other sentences. The only times we do allow more than one sentence in a snippet is when both sentences contain positively weighted information that offsets the negative cost of containing the punctuation. We did not count semantically important punctuation such as quotes, question marks, and exclamation marks in this feature.

2.2 Dynamic Programing Algorithm

For this research we chose to consider the best snippet for a given document as the best words within a span between index i and index j in the list of words in the document, where $j - i \leq k$ for some window size k, such that:

$$\sum_{x=i}^{j} f_1(x) + f_2(x) + \cdots + f_n(x)$$

for some feature functions f_1 to f_n that take in a words index and return a score. This conception of the problem has the advantage of allowing us to use a dynamic programing approach, but does have the downsides that our algorithm is only as good as the underlying feature functions. For a discussion of the feature functions used in this research, please see 2.1.

Our algorithm, then, is tasked with returning indices i and j representing a substring with maximal total score.

2.2.1 Correctness

We submit that our algorithm is correct in finding indices i and j representing a substring with maximal total score. We provide the following proof:

Optimal Substructure. Consider some maximal substring i to j with words $w_i, w_{i+1}, \ldots, w_j$. Now consider the subset of words of length one shorter than the previous, either $w_i, w_{i+1}, \ldots, w_{j-1}$ or $w_{i+1}, w_{i+2}, \ldots, w_j$. If either has a larger total score than our maximal solution, we would use that one instead and have a better maximal solution. Therefore this problem exhibits optimal substructure.

Recursive Definition. $DP(i, j)$ = The optimal substring within the window i to j. It does not necessarily contain all words between i and j, it could be some smaller substring within the window. There exists some function $S(i)$ which returns the score for the word at index i and $SUM(i, j)$ which returns the total score of the words at indexes i through j.

$$DP(i,j) = \begin{cases} S(i) & if\ i = j \\ max(DP(i+1,j), DP(i,j-1)) & if\ (j-i) > k \\ max(DP(i+1,j), & \\ DP(i,j-1),\ SUM(i,j)) & otherwise \end{cases}$$

Table. Our Dynamic Programing Table is a $w \times w$ table, where w = the number of words in the document. It stores in cell (i, j) what is needed to retrieve the maximal sequence of adjacent words less than k in length. If we are calling our function over a whole document, we would call $DP(0, w-1)$. After completing the call our snippet will be located in in cell $(0, w-1)$ of our table.

2.3 Architectural Speed and Quality Enhancements

The above algorithm runs in $O(w^2)$ where w is the word count of the document. This means our algorithm runs in time exponential to the length of the document it is generating a snippet for. Because of this, and the already discussed need to keep the time costs of our code light, we made several changes to speed up the operation of our base code.

2.3.1 Word Pre-Scoring

To avoid numerous calls for the score of a specific word, we first calculate the scores of every word and then use this array of scores in our algorithm. This is a slight speed improvement, but generally was just a cleaner implementation choice.

2.3.2 Minimum Window Size

Our algorithm allows for snippets to be any size equal to or less than the max size – a feature that allows our snippets to avoid unsavory aspects such as meaningless beginnings or ends to sentences. However, this also means that in some rare cases our snippets can be very short – possibly one word long in some instances. Because of this we added a minimum window size where instead of reaching a base case of one word, we return the sum of some minimum set of words. This is both an improvement in quality (we avoid snippets too short to be helpful) and a very slight speed increase (as we reduce a small number of recursive calls).

2.3.3 Window Decrement Size

To reduce a larger number of recursive calls we added the ability to set the amount a snippet window size decreases in sub-calls to the algorithm. Instead of considering every possible window, we instead only consider windows of size $k - l * t$, where k is the largest window size, l is the amount we decrement by, and t is the recursive depth of this call. While this change does prevent every combination of k or fewer words form being considered, it does not actually affect quality in our snippets as there are very few cases when using one or two words less or more will prevent a good snippet from being chosen. This reduced the runtime by a factor of a little less than 2 for small decrement sizes.

2.3.4 Paragraphs as Documents

The largest speed gain we made was by calling our algorithm on each paragraph in a document separately, as our corpus contains indices for paragraph breaks. The downside to the approach was that it became impossible for a snippet to straddle a paragraph break. However, as it is questionable if there are ever desirable snippets that straddle paragraph breaks, and as this gave us a speed increase of almost a factor of 10, it was deemed a good trade-off.

2.3.5 Snippet Stitching

To hopefully provide a better overall picture of a document, it was decided that instead of returning snippets of a longer length, we would return the best and second best snippets from a document with the restriction that those snippets must come from different paragraphs. While this addition is difficult to test (see 3), it appears to improve the usefulness of the snippets. By providing two different examples of query terms in context, the user is able to get a better grasp of the similarities and differences between returned documents, and thus is served better by having two snippets stitched together. It should be noted that this is only a viable option if the combined window size is large enough for two small snippets to be useful.

3. EVALUATION

Snippet generation is inherently difficult to test. A good snippet is one that helps the user understand the document. We chose to compare our system to a set of documents and snippets determined to be “good”.

3.1 Test Corpus

In collaboration with another research team, we created a corpus to test the quality of our snippet generation. We generated a set of queries, both single and multiple words, and searched yahoo.com for those terms. We collected the text and snippets of the returned documents to serve as our test corpus. While this does force the assumption that *Yahoo!* snippets are the ideal, which may not be the case, this corpus serves as a metric to compare varying implementations against each other.

We created two versions of the corpus, the first using regular web search and containing 880 documents and the second restricted to blog sites, with 1085 documents. In the first corpus, there appeared to be many documents that did not actually contain the *Yahoo!* snippet, making it impossible for us to generate the same snippet. This was a result of our use of a naive web scraper. We assumed *Yahoo!* snippets for blogs are more likely to be direct extractions from the text, and therefore created this corpus to give our system a better chance of generating matching snippets.

3.2 Methods

To evaluate our system with this corpus, we generate our own snippets from the document text. We then compare our snippet with the *Yahoo!* snippet (henceforth called the “correct” snippet) by quantifying word overlap in three different ways. Each of these use the values x: the number of terms in our snippet, y: the number of terms in the *Yahoo!* snippet, and n: the number of overlapping terms. We define the metrics as:

$$\begin{aligned} \text{Percent Correct} &= \tfrac{n}{x} \\ \text{Percent Missing} &= \tfrac{y-n}{y} \\ \text{Jaccard Coefficient} &= \tfrac{n}{x+y-n} \end{aligned}$$

We run this evaluation on all documents in the test corpus and generate an average value. We then compare variations of our algorithm to see if they improve our snippets, as compared to the correct snippets.

3.3 Results

Evaluating our system, we get mixed results. The values returned for our three metrics are relatively low, but when analyzed with respect to the test corpus, they are respectable. Results are higher for the Blog Corpus, proving our assumption that the Web Corpus documents did not typically contain the *Yahoo!* snippets. This problem may persist to a lesser extent with the Blog Corpus. Also, it is important to note that there may be multiple relevant snippets in a document, and ours may be just as useful as *Yahoo!*’s, even if they are not the same.

The following table shows some of our results:

Metric	Web Corpus	Blog Corpus
Percent Correct	34.4 %	43.9%
Percent Missing*	75.5 %	69.4 %
Jaccard	17.1 %	22.4 %

* Lower values are better for Percent Missing

Evaluation also proved our optimization techniques. We were able to generate snippets for nearly 1000 documents in about 20 seconds. With our base algorithm, single documents took seconds to complete.

4. CONCLUSIONS

While testing snippet quality is quite difficult, we believe we have been successful in our goals for this work. We consistently create coherent snippets containing query words with context around them that appear to be useful. Our algorithm is unique in that it allows for dynamically sized snippets, preventing misleading pieces of text from being returned. The algorithm is also very easy to change and improve by simply adding feature functions and varying the parameters (maximum and minimum window size, decrement size, etc). Most importantly, we return snippets very quickly, making it feasible to use this algorithm in a live search engine.

APPENDIX

Example search results, with document title and snippet generated by our system:

`query: school`

> **China's "robber king" gets life for crime school**
> ... thieves and rapists in a **school** for crime has been sentenced to life in prison ... The 68-year-old rogue charged fees to **school** dropouts to teach them his crime ...

`query: birthday`

> **Ronald Reagan Suffering From Alzherimer's**
> ... noticeable at Reagan's own **birthday** celebration when he mistakenly repeated several lines during a ... then-former British Prime Minister Margaret Thatcher said at Reagan's 82nd **birthday** celebration ...
>
> **Simpson Spends Birthday in Jail**
> ... Simpson is spending his 47th **birthday** today in jail ... No bail on his 47th **birthday**. We're told that he has received hundreds ...

`query: celebration`

> **Britain Celebrates V-E Day, Ugly Incidents in Germany**
> ... V-E Day **celebration** marking the 50th anniversary of the end of World War II in ... one long **celebration**- a **celebration** of the end of the war in Europe ...