The Generalized Document Summarizer

by

Jack I-Chieh Fu

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in
Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author .....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by .....
David R. Karger
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

The Generalized Document Summarizer

by

Jack I-Chieh Fu

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the design, implementation, and evaluation of a summarizer for plaintext, HTML, and other documents. Using algorithms and heuristics based on word frequency and other document statistics, summary sentences and keywords can be extracted from any ASCII source document. The summarizer is designed to handle a variety of documents (such as business letters, resumes, etc.), and the user is given extensive control over the relative weights of the features and other parameters. A series of tests and evaluations were done to determine the usefulness of the generated summaries in such areas as relevance evaluation of documents and information searches on the web.

Thesis Supervisor: David R. Karger
Title: Assistant Professor

Acknowledgments

```
start_up = proc()
   % req: thesis gets done.
   % mod: none.
   % eff: thank the appropriate people and give credit where credit is due.
   %      Must be done before May 23, 1997.

   t: thanks := thanks$create("It's done! Jack's M.Eng Thesis!\n")

   % call the thanking procedures in order
   thanks$thank_god(t)
   thanks$thank_profs(t)
   thanks$thank_xsoft(t)
   thanks$thank_family(t)
   thanks$thank_friends(t)
   thanks$thank_misc(t)

   % show output
   thanks$flush(t)
 end start_up


thanks = cluster is create, thank_god, thank_profs, thank_xsoft, thank_family,
                    thank_friends, thank_misc, flush

   % overview
   % thanks is a mutable datatype used to express gratitude.

   % A(r): r -> A
   %
   % A typical thanks is {t1, t2, t3, ..., tn} where ti is a thank.
   % A(r) = { r[i] | low(r) <= i <= high(r) }

   % I(r): r -> bool
   %
   % I(r) = true

   thank = string
   rep = array[thank]

   create = proc(s: string) returns (cvt)
      r: rep := rep$new()
      rep$addh(r, s)
      return(r)
    end create

   thank_god = proc(t: cvt)
      rep$addh(t, "First and foremost, I thank my Lord and God for being " ||
                  "able to finish this thesis and my five years at M.I.T. " ||
                  "May they have been to His glory.\n")
    end thank_god
```

```
    thank_profs = proc(t: cvt)
       rep$addh(t, "I thank Professor David Karger, my thesis advisor, " ||
                   "for his patience, insights, feedback, and " ||
                   "relentless pursuit of quality writing.\n")
       rep$addh(t, "I thank Professor Peter Elias, my academic advisor, " ||
                   "for taking me through my years in Course VI.\n")
       rep$addh(t, "I also thank Professor Fernando Corbato and Professor " ||
                   "Markus Zahn for their involvement in making my 6A " ||
                   "co-op assignments enjoyable and worthwhile.\n")
     end thank_profs

    thank_xsoft = proc(t: cvt)
       rep$addh(t, "I thank Andy Gelman, my boss at XSoft, for the " ||
                   "opportunity to work on such a great project and for " ||
                   "his input, advices, and suggestions.\n")
       rep$addh(t, "I also thank Mike Wilkens, Beth Bryson, Matthew " ||
                   "Christian, and my other colleagues who have made my " ||
                   "thesis work possible.\n")
     end thank_xsoft

    thank_family = proc(t: cvt)
       rep$addh(t, "I thank mom and dad for their mental, emotional, " ||
                   "financial, and otherwise psychological support " ||
                   "and encouragement.\n")
       rep$addh(t, "I thank my sister for being there when I needed " ||
                   "stress relief. It was very necessary for this thesis.\n")
     end thank_family

    thank_friends = proc(t: cvt)
       rep$addh(t, "I thank my friends who have taken the time and energy " ||
                   "to give me encouragement and assistance, including, but " ||
                   "not limited to, Fred Chen, David Chen, Jim Derksen, " ||
                   "Tony Eng, Wendy Fan, Edwin Foo, Ellen Hwang, " ||
                   "Michael Kim, Faydeana Lau, Jasen Li, Janet Liu, " ||
                   "Dave Sun, Chris Tserng, and Karen Zee.\n")
     end thank_friends

    thank_misc = proc(t: cvt)
       rep$addh(t, "I thank Anne Hunter -- not possible to have done this " ||
                   "thesis without her help!\n")
       rep$addh(t, "Also thanks to Lydia Wereminski of the 6A office who " ||
                   "coordinated all the details of the 6A co-op program.\n")
     end thank_misc

    flush = proc(t: cvt)
       for s: thank in rep$elements(t) do
          stream$putl(stream$primary_output(), s)
        end
     end flush

end thanks
```

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Throughout history, one direct consequence of technological advances has always been a dramatic decrease in the perceived distance between two objects. New transportation methods have effectively shrunken the world so that remote corners of the earth are no longer remote. As the distances between objects grow smaller, one's exposure to the world grows broader. For instance, journeys that used to take an entire lifetime to complete can now be done within a matter of hours or days. As a consequence, one can enjoy a broad range of experiences and knowledge impossible before now. Unfortunately, the expansion of the horizon comes with a price: complexity. One has to manage more things and think on global terms. One learns foreign languages and discusses international issues.

In the same way, technological advances in communication and telecommunication devices have revolutionized the very fabric of living. Cellular phones, video conferencing, and the Internet are but a few examples of how new technology is redefining the scope and the amount of information accessible to the average person. At the same time, there is more pressure to process the information at a quicker rate, so that others can make use of the information in a timely manner. These changes have all contributed to the dissemination of information—vital to educated decision-making and the further development of knowledge and technology—as well as the increasing complexity of information processing, information storage, and information retrieval.

1.1 The Problem of Information

In a very real sense, the consequence and the goal of technology *is* information. With each new discovery, one learns more about the world. With every trip to Africa or Australia, one finds more interesting tidbits about the local color and culture. And now that the Internet and the world-wide-web (WWW) have attained world-wideness, there is a deluge of information available to anybody who cares to look for it. For example:

- A keyword search at the popular Alta Vista WWW search site[1] on "The President of The United States of America" turned up more than 3,000,000 webpages. Although many of these pages contain useful and educational material on the President, there are undoubtably many others that are irrelevant for information-gathering purposes.

- A quick survey among friends here at M.I.T. reveals that receiving close to 50 email messages[2] a day is not unreasonable, even though most of these messages (around 70-80%) are not particularly useful.

- The USENET newsgroup, `rec.arts.anime.misc`, has over 4000 posts constantly. Many of these postings are advertisements that are neither related to anime nor interesting. `Kill` files can be useful in this case, but they are not very intelligent.

The problem of information is really the problem of over-information. There is so much information out there—how does one spend the least amount of time finding the relevant information that is useful? In other words, how can one filter out the noise in the input to arrive at a high signal-to-noise ratio? The motivation is not one of laziness. Rather, efficiency and the limited nature of one's resources (*e.g.*, time) demand an answer. If a person spends 15% of each workday filtering out unwanted

[1] `http://www.altavista.digital.com`.

[2] If 50 sounds like a reasonable number, just imagine having 50 pieces of U.S. mail waiting for you each day in your mailbox.

or useless information, in a year he would have spent almost 8 work weeks (almost two months) doing nothing else! Time is not the only problem: having to manually filter out the noise from the signal is a difficult job that requires tremendous patience and meticulous attention to details. Otherwise, a substantial portion of the relevant information might be accidentally deleted as a result of careless noise filtering.

Because of the increasing power and affordability of computer hardware and the user-friendliness[3] of software in recent years, there is a commercial and societal trend to "go digital." This involves making information available in a format easily stored, transmitted, and displayed by computers. Reasons for this trend include: (1) large amounts of data can be stored digitally and retrieved quickly using inexpensive storage devices such as hard disk drives, (2) digital storage is often more durable than other means of storing data (*e.g.*, paper, microfilm) due to the ease with which digital data can be replicated, and (3) digital information can be made easily accessible to computer users via WWW, ftp, or gopher, among other methods of data transfer. The digital trend exacerbates the deluge of information by making even more information available in even easier to access formats.

1.2 Possible Solutions

Technical solutions to this problem cannot be esoteric and complicated (at least not in its design or usability), because a large segment of the non-technical population will be using these solutions. Instead, there is a real need for a solution that is both technically feasible and socially acceptable. In other words, the solutions must be a reasonable solution that the majority of the population can learn to use and grow to like.

There are three general approaches to solving the problem of having too much information. They are:

1. **Remove the information sources.** The over-information problem can be

[3]Entire theses can and have been written on this subject. Suffice to say that this label is used in a relative, not absolute, sense.

avoided altogether by removing one's sources of information. Although this may seem somewhat extreme, it is actually practiced by some. However, another set of problems arises out of this lack of information. How does one keep in touch with the society? Is it possible to be isolated in such a manner if the society wishes to continue advancing technologically?

2. **Sort the information.** Sorting arranges documents in some useful order. This provides valuable organization and structure that can be save time. For example, emails may be sorted by senders or dates so that the reader can read the most important emails first or delete emails from a certain sender altogether. As another example, newsgroup articles might be sorted into subject threads so that a reader can delete uninteresting groups of articles immediately. However, sorting does not reduce the amount of information that the human reader must process ultimately. This is both an advantage (*i.e.*, the reader has full access to all documents) and a disadvantage (*i.e.*, the reader has to prune the documents manually).

3. **Filter the information.** Alternatively, the information can be filtered, which involves blocking or removing certain parts of the source from the human reader. `KILL` files for newsreaders are an example of filtering: unwanted articles with certain keywords are automatically deleted without human intervention. The main advantage of filtering is that it reduces the amount of information that a user has to read or manage. At the same time, the user loses control over the filtering process, and certain information may be lost involuntarily. Thus, he might not have absolute confidence in the filtering process.

1.3 The Summarization Solution

It is clear that there are various trade-offs for each of the three approaches. Filtering is arguably the most attractive approach, combining some possible intelligence with substantial information reduction. Unfortunately, filtering can be extremely expen-

sive. One can imagine a scenario where the filter has to "understand" a document before applying a filter to reduce the size of the document—such language-dependent information processing can take a long time. The goal, then, is to find a filtering method that works well *and* is fast.

Upon closer inspection, the filtering approach divides naturally into *selection* and *summarization* filtering. In both methods, source documents are examined and contextual information is generated for each source document. However, selection filtering eliminates information on the document-level by filtering out all documents that do not meet the specified criteria. A subset of the original documents is then returned to the human reader. In contrast, summarization filtering eliminates information on the sentence-level. This means that instead of removing entire documents, only part of each document is filtered out.

Selection corresponds more closely to the general or classic idea of filtering: that the filtered output will be a smaller subset of the input documents. The main disadvantage is that the human reader will have no control over document elimination, which may be good (it saves him time) or bad (important documents get filtered out without notification). In contrast, summarization filters on a finer-grain level by providing a summary for each document. A summary, by conventional definition, is the document's most relevant ideas and/or sentences after all the other sentences have been eliminated or removed from view. Such an approach eliminates the danger of throwing away large chunks of text.

This author feels that summarization has the following advantages over selection filtering:

1. **Control.** A summarizer will not eliminate entire documents, so the human reader ultimately retains the control for discarding irrelevant documents. This conservative filtering policy guarantees that such decisions are made by other beings with more intelligence than an automatic summarizer.

2. **Hierarchy Flattening.** A good summary has the effect of flattening the hierarchy of each document into a few sentences that are indicative of the entire

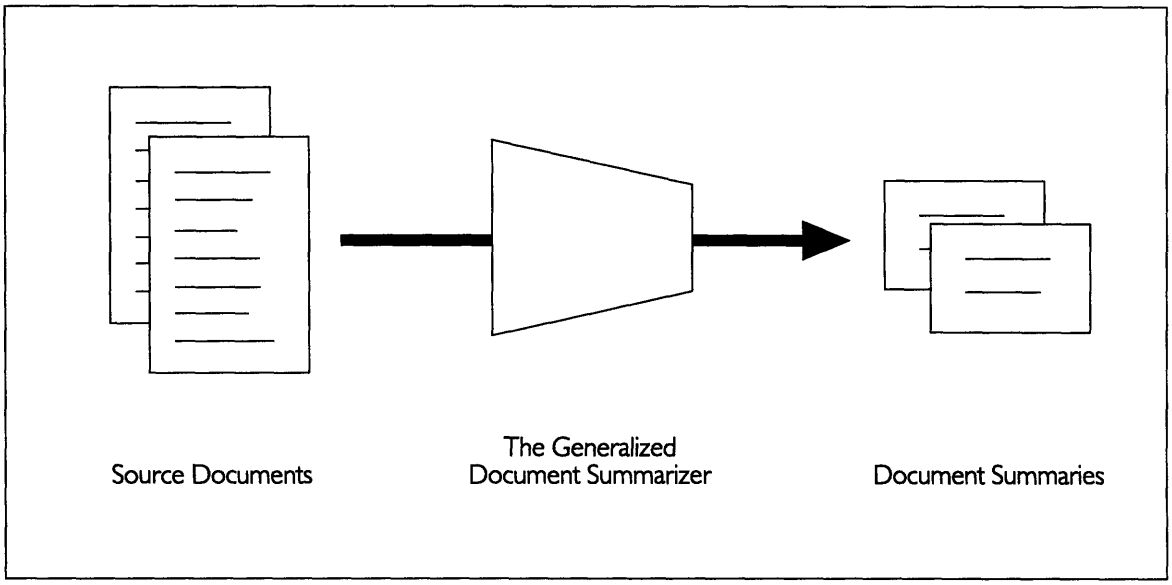

Figure 1-1: The Generalized Document Summarizer

document. Another way to think of this advantage is to realize that a summary is the result of not only summarizing the words in a document, but also the structure of the document. (Indeed, sometimes the structure of a document can give better insights into the main idea than the sentences can.)

In short, document summarization is a good way to process large amounts of information. Summarization combines intelligent filtering and control with good speed that is attainable by exploiting particular features or structures of the source.

1.4 The Generalized Document Summarizer

The Generalized Document Summarizer (GDS) described in this thesis is an attempt to instantiate a document summarization solution. Based on a plaintext summarizer [5], the GDS implements HTML capability as well as internal structures to support new file formats in the future.

Figure 1-1 is a high-level view of the GDS' function: to generate a condensed, short, and useful version of the original document. This is done by extracting sentences from the original document. The GDS calls such a collection of sentences a summary, even though, strictly speaking, a document summary need not be restricted

to only extracted sentences.

By allowing the user to customize certain aspects of the summarization process, the GDS can tolerate a number of input document formats. For example, it can be made to summarize HTML documents, business letters, etc. Furthermore, the GDS gives user fine-tune control over the application and the relative weights of various heuristics during the actual summarization process. In this sense, the GDS is both a "generalized document" summarizer, as well as a generalized "summarizer of documents."

The GDS is more than a proof-of-concept. It is also being used to evaluate the effectiveness of the summarization heuristics and the usefulness of the generated summaries.

1.5 Organization of This Thesis

- **Chapter 1**, "Introduction," includes a discussion of the problem addressed by this thesis, some possible solutions, and the organization of this paper.
- **Chapter 2**, "Background," reviews the history and background of summarization technologies and related research in the field.
- **Chapter 3**, "Summarization Heuristics," discusses the heuristics used by the GDS and presents guidelines for developing other heuristics.
- **Chapter 4**, "Overview," provides a big picture of the GDS: how it works and how to use it.
- **Chapter 5**, "Tokenizing the Document," takes a closer look at the document tokenization process and the associated token vocabulary.
- **Chapter 6**, "Generating the Summary," discusses the details of scoring sentences and the extraction of summary information from source documents.
- **Chapter 7**, "Evaluation," details the performance of the GDS using a number of criteria.

- **Chapter 8**, "Conclusion," contains suggestions for future work in the summarization field.

Chapter 2

Background

Summarization, as a technique for handling large amounts of information, has been around for a long time. This chapter discusses some of the relevant research efforts that have preceded the work on the GDS, as well as the advantages and disadvantages of various summarization approaches.

2.1 Linguistics and Computer Science

Research in the summarization field falls into either the linguistic camp or the computer science camp. Linguistic research usually focuses on determining specific semantics of words and phrases and answering the the general question of *what makes a summary a good summary?* [10, 6] Typically, heuristics are developed as the consequence of linguistic research efforts. These heuristics can then be incorporated into summarizer programs like the GDS. For example, research 0reveals that headings are useful in recalling unfamiliar text or topics [6]. This conclusion might motivate adding a summarization heuristic that favors heading sentences in a document's summary.

On the other hand, automatic text summarizers like the GDS present a set of problems and challenges that are very different from the linguistic aspect of summarization. Using the computer to summarize documents requires answering the question: What is the best way (*e.g.*, most efficient, most comprehensive, using the least amount of memory and resources) to represent the document?

Previous attempts to answer this question can be classified into two broad classes: natural language processing and word statistics analysis. While the two are at opposite ends of a spectrum and can both be incorporated in varying degrees, it is useful to discuss the advantages and disadvantages of these approaches as two extremes.

2.2 Natural Language Processing

Natural language processing involves trying to gain some "understanding" of the sentences in the source document. It is focused on the study of the language semantics, not just the syntax, of the source document. This usually requires sophisticated, "artificially intelligent" algorithms and elaborate data structures to hold sentence meanings and other information. The summary can then be generated directly from this understanding.

The main advantage of this approach, if the summarizer *is* capable of some language understanding, is that summary sentences do not have to be limited to the original sentences that appear in the document. They may be composites of words or sentence fragments from the source documents, or they may be original sentences created by the summarizer. Such summaries have the potential of being very readable, because the summarizer has full control over what kind of sentences to generate for the summary.

The main problem with natural language processing is that this is still very much an area of active research where no one knows a good method to approach it. Achieving some "understanding" of the document is difficult, and creating sentences is even harder. Furthermore, natural language processing is usually computationally expensive and therefore not very practical, except in limited domains where the content is predictable and a simplified set of semantics is adequate [7, 4]. Furthermore, natural language processing systems must necessarily be tailored to each language. Supporting multiple languages requires a tremendous amount of effort.

2.3 Word Statistics Analysis

In contrast, the word statistics approach does not attempt to gain an understanding of the natural language. Instead, the focus is on syntax, and summary sentences are selected from the set of existing sentences in the source document. Summary sentence selection is usually based on various clues such as word occurrences in the document. For example, while parsing an article on wombats, the summarizer might notice that *wombat* is a frequently encountered word. Consequently, sentences containing *wombat* might be included in the article's summary—and all of this would be done without the summarizer having any knowledge of what a wombat is.

There are a few advantages to word statistic analysis. First, the process is computationally inexpensive relative to the natural language processing approach. With the use of some language-dependent input parameters, a word-analysis summarizer can summarize documents in many languages without further modifications. Lastly, by extracting the summary sentences from the original document, the entire problem of generating meaningful summary sentences is avoided.

The main disadvantage of word statistics analysis is that summary sentences must originate from the source document. Consequently, the summary cannot be improved by combining sentences or creating new sentences.

Past efforts using the word statistics approach include Salton et al [11], Kupiec et al [5], and Paice et al [9].

Chapter 3

Summarization Heuristics

heuristic, adj., serving to guide, discover, or reveal; valuable for empirical research but unproved.

Since the goal of the GDS is to summarize documents by extracting sentences, rules need to be established to differentiate between sentences. Heuristics in the GDS are guidelines used by the summarizer to determine which sentences would make good summary sentences. Obviously, the choice of heuristics has a direct impact on the quality of the document summaries. This chapter presents some guidelines for developing good heuristics for summarization and describes some of the issues involved in implementing heuristics in the GDS. In addition, all heuristics used by the GDS are explained in detail.

3.1 In Search of Quality Summaries

Plainly stated, the purpose of the GDS is to generate summaries. Inherent in that statement is the assumption that these summaries would be "good" summaries. This is perhaps the most essential trait for a summarizer. Unfortunately, no definitive method exists for measuring the quality of a document summary. Some might suggest this demonstrates a lack of understanding of the nature of information and/or other linguistic issues. More likely, this lack of quantification is due to human diversity.

Different people will absorb information in different ways; while one person studies the table of contents, another may find the index more approachable.

Nevertheless, the notion of a "quality" summary exists. This author proposes the following characteristics as essential:

1. **High Relevance Assessment Potential.** One reason people read summaries is because they cannot or do not wish to spend the resources necessary to read an entire document. Therefore, a good summary must provide good clues as to how relevant the original document is. The reader should be able to use the summary to assess how interesting the actual document will be.

2. **Detection of Masquerading Sentences.** People not only read summaries to find documents that they are interested in reading, they also read summaries to find documents that they are *not* interested in reading. A good summary should reveal whether a document is "masquerading" as being useful. The difference between this characteristic and the first characteristic is that the first requires a summary to be *sufficient* for the purpose of relevance assessment, while this requires a summary to be *complete* as well. In other words, a good summary should guarantee that the source document is not about anything outside of the summary.

3. **Low Noise Level.** A good summary should have as little noise as possible. Noise, in the context of a document summary, includes sentences with too much detail and sentences that do not make sense by themselves. The former is bad because the reader of a summary may not be interested in the details. The latter is undesirable because if a sentence is taken out of context and not understandable by itself, then the reader has very little use for it.

As mentioned earlier, the GDS does not generate summary sentences from scratch. Instead, it selects summary sentences by applying a number of heuristics to each sentence. Operating on the sentence-level simplifies the summarization process, since each sentence can be examined independently. The drawback is that the GDS may

be too low-level (*i.e.*, not global enough) to provide summaries in which the sentences support one another and blend together. This is an inherent problem with sentence-level summarization. However, by presenting the summary sentences as a collection of sentences instead of as a summary paragraph, this coherence issue is avoided.

The heuristics used by the GDS were designed to minimize noise while presenting a relevant and indicative summary. Although guaranteeing a thorough summary (characteristic No. 2) is difficult, finding relevant sentences is achieved with heuristics that favor emphasis in the document and strategically-placed sentences.

3.2 Heuristics Development Guidelines

The GDS relies on its heuristics to pick out good summary sentences. Each heuristic describes the relationship between a *sentence feature* and the likelihood of that sentence being a good summary sentence. A sentence feature can be the number of words in a sentence, the types of punctuation in a sentence, and any other discernible sentence trait.

The typical heuristic is first developed as an intuitive hypothesis, and then tested in empirical experiments to determine its effectiveness. Tests are done using a set of test documents (the *corpus*) along with corresponding summaries, where the corpus is examined to see if the relation described by the heuristic is indeed observed in the corpus summaries.

A good summarization heuristic should aim at differentiating certain sentences from the rest of the document. In particular, each heuristic should find sentences that have one or more of the following traits:

1. **Relevance to the Document.** Sentences that convey the main ideas of a document are good candidates for the summary. Sentences that indicate the relative importance of a document within a set of documents are also useful.

2. **Relative Uniqueness.** Some sentences are good summary sentences not because they express a document's main idea or topic, but because they can

provide other information that may be useful to the reader. For instance, even though equations are usually too detailed, they may be good candidates for the summary of a mathematics paper. Hypertext links in HTML documents may provide another kind of summary information independently of the main ideas.

These traits were considered when the heuristics for the GDS were developed. They can be used as the starting point for developing additional summarization heuristics.

3.3 The GDS Approach to Heuristics

As mentioned in Section 3.2, heuristics are used to specify relationships between sentence features and the likelihood of a sentence being included in the final document summary. This section explains how evaluation of the heuristics translates into sentence scores.

To understand sentence scoring, think of each sentence as having a feature bit-vector. Each entry of this feature-vector corresponds to the existence of a sentence feature (as dictated by some heuristic). Intuitively, one can give each sentence a score based on how many features it has. Sentences with lots of features would score higher and be included in the document summary.

Of course, some features may be more important than others, so the existence of certain features may merit more points. Given a sentence and its feature-vector entries $x_1, x_2, ..., x_k$, this can be done by computing the score for each sentence as a weighted sum:

$$\sum_{i=1}^{k} a_i \cdot x_i$$

where $a_1, a_2, ..., a_k$ are weights corresponding to the k features. Even in this simple model, however, many variations exist which can change the score substantially. The GDS implements a variation of this scoring model, choosing to deviate in three dimensions: fractional vs. probabilistic weights, boolean vs. discrete features, and

independent vs. cutoff scoring.

3.3.1 Fractional vs. Probabilistic Weights

The first issue involves how the weights $a_1, a_2, ..., a_k$ are determined. A simple approach is to use fractional or integral weights. In such a case, each feature's relative importance (and, consequently, its effect on a sentence's score) is dictated by the equation:

$$r_i = \frac{a_i}{a_1 + a_2 + \cdots + a_k}$$

where r_i is the importance of feature i out of 100%. One advantage of having fractional weights is that the weights can be as arbitrary as desired. Since each a_i is nothing more than a number without a definite meaning, there is no problem with modifying the weights to account for feature dependence or other factors. Also, any individual feature can be turned off by simply assigning zero to its weight.

The second approach is to use probabilities. (This is also the GDS' approach.) Note that this is merely a particular instance of weighting. Instead of having arbitrary weights, one computes the probability that sentence s will be included in summary S given the k features. Using Bayes' rule:

$$P(s \in S \mid F_1, F_2, ..., F_k) = \frac{P(F_1, F_2, ..., F_k \mid s \in S) \cdot P(s \in S)}{P(F_1, F_2, ..., F_k)}$$

where F_1, F_2,...,F_k are the k features. Assuming conditional and full independence of the features (*i.e.*, the naive Bayes' classifier), the above equation becomes:

$$P(s \in S \mid F_1, F_2, ..., F_k) = \frac{\prod_{i=1}^{k} P(F_i \mid s \in S) \cdot P(s \in S)}{\prod_{i=1}^{k} P(F_i)}$$

Each of the above probabilities can be computed or distilled from the training corpus, leading to a clean method of determining summary inclusion or exclusion.

The main obstacle to using the probabilistic model is that the probabilities must

be determined. In the case of the GDS, this was done by surveying a large number of plaintext and HTML documents with human-generated summaries.

3.3.2 Boolean vs. Discrete Features

So far feature-vectors have been presented as bit-vectors. However, they can be generalized to "discrete feature-vectors" capable of holding more than just a boolean value. Such a feature-vector allows some n discrete values in each entry and can differentiate between n degrees of "presence" for each feature. For example, instead of recording whether a sentence has capital letters or not, the discrete feature-vector can record the number of capitalized words in each sentence. In contrast, by flattening a feature-vector into a true bit-vector, each entry is forced to register one of two values (a boolean). Consequently, features are either present or absent in a sentence.

The GDS uses only boolean bit-vectors to score each sentence for two primary reasons. First of all, detecting the presence or the absence of a feature may be easy, but differentiating between many degrees of presence can be hard. Not all features will map nicely into a set of discrete values. Secondly, it is not clear that the fine-tune control afforded by the discrete feature-vector will substantially improve the generated summaries. This is because most heuristics implemented in the GDS are not capable of specifying the exact effects of a feature's presence on such a fine level.

Discrete feature-vectors are not completely unused, however. The next section will show discrete feature-vectors used in cutoff scoring (where the more general feature-vectors are collapsed into bit-vectors).

3.3.3 Independent vs. Cutoff Scoring

In independent scoring, the values in each sentence's feature-vector are solely determined by that sentence. If the sentence has features i, j, but not k, the entries for i and j would be set to *true*, while the k entry would be *false*. Since the probabilities are predetermined, the sentence's score can be calculated without further ado. One immediate advantage is that the entire document does not need to be in memory or

even accessible to the summarizer all at once.

The other approach, cutoff scoring, determines the values of the sentence bit-vectors on a more global scale. The existence of a particular feature in a sentence does not automatically guarantee that the corresponding entry in the feature bit-vector will be set to *true*. Rather, for each feature that employs cutoff scoring, all sentences are ranked by how much of the feature is present in the sentence (reminiscent of discrete features). This score is first recorded as one of a discrete number of values. From those values, a user-specified n sentences will be marked *true*, while all others below the cutoff point will be *false*.

The GDS uses cutoff scoring to flatten the feature-vectors into pure bit-vectors. Many sentences have similar sets of features, and independent scoring would give them composite scores with minuscule differences. The very presence of such tiny differences is an indication that the summarizer cannot tell those sentences apart reliably—is a sentence with a score of 0.02560 really "better" than a sentence with a score of 0.02559? Given the rough resolution of the heuristics, it is not clear at all what the answer is.

Cutoff scoring works because the sentence differentiation is done separately for each feature *and before* the sentences are scored. It has to be done before the scoring because once the sentences have been given a composite score, differentiation becomes much more difficult.

3.4 Incorporating Document and Formatting Information

By handling formatted documents, the GDS improves upon the plaintext-only XLT Summarizer. However, document information (including the formatting) can be found in plaintext documents as well. The heuristics in this chapter all use one or more of the following four classes of document information:

1. **Emphasis Information.** Certain formats (or formatting tags) indicate emphasis within a document. Characteristics such as boldface, italics, underline, superscript, and subscript are all used to indicate the importance of certain words or sentences. Emphasis information is not restricted to actually formatting information: Uppercase words in a plaintext document also indicates some kind of emphasis. This kind of information can indicate the author's preference or focus so that summary sentences better reflect it.

2. **Structure Information.** Section headings, footnotes, and paragraph breaks are a few examples of structure information. The summarizer can use this information to (1) determine where paragraphs end, (2) handle special text such as headers and footnotes differently from the rest of the test, and (3) find the table of contents for a long document.

3. **Relationship Information.** Some formatting tags define the relationship between various documents. One obvious example is the hypertext link tag (`<A>`) in HTML, which links the current documents with a few target documents that might be of interest to the reader. This is different from structure information in that the former indicates *inter*-document relationship, while the latter indicates *intra*-document relationship (*e.g.*, relationship between the various parts of the same document).

4. **Context-Based Information.** The number of words in the document, the number of sentences, the actual diction, etc. are all part of this information. Plaintext heuristics (Section 3.5) usually make use of this information.

The GDS implements heuristics from all of the classes described above. The heuristics were developed and implemented as general rules that can apply to a number of different formats (such as HTML and LaTeX). However, they are described below in the context of HTML where applicable because (1) HTML summarization is expected to be the main use of the GDS, and (2) this provides a familiar environment in which the readers can relate.

3.5 Plaintext Summarization Heuristics

The following heuristics were developed for plaintext summarization: (1) sentence length, (2) direct theme, (3) sentence position, (4) keywords, and (5) uppercase words. All, with the exception of the uppercase feature, are very useful in the summarization of HTML and other document types as well. This can be attributed to the heuristics' ability to capture formatting-independent information from documents.

The probabilities for these five features were derived from training the summarizer on thousands of technical papers and journal papers. Therefore, the GDS is exceptionally good at summarizing such papers. (In some sense, technical papers are easier to summarize than most other documents, because they have very well defined sections[1].)

3.5.1 The Sentence Length Heuristic

Particularly short or long sentences are generally not very good summary sentences, although longer is better than shorter. Short sentences consisting of fewer than three words, for example, probably do not convey enough information to be a summary sentence. At the same time, sentences with lots of words usually have too many details to be useful in a summary.

3.5.2 The Direct Theme Heuristic

Direct theme keywords (not to be confused with the keywords heuristic from Section 3.5.4) are words or phrases that appear often in a document. They can be found by compiling a list of words and counting each word's occurrence in the document. The GDS keeps a list of common *stop-words* that should be excluded from word frequency analysis. These include articles (*i.e.*, the, a) and other words that have very little meaning outside of being a language construct. If these words were included in the analysis, then the results would be undesirably skewed heavily in favor of

[1]Technical papers (very conveniently) have a title, an abstract, a body, and a bibliography to mark, respectively, the beginning, a summary, the body, and the end, of the document.

using them as direct theme keywords. The remaining words are recorded individually as words and collectively as two-word to five-word phrases. These phrases and their relative frequency can provide good insights into a document's topics and main ideas. For instance, if a single-page document of 200 words has 20 occurrences of "PowerPC microprocessor," it is a good bet that the document is about PowerPC microprocessors.

Therefore, sentences that contain a document's direct theme keywords are usually excellent summary sentence candidates.

3.5.3 The Sentence Position Heuristic

Most documents are subdivided into sections and paragraphs. By exploiting the logical structure of well-written documents, the GDS places more emphasis on sentences at the beginning or the end of paragraphs. The idea is that these sentences provide the main ideas of the paragraphs while staying away from the details. The position of the paragraph within a document can also affect how a sentence is scored with this heuristic. Paragraphs at the beginning and the end of a document typically provide better summary sentences.

3.5.4 The Keywords Heuristic

This heuristic values sentences that contain certain "hint" words and phrases. In English documents, for example, the summarizer looks for phrases such as "this paper" and "in conclusion," since they are usually followed by sentences that contain a good amount of overview but not too much detail. In order to keep the GDS language-independent, this list of word is kept separate from the summarizer in a user-customizable file.

3.5.5 The Uppercase Feature

In plaintext documents where there are few formatting options, uppercase letters are often used to indicate boldface or to attract attention. Although sentences with

boldface words are not necessarily good summary sentences, this feature can be very useful in determining the title sentence for a source document. It should be pointed out that even in HTML documents uppercase words are often used to emphasize or bring attention to a sentence.

3.6 The Challenges of HTML Summarization

HTML summarization is of particular interest right now because of the unprecedented expansion of the world-wide-web. At the time of the GDS' development, HTML 3.0 and above did not exist. Consequently, the GDS implemented a subset of the current HTML tags. For a list of supported HTML tags, see appendix A.

Adopting HTML for the GDS added an important dimension to the summarization capabilities of the GDS. However, there were many challenges along the way, and many of them still do not have very satisfactory solutions at this point. Some of these challenges are:

1. **Assigning Semantic Meanings to HTML Tags.** Because of the sheer number and the variety of HTML tags, determining the semantic meanings for each tag was not a trivial task. While some tags overlapped semantically (*e.g.*, `<B>` and `<STRONG>`), many tags differed just enough to make classification difficult. In the GDS, HTML tags are roughly divided into a handful of categories for this purpose of semantic classification. In theory, this may be a gross simplification. But in practice, this proved to work well.

2. **Incorrect HTML Tag Usage.** In many of the webpages surveyed, HTML tags were used incorrectly (both syntactically and semantically). The reasons range from lack of HTML knowledge to wanting to achieve a very specific and particular look. This problem is exacerbated by the number of non-standard tags supported by popular web browsers like Netscape.

3. **Webpages are Rarely Organized or Structured.** At least, not in the same logical and well-structured style as most technical papers. However, HTML doc-

uments do generally fall into a number of formats, and these could be "hard-wired" into the GDS for better HTML summarization.

Like plaintext summarization, a set of orthogonal and independent heuristics are employed in the scoring computation. Any heuristic can be turned on or off for either debugging purpose or customization. The following sections discuss the HTML heuristics implemented in the GDS.

3.6.1 The HTML Boldface Heuristic

This feature checks for the existence of formatting tags that denote boldface or strong wording. Sentences with more boldface words are usually better summary candidates than sentences with few or no boldface words. In order to filter out trivial instances[2] of boldface words, only sentences with more than a threshold number of boldface words are considered.

The HTML tags that fall under this category are <B>, <STRONG>, and <BLINK>. Although the last tag does not actually generate boldface text, the effect is similar (*i.e.*, an emphasis used to draw attention).

3.6.2 The HTML Emphasis Heuristic

The emphasis feature favors words "emphasized" by HTML tags such as <EM> or <I>. The emphasis feature is very similar to the boldface feature. In fact, the GDS does not make any semantic differentiation between these two currently. Note that the <CITE> tag is considered an emphasis tag. Although this is not the original purpose of this logical HTML tag, <CITE> is classified as such because it is often misused to achieve the italics look.

[2]Such as the usage of a boldface drop-cap that begins a document. For example: "**O**nce upon a time..."

3.6.3 The HTML Hypertext Links Heuristic

Ultimately, the quality of a HTML summarizer is determined by its ability to provide useful summaries for webpages. Since people use the web to gather information or to find answers to particular questions, hypertext links are very important; if a webpage does not contain the information one seeks, the next best thing would be to point to a few pages that might provide the information.

The hypertext links heuristics is based on this idea and values sentences with hypertext links. Ideally, the hypertext itself (*i.e.*, the text between the `<A>` and `</A>` pairs) would provide some clue to the topic and subject of the target webpage. This would be useful in giving the reader: (1) an idea of what pages can be accessed from the current HTML document, and (2) some idea of what the target is about.

Unfortunately, many hypertext links do not provide this kind of summary information. For example, the word *here* is often used as the hypertext to link to webpages. ("Click *here* to go to...") This is not a relevant hypertext. As a result, one must be careful when examining sentences with hypertext links, lest one with very little meaning is returned as part of the summary. The GDS achieves this by ignoring hypertext links with three or fewer words.

3.6.4 The HTML Lists Heuristic

HTML lists are generally used for two purposes:

1. Providing Details. Examples include a list of features for a product, a list of supported file formats for an application program, and a list of event highlights from a company ski trip. These lists typically consist of only a few words or phrases per line.

2. Organizing Big Ideas. Examples include a list of words and their definitions, a list of job openings at a company, and a list of recommended restaurants in the San Francisco bay area. Each item in these lists are usually long, packed with explanation and information.

If a list is used to provide details, it is usually a good idea to leave it out of the summarization process. Most items in such a list would be incomplete sentences, rendering them useless as part of a summary (unless the entire list is included). On the other hand, lists of the second type could be very useful. These lists usually have long sections of one or more complete sentences that correspond well to paragraphs (and therefore can be treated as paragraphs).

Unfortunately, differentiating between these two types is very difficult. Classification based on the length of a list's items is unreliable at best, and there is no guarantee that any list will provide a good summary compared to non-list sentences. In general, sentences found within lists are not considered good summary sentences. However, some webpages consist of nothing but lists. The GDS performs poorly with these pages.

3.6.5 The HTML Headings Heuristic

Like the HTML lists heuristic mentioned above, the presence of the heading HTML tags within a sentence will decrease its probability of being a summary sentence. The rationale behind this scoring is that although headings (*i.e.*, `<H1>`, `<H2>`, etc.) are used to label different sections of a webpage, they often contain little or no actual information. Many headings will be nothing more than just "Introduction," "Conclusion," or "Welcome to Our Page!" Heading tags are sometimes used to increase the font size for a phrase or a sentence, but this is relatively infrequent.

3.7 Other Heuristics

Documents with formatting information can provide extra insights into which sentences are good summary sentences as well as hints on how to avoid certain bad sentences. With HTML, the following heuristics proved to be useful:

1. **Start-of-Text and End-of-Text Detection**. The start-of-text is the place in a document where the body of the actual text begins. In a technical article, for

example, the start-of-text would come after the title page, the abstract, and any preamble. Similarly, the end-of-text marks where the body text ends. This is usually right after the concluding paragraphs but before the appendices and the index. Finding the beginning and the ending of the text body is useful because: (1) the title sentence (and the abstract, if one exists) is usually before the start-of-text, and (2) very few sentences after the end-of-text are good candidates for the summary, because typically only the index and appendices (or, in HTML documents, copyright and webmaster credits) come after the end-of-text. With plaintext documents, detecting the start-of-text involved keeping track of how many consecutive "normal" sentences with "normal punctuation" have been encountered in the document. In a similar fashion, the end-of-text is detected by looking for an appendix or an index. HTML provides the GDS with extra information (*e.g.*, the `<HEAD>` and `<BODY>` tags) for better detection of both start-of-text and end-of-text.

2. **Title Detection.** For the title sentence, the plaintext summarizer looked for a sentence before the start-of-text that: (1) is at least of a certain predetermined length, and (2) contains uppercase words or other emphasis. This title detection heuristic is improved in HTML summarization because `<TITLE>`, `<H1>`, and other fontsize tags provide good indication of where the title sentence may be. Note that the text inside of the `<TITLE>` tag may not be a good title sentence. Since this is the text that appears as the browser's window title, it is often quite unrelated to contents of the current webpage. Typically, the first emphasized[3] sentence after the `<TITLE>` is the best title sentence in the majority of webpages surveyed.

3. **Paragraph Detection.** In plaintext documents, there are two ways to denote the end of a paragraph: (1) use a single carriage-return (CR) between paragraphs but none between the lines in each paragraph, or (2) use two CRs between paragraphs and one CR between the lines in each paragraph. If unspec-

[3]by heading, boldface, or fontsize tags.

ified, the GDS automatically detects between the two possibilities by examining the average number of words between CRs and the frequency of CRs in the document. In HTML documents, all CRs are ignored and only paragraph-marking tags like <P> and
 are used to denote paragraph boundaries. This, of course, leads to better paragraph detection because the GDS now knows exactly where the paragraph breaks are.

4. **Avoiding Navigation Menubars.** "Navigation menubars" are lines of hypertext links at the beginning or the end of a webpage that helps one navigate through a collection of webpages. One typical navigation menubar looks like:

   ```
   Home | Prev | Next | Help | What's New
   ```

   Navigation menubars present two problems: (1) the GDS reads this line as one single sentence, since it lacks a sentence-ending punctuation. However, this is not a sentence and would be a poor summary sentence; and (2) the links heuristic would favor this sentence because of the large number of hypertext links.

   A heuristic was developed to specifically avoid navigation menubars. For each sentence, the GDS examines (1) the ratio of actual alphanumeric words to the number of non-sentence-ending punctuation marks, (2) the number of hypertext links present, and (3) the number of words in the actual hypertext links. If these numbers are above a certain threshold, the sentence is eliminated from summary consideration.

   This heuristic is an example of the kind of problem unique to a particular document format. Unfortunately, the GDS does not have a large number of such heuristics to handle the many possible document formats, but the scoring mechanism allows for such extensions where and when they are deemed necessary or useful.

3.8 User-Specified Summarization Focus

The GDS also provides a method for users to specify special words that are of interest to them. When the GDS encounters these words, they are given special consideration. The user can ask that sentences (or paragraphs, if so desired) containing those special words be given higher or lower scores. This is useful when the user is looking for a specific type of information from the source documents. For example, someone who is summarizing business letters may wish to find out all references by looking for words such as *Mr.* and *refer.* This ability to spot-emphasize certain words gives users tremendous control over the generated document summaries.

Chapter 4

Overview

The Generalized Document Summarizer is a complex program. Therefore, this chapter provides an overview before the later chapters delve into more detailed discussions. Since the GDS has its root in previous research and development work at Xerox, there is a historic and development overview to differentiate the present work from past influences. This will be followed by an overview of the usage and possible applications for the GDS. Lastly, the summarization process will be mapped out to provide a framework for later discussion.

4.1 History

Research in summarization technology and techniques first began at Xerox's Palo Alto Research Center (PARC) in the 1970's. It was part of a larger research effort in state-of-the-art linguistics tools and natural language processing. The PARC summarizer was written in Lisp and implemented all of the plaintext algorithms discussed in Chapter 3. After over a decade of research, the summarizer (along with the other tools developed by the PARC linguists) was handed over to Xerox's XSoft[1] division for commercialization. At this point, most of the code base was rewritten in C and C++. XSoft marketed the suite of linguistic tools under the name Xerox Linguistic Technology (XLT), and the summarizer was christened the XLT Summarizer.

[1]XSoft went independent in November of 1996 and is now called InXight, Inc.

Table 4.1: Summarizers from Xerox at a Glance

Name	Plaintext Input	HTML Input	Other Formats
XLT Summarizer 1.5	yes	no	no
XLT Summarizer 2.0	yes	yes	no
The GDS	yes	yes	yes

Later versions of the XLT Summarizer were incorporated into other Xerox products, including Visual Recall, a unique tool that provides document searching searching, summarization, and visualization capabilities.

While at Xerox XSoft in 1996, this author was involved with developing version 2.0 of the XLT Summarizer. On top of the plaintext summarization capability, HTML document parsing and summarization were added. Version 2.0 began shipping in late 1996, when the entire suite of tools was renamed LinguistX.

The GDS is based directly on the code base of LinguistX Summarizer 2.0. However, new customization capability for fine tuning the summarizer has been added. The tokenizer and summarizer now can read document formats other than plaintext and HTML—hence the *generalized* designation[2].

Table 4.1 lists the different summarizers from Xerox and compares their features. Note that version 2.0 of the XLT Summarizer is also known as the LinguistX Summarizer 2.0. For the purpose of this thesis, the term *XLT Summarizer* will refer to version 1.5, which can only handle plaintext source documents. Because the GDS and the XLT Summarizer 2.0 share much code in common, the latter will not be discussed separately.

[2]It should be noted that even though the GDS can summarize other types of documents, HTML remains an integral and essential part. Part of the design goal was to make sure that the GDS would perform well with HTML documents; this is important given the popularity of the WWW and the proliferation of HTML documents.

4.2 Development of the GDS

Most of the development work for the GDS was done over a seven-month period[3] at Xerox XSoft in Palo Alto, California. While the GDS was based on the XLT Summarizer, substantial additions and modifications have been made to arrive at the current GDS:

- New tokenizers were developed in order to parse documents with HTML or other formatting information. This involved defining a new tokenizer vocabulary (see Chapter 5) as well as rewriting tokenizers for faster performance.
- A number of heuristics were developed to summarize the new document types. Although some are general and can handle most source documents, many of the new heuristics aim at HTML document summarization in particular.
- A summarizer capable of keeping track of HTML and other formatting information was designed and implemented. This is the heart of the GDS and incorporated both heuristics developed under the XLT Summarizer and the new formatting-specific heuristics.
- Provisions were made to make the GDS extensible and highly user-configurable, allowing users to fine-tune the summaries generated by the GDS.

4.3 The LinguistX Software Suite

As mentioned earlier, the LinguistX summarizer is shipped as part of a larger linguistics tool package. This is not only because the tools are related in functionality, but also because the summarizer can take advantage of the other tools to generate optimal summaries. The GDS allows users to design "plug-in" tools to aid the summarization process. In the absence of other tools, the LinguistX tools are used as the default. See Section 4.5 for more details.

[3]From May, 1996 to December, 1996—as the fourth and final 6A Co-op assignment.

Below are descriptions of the other tools in the LinguistX software suite and how they can be used with the GDS:

- **Tokenizer**: tokenizers for several European languages and a few Asian languages are available for use with the summarizer to parse source documents. However, the summarizer has only been tested with English, French, and German plaintext and HTML documents. Although the GDS uses the LinguistX tokenizers as the default, other custom-made tokenizers can be used instead by providing a pointer to the proper procedures. However, any tokenizer used by the summarizer must have the same set of attribute vocabulary as the default LinguistX tokenizers. See Section 5.4 for the complete list of token attributes that must be supported.

- **Stemmer**: stemmers also exist for quite a few languages. These provide the root or stem form of any given word. For example, *swim* is returned when the word *swimming* is passed to the English stemmer. The stemmer can also work in reverse, providing various forms and conjugations from the root word. Stemmers can be used by the GDS to group related words together during word frequency analysis for more optimal summaries.

- **Tagger**: taggers are used to assign parts-of-speech information to words in each sentence. The GDS does not take advantage of this information currently; however, parts-of-speech information could give insights into the semantics of a document.

- **Noun-Phrase Detector**: noun-phrase detectors have been developed to extract common and proper noun phrases (*e.g.*, names, addresses, geographic locations). Again, this is not currently used by the GDS, but extensions could be made to use this information for better keyphrase generation (see Chapter 6).

- **Language Identifier**: LinguistX also has a language identifier that can determine the language of any given source document, assuming that the document is

in one of the supported languages (*i.e.*, English, French, German, Spanish, Portuguese, Dutch, Italian, and Japanese). The GDS can use this to automatically detect the language of source documents.

4.4 Usage and Applications

The GDS interface consists of a handful of application programming interface (API) procedure calls. This was done because the target users of the GDS are original equipment manufacturers (OEMs), not end-users. The OEMs could then fine-tune the GDS for the particular types of source documents to be summarized and ship it with a user-friendly and customized interface suitable for their particular applications. See appendix A for the API specification.

Although the API may be used in other programming languages, it is provided as a C header file for compatibility. The GDS was implemented in C++, because the object-oriented design allows one to quickly build variations of the stock summarizer (using class inheritance) for experimentation, evaluation, or production.

Besides the obvious application of summarizing documents, the GDS can be incorporated into web search engines for better, more relevant summaries. In addition, the API interface lends itself well to embedding summarization capability into applications. For example, email readers can call the GDS to display summaries for each email message, or a word processor can use GDS-generated summaries to search for documents matching certain user-specified criteria, (such as having the words "white house" in the summary).

4.5 The GDS Summarization Process

This section gives an overview of how summarization actually works in the GDS. Only the main ideas are presented here—the low-level details of the process can be found elsewhere in this thesis. (Chapters 5 and 6 discuss tokenization and summarization in great details.)

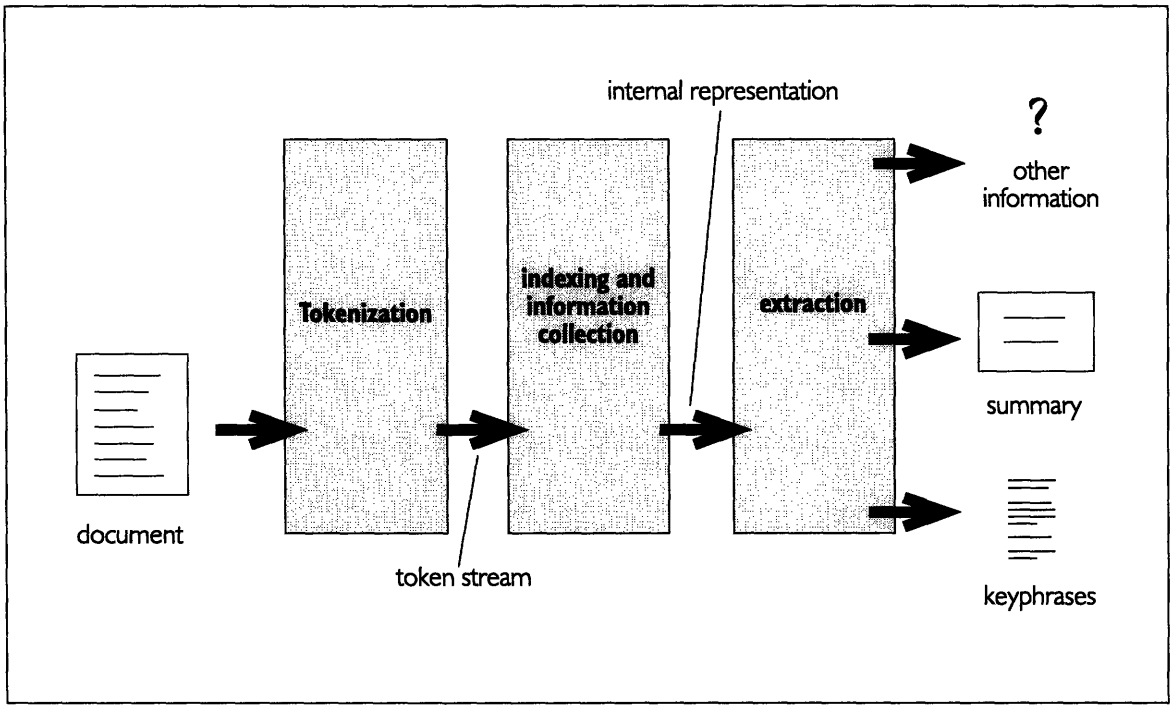

Figure 4-1: The Three Phases of Summarization in the GDS

As Figure 4-1 indicates, the summarization process can be divided into three distinct phases. In the tokenization phase, source documents are parsed and converted to a stream of tokens. In the indexing and collection phase, the tokens are examined. Information collected during this phase is placed in complex, internal data structures for later use. Finally, the extraction phase is where the summary sentences are generated, possibly along with other information about the source documents.

The tokenization phase is delegated to a tokenizer, which is a procedure that, given a buffer of text (in plaintext, HTML, or another recognized format), returns an array of tokens (the *token stream*) that correspond to the words and punctuation marks in the source buffer. The GDS is designed so that there is little interdependence between the tokenization phase and the other two phases. The rationale is that special-purpose tokenizers or tokenizers for other languages may be substituted for the default English tokenizer to accommodate document types not natively supported by the GDS.

In the indexing and collection phase, an (optional) stemmer is used to gather word frequency and other statistics. If the stemmer is absent, this phase will continue uninterrupted; however, the quality of the summary may suffer. This is because,

intuitively, stemming should improve the "accuracy" of the word counts. Without stemming, for instance, *come* and *came* would be considered two different words. Such schizophrenic classification would result in word counts of suboptimal accuracy and could adversely impact the quality of generated summaries.

After the collection phase, the extraction phase assigns each sentence a score based on the existence of pre-determined "features" in each sentence. Features are sentence characteristics that could provide hints as to whether a sentence would be a good summary sentence. Sentences with high scores are presumed to be indicative of a document's main ideas and returned as part of the summary. Presently, a dozen or so heuristics involving these features are implemented in the GDS. Then the the GDS extracts a number of sentences from each source document as the summary. In addition, the GDS tries to find one title sentence for the document separately from extracting the actual summary. Other information, such as a list of keyphrases, may also be available from the internal data structures at this point.

This three-phase process is repeated for each source document, with the performance roughly linear in the size of the source document.

The next two chapters will provide more details on tokenization, information collection, and the extraction of summary sentences.

Chapter 5

Tokenizing the Document

As pointed out in Section 4.5, the first of the three summarization phases is the tokenization phase. While the tokenizer can be seen as an entity separate from the rest of the GDS, a good tokenizer is essential to optimal summaries. This chapter describes the work done on the tokenizer, the tokenizer's role in summarization, and the token vocabulary used by both the tokenizer and the summarizer.

5.1 Development and Design Emphasis

As part of implementing the GDS, the tokenizer used by the XLT summarizer was completely rewritten. The rewrite was executed with two goals in mind: faster tokenization and a smaller token vocabulary. A smaller token vocabulary is desirable because it can reduce the overall complexity of the summarizer and ease the development of additional tokenizers.

The GDS relies on its tokenizer in many ways. If the tokenizer does not parse the source document correctly, it would be impossible for the GDS to generate quality summaries. In a way, the tokenizer acts as a translator for the rest of the summarizer—it translates source documents from a natural (human) language into a form that can be understood by the GDS.

The tokenizer is language-dependent, for it needs language details for tokenization. However, in an effort to keep the rest of the summarizer general and useful across dif-

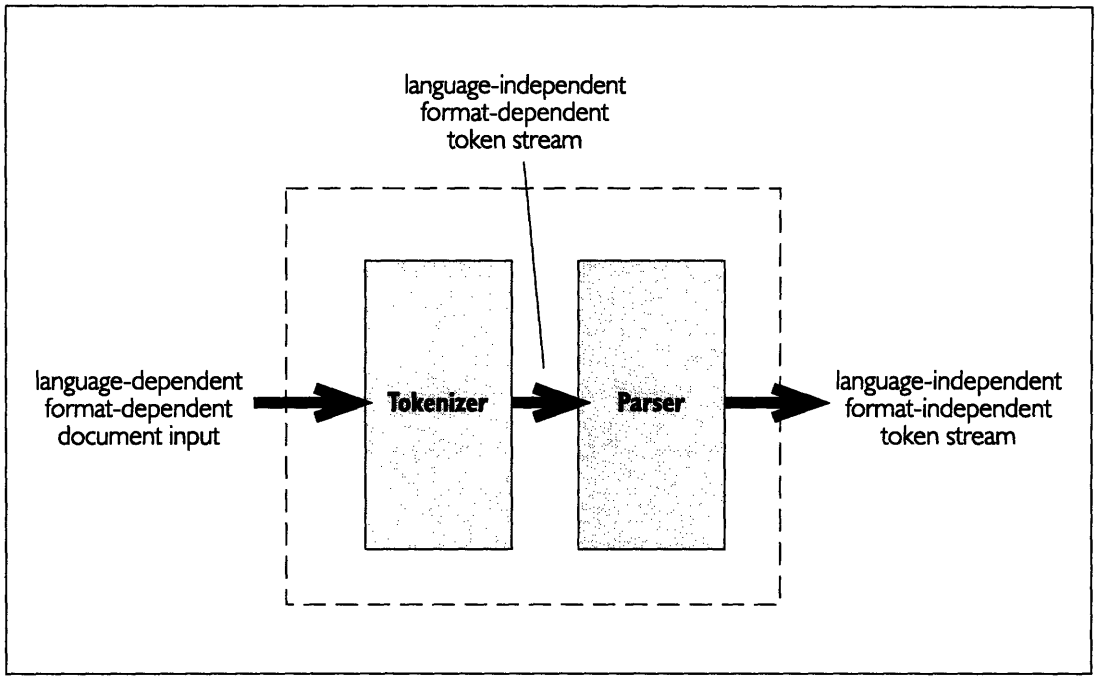

Figure 5-1: The Tokenizer and the Parser in the Tokenization Phase

ferent human languages and input file formats, all language-dependent operations are done in the tokenization phase using two modules: a callback-function tokenizer and a customizable parser. The GDS achieves its language independence (*e.g.*, English, French) through the tokenizer and its format independence (*e.g.*, HTML, LaTeX) through the parser.

As indicated in appendix A, the tokenizer is declared as a callback function in the GDS. The main reason for this architecture is to allow different tokenizers for foreign languages and/or special-purpose document tokenization. The parser, on the other hand, is customized through a file that the GDS reads during initialization.

5.2 The Tokenization Process

The tokenizer is responsible for breaking the input characters into individual chunks called *tokens*. Tokens are grammatical units specific to each language. (Typically, they are words and punctuation marks[1].) As the tokenizer breaks the input docu-

[1]In the GDS, tokens are the smallest unit used in the summarization process. Furthermore, each token is assumed to contribute equally to the contents/meaning of the document—with the

ment into tokens, it associates zero or more attributes with each token. The latter summarization phases use these attributes to determine how each token can be used.

The LinguistX tokenizer is more than just a simple tokenizer, however—it also performs limited lexical analysis on the source text. In contraction expansion, for example, *don't* is broken into two separate tokens corresponding to *do* and *not*. Lexical information is also used to resolve ambiguities in the source text. Whenever the tokenizer encounters a string of characters that it does not know how to parse, the text is passed to a finite-state transducer (FST). The FST, containing grammatical rules for tokenization, then determines the correct interpretation of the text in question. Since these rules are language-dependent, a FST (and a tokenizer) must be created for each language that the GDS supports. The FSTs, their creation, and their usage are issues beyond the scope of this thesis. For the GDS tokenizer, the transducers were created independently by a group of linguists of the code development.

The stream of tokens is then passed to the parser, which is implemented as part of the GDS and not a callback function. The parser is responsible for further processing of the tokens. Formatting tags that are specific to a particular file or document format must be parsed into general tokens that the summarizer can use. In this manner, the summarizer is completely language and format independent. These tokens are defined in Section 5.5.

In the absence of a user-specified tokenizer, the GDS uses the LinguistX English tokenizer as the default. Although the LinguistX tool suite provides tokenizers for quite a few languages, the GDS has only been tested and evaluated for English, German, and French[2].

---

exception of common words like "the" that contribute nothing. In languages like German, however, extra content can be added to a word by the usage of compound words. In such a manner, a token's contribution to the document content can arbitrarily inflate as new words are tacked on to the compound word.

[2]Although I have taken three years of German, I confess to knowing nothing whatsoever about the French language. The German and French versions of the GDS were actually tested by cool, foreign-language-speaking people at InXight, Inc.

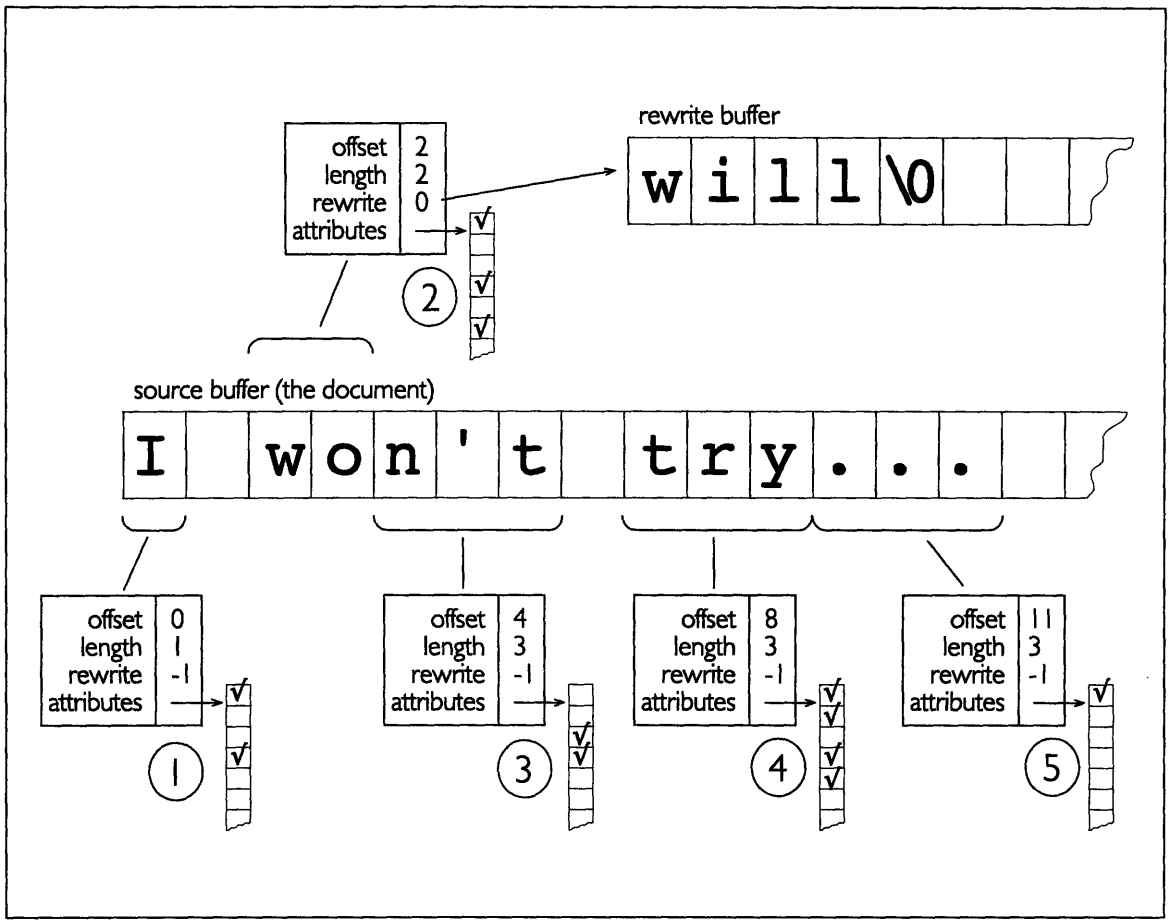

Figure 5-2: Five Sample Tokens

5.3 Keeping Track of Tokens

Since many tokens are processed during both the tokenization and the summarization phases, tokens are designed to be small and easily copied, allocated, and deallocated. The GDS defines a token structure as:

```
typedef struct
{
    int offset;
    int length;
    int rewrite;
    short attributes;
};
```

- `offset` is the offset (in bytes) to the actual text of the token in the source buffer.

- `length` is the length (in bytes) of the token's actual text. This always describes the original text, not the rewritten text (if one exists).
- `rewrite` is the offset (in bytes) to the rewritten text of the token in the rewrite buffer. A value of -1 indicates the the token was not rewritten.
- `attributes` is a bit-vector of token attributes, initialized to 0 upon creation.

The usage of the tokenizer is simple: a buffer of source text is passed to the tokenizer along with an empty *token buffer* and an empty *rewrite buffer*. Upon completion, the token buffer will contain the tokens corresponding to the source buffer. The source document may be split up into smaller chunks to tokenize, since the tokenizer does not need to see very much to know how to parse each sentence.

If any word in the source buffer was rewritten, its corresponding token will have an offset that points into the rewrite buffer so the rewritten text can be retrieved[3]. If either the token buffer or the rewrite buffer becomes full before the source buffer is exhausted, the tokenizer returns the buffers and indicates where it stopped so that tokenization can resume later with another call to the tokenizer with fresh buffers.

In most cases, the tokenizer can determine a token's attributes by an examination of the characters seen while traversing the source buffer. For example, if a word contains nothing but alphabetical characters, it is classified as an alphanumeric token. (In plain English: this is a regular word.) In the event that the FST was invoked in parsing the word, the FST will determine the attributes of the token.

The complete set of attributes used by the tokenizer to interface with the parser is the tokenizer's *token vocabulary*. Because of the plug-in tokenizer architecture, a standardized tokenizer vocabulary is necessary to guarantee that the rest of the summarizer will know how to interpret the tokens generated by the (possibly "foreign") tokenizer. The tokenizer's vocabulary is the interface between the summarizer and

[3]When the tokenizer encounters a word or a phrase that needs to be split up into more than one token *and* the resulting tokens will be ambiguous, it rewrites the word so the original meaning is not lost. The word *didn't*, for example, is split up into two tokens of *did* and *n't*. It is perfectly clear that *n't* is the contracted form of *not*. However, the word *won't* cannot be split up into *wo* and *n't* without problem, because *wo* could be either the exclamation "wo!" or it could be *will* if desired. The latter, of course, is the correct interpretation of *won't*.

Table 5.1: Attributes in the Tokenizer Vocabulary

Attribute	Examples
Alphanumeric	"this", "interlocutor"
Punctuation	commas, periods
Pre-Punctuation	single and double quotes
Post-Punctuation	single and double quotes, question marks
Sentence Punctuation	periods, exclamation marks
Tag	<HTML>, <APPLET>
Whitespace	carriage-returns

the tokenizer. Since the GDS can be used with a number of different tokenizers, the standard tokenizer vocabulary is "published" in a C header file. Every tokenizer to be used with the GDS must return tokens using this published set of attributes.

5.4 The Tokenizer → Parser Vocabulary

The tokenizer vocabulary currently consists of eight different attributes. Table 5.1 lists the different attributes and gives examples for each. Note that the examples that are not exhaustive. Any token can have more than one attributes. Most of the attributes are independent of one another—hence the implementation of the token attributes field as a bit-vector. If the tokenizer fails to observe any attribute in the current token, it leaves the attributes field alone. Tokens with no attributes will later be examined by the parser in an attempt to give them some attributes.

The tokenizer vocabulary is designed to be largely independent of any formatting in the source documents. Formatting tokens such as end-of-paragraph (EOP) marks or HTML tags can be accommodated by this vocabulary without any modification.

Below, each of the tokenizer attributes is explained in greater detail.

- **Alphanumeric**: The majority of the tokens returned by the tokenizer will have the alphanumeric attribute. Intuitively, a token is alphanumeric if it contains only letters and numbers. However, in the GDS, a token is alphanumeric if it should be considered a "regular word." The LinguistX tokenizer will only give

this attribute to tokens that contain either letters *or* numbers (but not both). Tokens that are a mixture of letters and numbers will not be recognized by the tokenizer as alphanumeric. This is done so that the parser can examine these tokens in detail to guarantee that tokens like "3.14," "A&W," and "$1,560,000" are all given this attribute ultimately.

- **Punctuation**: The token for any punctuation will have the punctuation attribute. Although punctuation marks are usually single characters, multiple-character punctuation marks such as the ellipse and the *em*-dash are included in this set. Basically, all punctuation defined in the ISO 8859-1 character set will have this attribute. Punctuation tokens usually also have at least one of the other punctuation attributes (pre-punctuation, post-punctuation, and/or sentence-punctuation).

- **Pre-Punctuation**: Pre-punctuation tokens are punctuation marks that can precede words without any whitespace in between. Examples include the single and double quotes (the opening quotes), as well as the left parentheses. Of all of the punctuation tokens, only pre-punctuation tokens can start a sentence.

- **Post-Punctuation**: Post-punctuation tokens are punctuation marks that can follow alphanumeric tokens without any whitespace in between. These include commas, periods, apostrophes, single and double quotes, and the right parentheses.

- **Sentence-Punctuation**: Sentence-Punctuation tokens are punctuation marks that can signal the termination of a sentence, such as periods, question marks, and even ellipses. Note that sentence punctuation tokens must also be post-punctuation.

- **Tag**: Tokens with the tag attribute are tokens used to convey formatting information. Typically, these tokens do not belong in the actual document context and are just auxiliary information.

- **Whitespace**: The whitespace attribute is given to tokens that indicate the end of a line (newline, carriage return, etc.), the end of a paragraph (EOP), or the end of a document (EOF). These are usually control characters that have special meanings in ISO 8859-1. Note that although the space character is used to find the start and the end of each word, it is not part of any token. Therefore, a string of spaces does not constitute a token.

In previous versions leading up to the GDS, HTML tag recognition was hardwired into the tokenizer. In those versions, two extra attributes were supported: the `html_tag` and the `html_junk` tags. The `html_tag` attribute is used to indicate HTML tags that the summarizer should examine and use. The `html_junk` attribute is used to mark chunks of HTML code that are intrinsically useless to the summarizer. For example, text between `<APPLET>` and `</APPLET>` tags are used to specify Java applets and thus contain no useful information. The `<IMG>` tag used to specify images is likewise unparseable by either the tokenizer or the summarizer.

Although hardwired HTML recognition has since been removed and replaced with the more general "tag" attribute, some special provisions remain in the tokenizer to handle special HTML tags (such as the `<!-- -->` comment tag) that do not follow the general HTML tag construction syntax.

5.5 The Parser → Summarizer Vocabulary

Although the parser must understand the tokenizer's vocabulary, it is not limited to just the attributes defined by the tokenizer. In general, the parser will recognize a broader set of token attributes. As mentioned before, these attributes must be general so that the summarizer can understand a variety of formatting and other information without knowing anything about the details of tokenization or parsing.

After the tokenizer has tokenized a buffer of text, the resulting tokens are returned to the parser. The parser does two things for each token:

1. Examine and modify the token's attributes as necessary. In particular, formatting tags such as HTML tags must be generalized.

Table 5.2: The Summarizer Vocabulary

Attribute	Examples
Alphanumeric	"this", "interlocutor"
Number	$45, 3.14
Punctuation	commas, periods
Pre-Punctuation	single and double quotes
Post-Punctuation	single and double quotes, question marks
Sentence Punctuation	periods, exclamation marks
Junktag	<APPLET>, <IMG>
Boldface-Tag	<B>
Italics-Tag	<EM>
List-Tag/Demote-Importance	<OL>
Heading-Tag	<H1>
Link-Tag/Special	<A>
User1-Tag	(could be anything)
User2-Tag	(could be anything)
Whitespace	<P>, <HR>
Ignore	(could be anything)

2. Record various information about each token for use during the sentence extraction phase..

The second step is addressed in Chapter 6, while this section discusses the summarizer's expanded token attribute set and the modification of token attributes.

Table 5.2 lists the complete summarizer vocabulary. This is, of course, a superset of the tokenizer vocabulary listed in Table 5.1. Note that the parser is responsible for interpreting tag tokens into the attribute set of the summarizer. The alphanumeric, punctuation, pre-punctuation, post-punctuation, sentence-punctuation, and whitespace attributes all retain their meaning as explained in Section 5.4. The meanings of the remaining attributes are:

- **Number**: Ideally, this attribute should be used to label all alphanumeric tokens that are numbers, including "25.01," "60," and "$1," so that they can be processed as numbers. In practice, it would be very expensive to examine every alphanumeric token just to find out if it is also a number—especially considering the relatively rare occurrence of numbers in documents. Consequently, only

tokens with no attributes are examined to see if they are numbers. This means that all decimal numbers, numbers with commas, and any other number with a punctuation mark in it will be marked as numbers, while the plain "60" remains alphanumeric only. Since numbers *are* also alphanumeric, this is a trivial issue.

- **Junktag**: The parser recognizes useless formatting tags and gives them this attribute. For example, `<IMG>`, `<APPLET>`, and `</APPLET>` tags will all become junk tags in the summarizer vocabulary. In general, junktag tokens "behave" in two ways: (1) the tag itself is useless, or (2) the tags along with the text in between are useless. `<IMG>` is of the former type, while `<APPLET>` and its corresponding `</APPLET>` tags are of the latter type. The parser is given a list of formatting tags that should be junked, along with detailed information on which type of junktag each is. This attribute is mutually exclusive with the -Tag attributes.

- **Boldface-Tag, Italics-Tag, List-Tag, Heading-Tag, Link-Tag**: These general and generic tags convey formatting information from the parser to the summarizer. They correspond to boldface, italics, lists (such as `<OL>` in HTML or `itemize` in LaTeX), headings, and links (such as hypertext links). Note that the parser is free to map a token to any of these tags—it is completely up the user to specify how formatting information should be interpreted. For example, a user may choose to ignore all ordered lists while parsing HTML documents (*i.e.*, `<OL>`) but map all unordered lists (*i.e.*, `<UL>`) to List-Tag.

- **User1-Tag, User2-Tag**: These two generic tags are provided so that the user can define other categories of tags. These custom tags are coupled with probabilities as well, so they can be incorporated into the summarization process.

- **Ignore** tokens. This is a generic attribute that tells the summarizer to ignore this tag (*i.e.*, exclude it from further processing). This attribute is typically used to label tokens in between junktag tokens[4]. For instance, all tokens between a

[4]Note that the Ignore attribute overrides all other attributes that a token might have. Tokens

pair of <SCRIPT> and </SCRIPT> tags should be ignored.

in between junktags are (appropriately) meaningless. Since the tokenizer actually took the time to assign attributes to these meaningless tokens during tokenization, tokenizing certain documents (*e.g.*, ones with lots of junktags) can be inefficient. However, the tokenizer cannot avoid this inefficiency, because tokenization is done locally without any information about the document as a whole or what the formatting tags mean.

Chapter 6

Generating the Summary

After tokenization, the summarization proceeds into the information collection and summary extraction phases as outlined in Chapter 4. Because they are closely related and share many data structures, this chapter will discuss the design and implementation of both phases in the GDS.

After the source document has been converted into a stream of tokens (in the summarizer's token vocabulary), the GDS makes a single pass through the token stream, collects word frequency and other statistics, and ultimately scores and ranks the sentences. From there, a summary of n sentences and other information can be easily retrieved.

6.1 Collecting Document Information

The first step is to collect information on the source document. Since the GDS does not process the document text directly, it gathers the necessary information from the tokens and their attributes. The GDS records information on four levels: by word, by sentence, by paragraph, and by the global document environment.

6.1.1 Word Indexing

For each alphanumeric or number token that it encounters, the GDS checks to see if the token's text is found in a user-supplied list of common *stop-words*. This is done so that the document's word statistics will not be skewed by the presence of common words (*e.g.*, *the*, *this*) that do not have distinct meanings. If the word is a common word, it is dropped from further processing. Otherwise, it is entered into a word table with three pieces of information: (1) an unique word ID for the word, (2) the number of times the word has appeared in the document so far, and (3) the sentence ID of each sentence that contains the word.

The GDS API allows the user to provide a stemmer so that word statistics can be gathered using morphological stems instead of the actual words. This should provide a more accurate picture of word frequency and usage within a document. For example, occurrences of the words *swimming*, *swam*, and *swimmer* will all be counted under the word *swim*. One immediate advantage of using the stemmer is that similar words and phrases are less likely to compete with each other when the GDS is picking the frequently-used words.

6.1.2 Sentence Collection

On a higher level of organization, the GDS keeps phrase and sentence information. Phrases consist of two to five words, ending with either a punctuation or a word that can end a phrase[1]. These phrases are important, because the GDS uses them as indicators of the document's main themes.

Tokens are also collected into sentences whenever the GDS encounters a token that indicates the end of a sentence. This could come in the form of a sentence-punctuation token, a whitespace token, or just the abrupt end of the source document. After a sentence has ended, the next token will usually cause the creation of a new sentence data structure, complete with a new sentence ID, a new position ID (corresponding

[1] A list is shipped with the GDS outlining the *phrase stop-words*. Basically, it contains words that should not be part of word phrases.

to a sentence's position within a paragraph), a set of initial scores for the sentence, as well as fields to record the beginning and ending offsets for the sentence so the user can quickly retrieve the sentence text later.

6.1.3 Paragraph Creation

Finally, sentences are grouped into paragraphs, which are delimited by whitespace or EOP tokens. Each paragraph data structure records: (1) the number of sentences in the paragraph, (2) the sentence IDs of the first and last sentences in the paragraph, and (3) the position of the paragraph in relation to the rest of the document.

6.1.4 Maintaining Global Environmental Information

As the summarizer is reading the token stream, it needs to maintain state information so the formatting context of the document is not lost. The GDS uses a set of stacks to match the beginning and the ending tags. A stack is a natural data structure to use for handling matching tags that can be nested. However, since some formatting tags do not have strict matching requirement (*e.g.*, `<P>` does not have to be closed by a corresponding `</P>`), a strict push/pop protocol is not always observed. It is also possible for a tag to pop frames from stacks belong to other tags. For example, encountering a `</HTML>` will cause all HTML tag stacks to become empty immediately, for the HTML document has ended.

Note that using a single stack to keep track of all tags will not work. This is because the formatting information for any word is the aggregate of the formatting tags that are active at that point, not just the most recent formatting tag. Furthermore, it is not possible to handle out-of-order tag nesting (*e.g.*, `<A><B>blah</A></B>`) correctly with only one stack[2].

[2]Although the formal HTML language specification does not allow out-of-order tag nesting, most of the web browsers do. As a consequence, it would be unwise to have implemented only the strict specification. Instead, the GDS opted for a more tolerant approach to HTML documents.

6.2 Sentence Scoring

At the end of the collection phase, the token stream has been read and all the word, sentence, and paragraph structures have been set up. The GDS then enters the third and final phase in which the sentences are scored using a combination of the heuristics described in Chapter 3. The summarizer must achieve three things here: (1) it must find some number of *direct theme* keywords, (2) it must find a title sentence for the document, and (3) it must score the rest of the sentences in preparation for generating the document summary. This section describes how these tasks are completed.

6.2.1 Direct Theme Keyword Selection

Selecting the direct theme keywords (see Section 3.5.2) is a simple procedure. For each word in the word table, a score is computed with consideration to the length of the word as well as the frequency of the word in the document. Intuitively, longer words that appear more frequently in the document will have higher scores and are better direct theme keywords. These direct theme keywords are returned by the `xlt_extract_keyphrases()` API call. Currently, formatting information is not used to select the keywords.

6.2.2 Title Sentence Detection

Not all sentences are created equal—some sentences are more relevant than others. The most important sentence in the source document for the GDS to find is the title sentence. The title sentence may or may not be the title given or designated to the document, although it is almost always found near the beginning of a document. The title sentence is detected by giving a *title sentence score* to each of the candidate sentences. Scores are given based on: (1) the presence of uppercase words, (2) the presence of direct theme keywords, (3) the length of the sentence, (4) the presence of other sentences immediately before or after this sentence, (5) the location of the sentence in the document, and (6) other formatting information.

For plaintext documents, title sentence detection is limited to sentences that ap-

pear before the start-of-text. In HTML documents, however, title sentences are often within the <BODY> section of the document. The GDS has a customizable parameter to "dip" into the body section of HTML documents to look for title sentences if no suitable candidate exists in the header section.

6.2.3 Scoring Body Sentences

Unlike the title sentence, other sentences in the body of the source document are scored using a standard set of heuristics and probabilities discussed in Chapter 3. If the source document is plaintext only, only the five plaintext heuristics are applied. For HTML and other formatted documents, the other heuristics are figured into the sentence score.

The summarizer computes two scores for each sentence. The main score is computed as the product of probabilities based on the model described in Section 3.3.1. This score reflects the result of all of the applied heuristics. The second, auxiliary score is a "no-paragraph" score that does not apply the sentence-location heuristic of Section 3.5.3. (It is called the "no-paragraph" score because it ignores the position of sentences within each paragraph.)

6.3 Summary Extraction

After all sentences have been scored, generating the document summary is simply a matter of finding the sentences with the highest scores. For summaries with fewer than five sentences, the GDS uses only the main score to rank the sentences. For summaries longer than five sentences, the GDS returns the five sentences with the highest main scores, and then finds the other summary sentences by looking at only the auxiliary "no-paragraph" score.

The rationale behind this approach is that if the user wants a short summary, the GDS wants to return sentences that provide the main ideas in a document, thus sentence position within each paragraph plays an important factor. For longer summaries, however, the user is probably looking for more details. In such summaries,

sentences in the middle of a paragraph might be good, because they typically provide more details than the first and the last sentences in each paragraph.

Before returning the summary sentences, the GDS sorts them in the order they appear in the document, since each sentence is tagged with its main score, the user can view the summary in order of the scores or in document order.

Other aspects of the document can also be queried at this point. The document's title sentence, direct theme keyphrases, and word statistics can all be returned to the user if such information is considered interesting.

Chapter 7

Evaluation

A summarizer is only as useful as the document summaries that it generates. Unlike other types of software, however, there is no straight-forward, quantitative method for measuring the quality of a document summary. This is mainly because there is no optimal model for human learning—everyone seems to absorb and approach information in a different manner. This chapter describes four studies that provide a practical evaluation of the GDS in light of the aforementioned difficulties. In general, the GDS appeared to have improved upon its predecessor while expanding the number of document formats that can be recognized.

In order to evaluate the GDS, the four surveys focused on two desirable qualities: (1) high relevance assessibility, and (2) comparatively high-quality summary sentences. As mentioned in Section 3.1, high relevance assessibility means that a summary is highly indicative of the content of the actual document. In the most idealistic scenario, the summary would contain everything one needs to know about the source document, and reading the original document would be unnecessary. "High-quality summary sentences" is a subjective measure determined by the human subjects in these surveys.

By using relative comparisons between the GDS' summaries and summaries generated via alternative methods, one can rely on relative measure of quality instead of trying to establish some absolute measure. Furthermore, concentrating on the GDS' performance in these two dimensions provides focused insights into possible

improvements and modifications in the future.

7.1 GDS vs. Human-Generated Summaries

In the first study, document summaries generated by the GDS were compared with summaries generated by human readers. The survey corpus consisted of 60 HTML documents collected using the world-wide web. The search engine[1] used was asked to return random documents in order to make sure that the corpus was not limited to a narrow range of topics.

An additional 100 HTML documents were collected at random to provide some observations into HTML trends:

- Only a handful of the webpages were over 2 pages in length. Most were between 1 and 2 pages (excluding graphics).
- Most webpages had 10 or more graphic images. Over 50% had a "title" or title banner that was graphics-only (and thus not readable by the GDS).
- Physical HTML tags (*e.g.*, `<B>`, `<I>`) were used 2 times to 1 over logical HTML tags (*e.g.*, `<STRONG>`, `<CITE>`).

A in-depth survey could give insights into how to create other heuristics for summarizing web documents. (Such a survey was done at the beginning of the GDS project to determine useful heuristics for summarizing HTML documents.)

The corpus was distributed to a number of undergraduate and graduate students. Each document was printed using the Netscape browser so that the human readers did not have to read HTML tags. Also, images were removed from the documents so the subjects did not have more information than the GDS while summarizing the documents.

The subjects were then instructed to select for each webpage: (1) a title sentence for the entire document, and (2) ten sentences good for a summary of the document.

[1]The Webcrawler search engine was used in collecting the corpus for this survey. See `http://www.webcrawler.com`.

Table 7.1: Percentage of GDS Summaries Matching Human-Generated Summaries

Title Sentences	Ten-Sentence Summaries	Five-Sentence Summaries
77.6%	61.0%	48.7%

For short or ill-formed web pages, the summary could be shorter and the title sentence did not have to be present.

The same 60 HTML documents were processed by the GDS. The summarizer was instructed to produce a title sentence and ten summary sentences for each document. The results are summarized in Table 7.1. With all heuristics turned on and at the default settings, the GDS' title sentences matched 77.6% of the those selected by the subjects (38 out of 49). In the actual summaries, the numbers were not as high. Of the summary sentences returned by the GDS, 61.0% of them were in the human-generated summaries (180 out of 295).

These numbers improved upon the results obtained using the XLT Summarizer [5] on plaintext technical papers. However, note that asking the GDS to generate ten summary sentences per document gave it an advantage, since the average length of the human-generated summaries was less than ten sentences. When using five-sentence summaries from the GDS, the percentage dropped to 48.7% (112 out of 230). Nevertheless, the GDS performed well when the generated summaries were about ten sentences long. This indicates that a better summary can be generated by asking the GDS for more sentences. Of course, this needs to be balanced with one's willingness to read through all of the summary sentences.

Another interesting point can be observed by plotting the sizes of the source documents vs. the "accuracy" (percentage) of the GDS' summaries. As shown in Figure 7-1, there is no noticeable correlation between document size and the accuracy of the GDS. (the percentages correspond to the five-sentence summaries, not the ten-sentence summaries.) Intuitively, it seems that the GDS should do better on small documents because it generated summaries of a fixed size. Assuming that the human-generated summaries are the ideal summaries, this figure shows that other factors

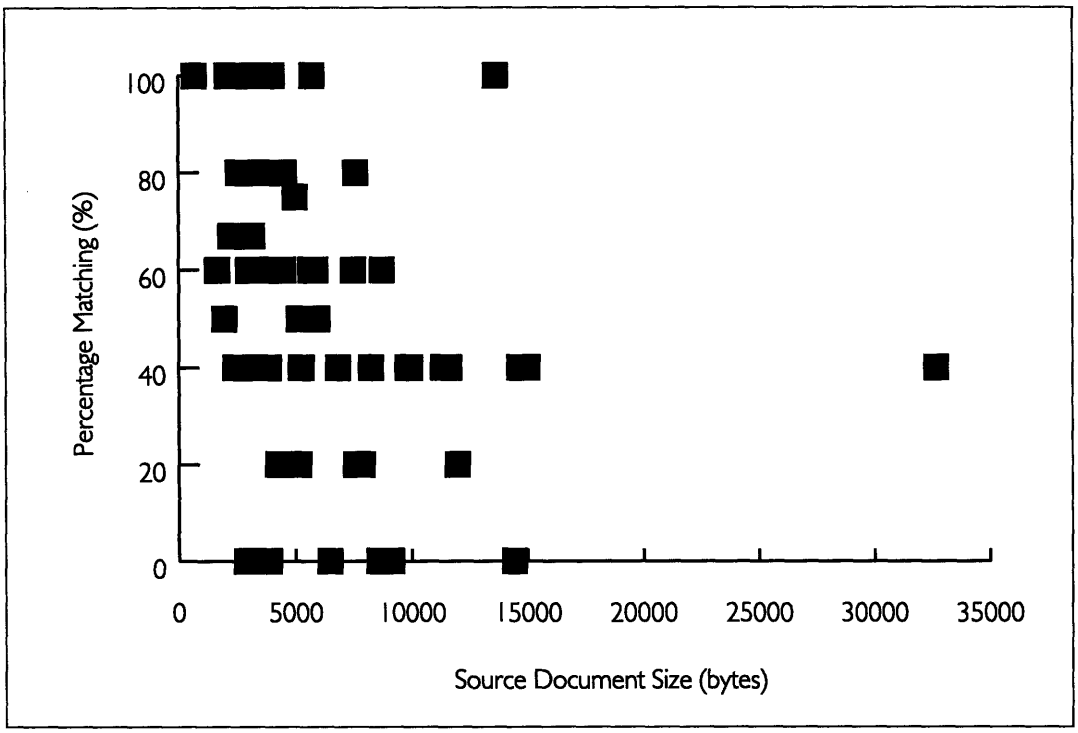

Figure 7-1: Document Size vs. Matching Percentage with Five-Sentence Summaries

could significantly impact how good the GDS summaries are. In particular, unusable HTML elements such as images, forms, or strangely-formatted HTML source can all cause the summarizer to stumble. In such cases, generating larger summaries can absorb the mistakes.

7.2 GDS vs. Simple Summaries

In the second study, subjects were asked to compare the relative quality of pairs of document summaries. This was purely a subjective preference.

The subjects were each given 28 pairs of summaries on various subjects. Each pair was constructed in the following manner. Using another world-wide web search engine[2], a single-word query was submitted, and the first two non-identical webpages returned by the search engine were used. For each document, the first of the two summaries was obtained directly from the search engine's description of the page. In

[2]This second survey made use of the Infoseek search engine at `http://www.infoseek.com`

Table 7.2: Preference of GDS Summaries over Alternative Summaries

	Number	Percentage of total
Preferred GDS Summary	117	69.6%
Preferred Alternative Summary	51	30.4%
Total	168	100%

this case, Infoseek used the first 100 or so characters as the summary. The second summary was a two-sentence summary generated by the GDS.

The subjects were instructed to select one out of each pair that they "prefer as the summary of a webpage with the given title sentence." Of the six subjects and the 168 summary pairs, the GDS' summary was preferred 117 times. In other words, the subjects preferred the GDS summaries 2.3 to 1 over the other summaries. (See Table 7.2.)

The result of this survey was not overwhelming positive. Although the GDS summaries were preferred most of the time, the subjects liked the alternative roughly one out of three times. One reason for this performance, as mentioned in the previous study, is that the GDS is not optimal when generating very short summaries. This is a weakness that should be addressed in a future version.

7.3 Extracting Details from Documents

In the third survey, subjects were asked to answer eight simple questions using the information in a set of given documents.

The corpus used in this experiment was a small set of long-ish documents (1 to 2 pages). For each webpage, one to two questions were formulated based on the contents of the full document. Each subject then gets the document in one of three forms:

- Full-length: the entire document.
- First-Paragraph: only the title sentence and the first paragraph. (In documents where the first paragraph was very short, the second paragraph was also

Table 7.3: Extracting Document Details Using a Variety of Methods

Question	Full Document Time	First Paragraph Time	Summary Time
1	30 s	30 s	30 s
2	4 min	1 min	30 s
3	30 s	n/a	30 s
4	30 s	n/a	30 s
5	2 min	n/a	45 s
6	1 min	n/a	45 s
7	2 min	30 sec	30 s
8	2 min	1.5 min	n/a

included.)

- Summary: a ten-sentence summary generated by the GDS along with a list of keyphrases.

Each subject answered the questions and recorded the amount of time spent researching the answer (*i.e.*, by reading the given documents). For this survey, the full-length and summary formats were expected to be more comprehensive and useful than the first-paragraph format, while the first-paragraph and summary formats were expected to provide the answers in less time.

The average time taken by the subjects to answer each question is listed in Table 7.3. The subjects were instructed to round the time to multiples of 30 seconds. The entries with "n/a" indicates that the question was not answered (or not answerable) using that particular format. As expected, all answers were found using the full document, although it is also the slowest methods in most cases. The first-paragraph format was fairly fast in most cases, although it could not provide the answers to half of the questions. However, the summary approach also had trouble extracting details for some of the questions.

7.4 Picking Relevant Documents

In the last survey, the GDS summaries were tested to see if they would be good indicators of relevance in topical searches. This was done by presenting the subjects with a number of web query results. In one group, the subjects were given query results generated directly by Alta Vista[3], while another group was given query results with the GDS summaries. The subjects were then asked to rank the order in which they would traverse the query results if they had to answer some general question on the particular topic.

Designing this particular survey was difficult for a couple of reasons:

1. **How to determine which documents were relevant?** There is obviously more than one way to determine relevance, all of which are subjective. In this survey, the ten webpages from each query result were classified as "highly relevant," "somewhat relevant," or "not relevant" based on their actual contents.

2. **How to score each subject's ranking for the documents?** Assessment of the subjects' ranking was done by considering the ranking given to the "highly relevant" webpages. By averaging each document's ranking over all test subjects, one can get a sense of each group's consensus as to how important that webpage is. A lower average ranking would indicate a document considered more relevant by the subjects.

Figure 7.4 displays the results of this survey. For each of the seven query results, the average ranking of relevant documents for both types of summaries are shown. (Lower is better, with 1 being the lowest number possible.)

While the figure shows that the GDS summaries help readers assess relevance a little better than the other approach, this survey is is somewhat inconclusive. There is too little difference between the rankings and the composite scoring to seriously differentiate between the two approaches. At the very least, the summarizer did not fare worse than the web search engine.

[3] `http://www.altavista.digital.com`

Table 7.4: Average Ranking of Relevant Documents

Query	Web-Engine Style	GDS Summary Style	Improvement
1	3.3	2.7	-0.6
2	4.6	4.0	-0.6
3	5.0	2.7	-2.3
4	1.0	1.0	0
5	3.3	3.0	-0.3
6	2.3	2.0	-0.3
7	4.3	3.7	-0.6

Chapter 8

Conclusion

The GDS has demonstrated that the word-statistics approach to document summarization continues to produce effective summaries when expanded to include formatting information. There is still much room for improvement, however. This chapter discusses some of the strengths and weaknesses of the GDS and concludes with some suggestions for future research in summarization techniques.

8.1 Strengths of the GDS

Because of its root in plaintext summarization, the GDS excels in summarizing text. The formatting information provided by HTML can be useful in summarization, but the GDS still performs better on documents with more text than HTML tags.

Having said that, one should point out that the GDS actually keeps track of more information than it is using. The internal data structure and organization is capable of holding (and does hold) minute details about the document (HTML or otherwise). Some of this remain unused because currently there are no appropriate heuristics to utilize them.

One key strength of the GDS is title sentence detection. In both plaintext and HTML documents, the GDS is able to pick out relevant title sentences consistently. For the actual document summary, its performance increases dramatically as the summary size increases. In particular, manageably-sized ten-sentence summaries seem to

be the best compromise between reading a short summary and having a comprehensive, relevant summary.

In addition, the user is given the flexibility and control to bias the GDS-generated summaries. By allowing arbitrary word and sentence emphasis, the Generalized Document Summarizer can be fine-tuned into a specialized, limited-domain summarizer that excels at summarizing a particular type of document.

8.2 Summarization Pitfalls

The main weakness of the GDS lies in its inability to handle non-textual elements found on many webpages. Elements such as graphics can be very useful, especially when words are contained in these images. HTML tables can also cause problems. The GDS actually reads the words within each cell of a table, but the information contained therein often cannot be treated as regular sentences. For example, a feature-comparison table on the Nikon F5 and Canon EOS 1N cameras probably has very little to offer in terms of textual, extractable information. Other times, however, the HTML table tag is used to construct a two-column look. In these cases, the text within the table cells are very relevant.

The GDS also has problems processing words or sentences that are not in the normal document layout. For example, tables of contents, indices, and itemized lists can all be problematic. While human readers can adjust to radically different document layouts without much effort, such layout recognition and processing must be built directly into the summarizer for it to work. As another example, headers or footers that repeat on every page of a document can skew the word statistics analysis; Worse—it can break up sentences that cross page boundaries.

8.3 Future Work

As discussed above, the GDS and summarization technology as a whole have much work ahead. With the growing societal and scientific usage of computers and digital

media—and the ensuing explosion of information—summarization must move away from traditional text-only processing to incorporate and make use of multimedia information.

Already research is being done in the area of graphic image and audio information summarization [1]. This will become more important as HTML documents move into the HTML 3.2 standard, with its sophisticated multimedia capabilities and display control. While plaintext cannot be replaced entirely, making use of multimedia elements should improve the quality of the summary.

Future summarizers might also want to include the ability to handle multiple layouts. Optical character recognition (OCR) technology is fairly advanced today, and the summarization process can benefit from a similar ability to handle complex layouts.

The HTML markup language itself presents many opportunity for improvements in summarization technology. Summaries can be extended to summarize hypertext-linked documents. Another possibility is to allow the user to extract arbitrary elements from HTML documents. For example, one might want to see all hypertext links, all tables, or all `<META>` tags in a document. While this information is not strictly a summary, it can provide a different dimension of insight.

Appendix A

The GDS API

The Generalized Document Summarizer is available from InXight, Inc. on Solaris, Win32, MacOS, and BeOS. Wherever possible, it is shipped as a dynamically-linked library (DLL). On platforms that do not support DLLs, a static library is provided.

The application programming interface (API) [12] contains five C-style procedure calls as described below. A C header file is included with the GDS to declare these function prototypes, the data structures, and the GDS constants.

A.1 Procedure Calls

In order to perform document summarization, the user needs to first create one or more *summarizers* (of type `xlt_summarizer`). A valid summarizer object can be asked to summarize a document by providing it with a callback function that knows how to read from the source document. (`xlt_summarize()` is the function to call to do the actual summarization.) After summarization is complete, use `xlt_extract_sents()` to extract a summary and `xlt_extract_keyphrases()` to extract a list of keyphrases.

Figure A-1 shows fragments of a program that calls the GDS API calls to summarize a file. In this example, the file is opened using `fopen()`. However, C++ I/O (*e.g.*, iostream) is also supported.

A.1.1 xlt_make_summarizer()

```
xlt_summarizer
xlt_make_summarizer (
     xlt_stemming_function_type  stemming_function,
     void*                       stemmer_object,
     const char*                 language_file,
     const char*                 index_dropfile,
     const char*                 debug_file,
     int*                        error);
```

Creates a summarizer object from the given language configuration file. A method for stemming is also specified if stemming is desired. If object creation fails, `NULL` is returned, and `error` is stored with a negative error code (see Section A.4). Parameters

```
/* create a summarizer object */
xlt_summarizer sumHdl;
int status;
sumHdl = xlt_make_summarizer(stem_func,   /* (callback) stemming function */
                             stemHdl,     /* the stemmer object */
                             wordFile,    /* name of summarizer wordfile */
                             dropFile,    /* name of dropword file */
                             debug,       /* debug string */
                             &status);    /* object creation status */

/* make sure creation was successful */
if (status != XLT_NO_ERROR)
{
   /* die */
}

/* open file and summarize it*/
FILE *fp = fopen(``source.html'', ``r'');
status = xlt_summarize(sumHdl,     /* the summarizer object */
                       tokHdl,     /* the tokenizer object */
                       read_func,  /* (callback) source reading function */
                       (void*)fp,  /* the file pointer for read_func */
                       sumOpts,    /* summarizer option bit-vector */
                       0);         /* not guessing the document size */

/* make sure creation was successful */
if (status != XLT_NO_ERROR)
{
   /* die */
}

/* open file and summarize it*/
FILE *fp = fopen(``source.html'', ``r'');
status = xlt_summarize(sumHdl,     /* the summarizer object */
                       tokHdl,     /* the tokenizer object */
                       read_func,  /* (callback) source reading function */
                       (void*)fp,  /* the file pointer for read_func */
                       sumOpts,    /* summarizer option bit-vector */
                       0);         /* not guessing the document size */

/* make sure summarization was successful */
if (status != XLT_NO_ERROR)
{
   /* die */
}

/* allocate structure for holding summary and keyphrases */
xlt_extract* sents = (xlt_extract*) malloc(numSents * sizeof(xlt_extract));
xlt_extract* phrase_buffer = (xlt_extract*) malloc(numPhrases * sizeof(xlt_extract));
char* buffer = (char*) malloc(numPhrases * MAXLEN * sizeof(char));

/* get the summary */
int gotSents = xlt_extract_sents(sumHdl,    /* the summarizer object */
                                 sents,     /* struct to hold summary */
                                 numSents); /* # of sentences desired */
if (gotSents > 0)
{
   /* show summary sentences */
}

/* get the keyphrases */
int gotkeys = xlt_extract_keyphrases(summHdl,

/* if we got some key phrases, display them. Otherwise, complain.
if (gotkeys > 0)
{
   /* show keyphrases */
}

/* delete summarizer object */
xlt_free_summarizer(sumHdl);
```

Figure A-1: A Simple Sample Driver to Summarize Documents

to `xlt_make_summarizer()` are as follows:

- `stemming_function` is a stemming function. See Section A.2 for details.
- `stemmer_object` is specific to the stemming function (the first parameter). There is no restriction on what this object can be. The summarizer merely passes it to the stemming function when stemming a word.
- `language_file` is a null-terminated string containing the path and filename of the supplied language configuration file for the language desired. These configuration files are provided, and end with the `.sum` extension. For example, the English language configuration file is `english.sum`[1].
- `index_dropfile` is a null-terminated string specifying a file of additional stop-words. A stop-word list already exists in the language configuration file, but for some applications additional words may be desired. This parameter is normally set to `NULL`.
- `debug_file` is a null-terminated string specifying a file in which to store a log of summarizer actions, useful for debugging. This parameter is usually set to `NULL`.

A.1.2 xlt_summarize()

```
int
xlt_summarize (
     xlt_summarizer              sumHdl,
     xlt_tokenizer               tokHdl,
     xlt_read_function_type      fSource,
     void*                       pSourceContext,
     int                         summarizeOptions,
     long                        docSize);
```

[1]This file contains GDS-specific information such as the list of stop-words. It can be modified to alter the behavior of the summarizer.

Reads in a document and summarizes it, allowing sentence extracts and key phrases to be retrieved using `xlt_extract_sents()` and `xlt_extract_keyphrases()`. The summarizer reads in the document chunk by chunk, by calling `fSource` on `pSourceContext`. The same summarizer and tokenizer objects may be used more than once to summarize multiple documents. The function returns `XLT_NO_ERROR` if successful, and a negative error code if unsuccessful.

Its parameters are:

- `sumHdl` is a valid summarizer object created by `xlt_make_summarizer()`.
- `tokHdl` is a valid tokenizer object that can tokenize the given source document and produce the tokenizer vocabulary described in Section 5.4.
- `fSource` is a callback function that reads the actual source document. See Section A.2 for more details.
- `pSourceContext` is passed on to the document-reading function (*i.e.*, `fSource`).
- `summarizeOptions` can be zero or more of the constants enumerated in Table A.1 `AND`'ed together.
- `docSize` is a hint of how large the source document is. If this parameter is set to 0, the GDS will not make any assumption about the size of the document. Since this number is only a hint, it does not have to be accurate.

A.1.3 xlt_extract_sents()

```
int
xlt_extract_sents (
     xlt_summarizer              sumHdl,
     xlt_extract*                sents,
     int                         wantSentences);
```

After summarization, this call retrieves sentence extracts from the given summarizer object. Character offsets into the original document are stored in `xlt_extract`

Table A.1: Possible Input Values for summarizeOptions

Option	Description
XLT_SUMMARIZE_HTML	Summarize an HTML document. If this option is not specified, the summarizer assumes that the source document is plaintext, not HTML.
XLT_SUMMARIZE_PAR_BREAK_ON_EOL	Interpret a single newline character as a paragraph break. If this option is not specified, the summarizer auto-detects whether paragraph breaks (*i.e.*, EOP markers) are one or two newline characters.
XLT_SUMMARIZE_PAR_BREAK_ON_2EOL	Interpret two consecutive newline characters as a paragraph break. The default is to auto-detect EOP marker length.

records in `sents`. The function returns the nonnegative number of extracts stored (may be 0).

- The parameter `wantSentences` specifies the maximum number of summary sentences that may be stored in `sents`.

Sentences are stored in the order in which they appear in the document, with the exception of the title sentence (determined by the summarizer). The title sentence is always the first entry in `sents`. If an error occurs, a negative error code is returned.

A.1.4 xlt_extract_keyphrases()

```
int
xlt_extract_keyphrases (
     xlt_summarizer              sumHdl,
     char*                       buffer,
     int                         bufferSize,
     xlt_extract*                phraseOffsets,
     int                         wantPhrases);
```

After summarization, this call retrieves key phrase data from the given summarizer object. Keyphrase text is stored in `buffer`, and the array of extracts `phraseOffsets` is stored with offsets to the phrases. The function returns the number of phrases stored (may be 0). Note that a successful `xlt_summarize()` call must have been done on the summarizer object before this function can be used successfully.

- The parameter `sumHdl` is the summarizer object from which to retrieve the keyphrase data.
- The parameter `bufferSize` specifies the number of characters that `buffer` can hold.
- The parameter `phraseOffsets` is an array of `xlt_extract` records, each of which specifies the offset (into `buffer`) and length of a phrase. The phrases are not assigned scores but are sorted according to relevance. The most relevant phrase is written first.
- The parameter `wantPhrases` specifies the maximum number of phrases that can be stored in `phraseOffsets`.

A negative error code is returned if the operation failed.

A.1.5 xlt_free_summarizer()

```
void
xlt_free_summarizer (
     xlt_summarizer              sumHdl);
```

Deallocates resources associated with the given valid summarizer object.

A.2 User-Defined Callback Functions

The GDS API defines two callback function types. This allows the user to have ultimate control over the stemming and the document-access functionality. The first

type is `xlt_stemming_function_type`, used by the summarizer to call a user-specified stemmer. (See the `xlt_make_summarizer()` call.) This is defined as:

```
int
function (
     const char*                word,
     char*                      buffer,
     int                        buffer_size,
     void*                      stemmer);
```

Where `word` is the null-terminated string to be stemmed, `buffer` holds the result of the stemming, and `stemmer` is the object that the summarizer was given to pass to the callback function. The function returns 0 on success.

The second callback function is for tokenizing the source document. The GDS defines the `xlt_read_function_type` as:

```
int
function (
     void*                      source,
     char*                      buffer,
     int                        buffer_size);
```

Where `source` is the document (usually an `iostream` or some other file descriptor/handle), `buffer` holds the next chunk of text from reading the document, and `buffer_size` is the number of characters that can be read at once. This callback function is responsible for reading from the source document and writing into the provided buffer. The function returns the actual number of characters written into the buffer.

A.3 Data Structures

`xlt_extract` is the only public data structure of interest to the users of the GDS. It is defined as:

Table A.2: The GDS API Error Codes

Error	Description and Comment
XLT_NO_ERROR	Success.
XLT_SUMMARIZER_INIT_ERROR	The GDS was unable to read the .sum file correctly. This file has been corrupted.
XLT_INSUFFICIENT_MEMORY XLT_OUT_OF_MEMORY	The GDS has run out of memory and cannot proceed.
XLT_OTHER_ERROR XLT_SOURCE_ERROR XLT_MISC_ERROR	Miscellaneous fatal errors—please report these.

```
typedef struct
{
     long start;
     int len;
     int score;
} xlt_extract;
```

`xlt_extract_sents()` and `xlt_extract_keyphrases()` both use this data structure to return information. In the former case, the offset and length describe a summary sentence. In the latter case, the offset and length describe a keyphrase. `start` is the byte offset into the source document, `len` is the number of bytes, and `score` is the score of the summary sentence (normalized so the best sentence has a score of 100).

A.4 Error Codes

Table A.2 enumerates the most common error codes defined by the GDS API.

Appendix B

Supported HTML Tags

Table B.1: Supported HTML Tags

Tag	Support	Ending	Comments
A	standard	required	Used in the HTML hypertext link heuristic. Note: Browsers often allow <A NAME> tags without matching </A>. Often the only thing between a pair of <A> and </A> tags is an image, which is not very useful.
ADDRESS	standard	required	Considered a junktag. Although this information (the text in between) could be useful, it can disrupt the flow of text.
APPLET	Netscape	required	Considered a junktag. The text in between defines a Java applet, which the GDS does not understand.
AREA	Netscape	none	Considered a junktag and has no possible future use.
B	standard	required	Used in the HTML bold heuristic. Used much more frequently than its cousin <STRONG>.
BASE	standard	none	Considered a junktag since it contains no text.
BASEFONT	Netscape	optional	The GDS keeps track of font size information. At this point, however, no heuristic exists to take advantage of it.
BGSOUND	Explorer	none	Considered a junktag since it contains no text.
BIG	Netscape	required	Similar to <BASEFONT> and is used to keep track of the font size information.
BLINK	Netscape	required	Used in the HTML bold heuristic. Very rarely encountered.

The GDS supports tags from the HTML 2.1 standard. Specifically, the exhaustive list of tags is taken from the excellent *HTML: The Definitive Guide* [8]. The table below describes how the GDS treats each HTML tag in the absence of user customization.

Tag	Support	Ending	Comments
BLOCKQUOTE	standard	required	The tag itself is ignored, but the text in between the <BLOCKQUOTE> and </BLOCKQUOTE> pair is read as if it were regular text.
BODY	standard	optional	This tag is used to determine the start-of-text for HTML documents. Typically, the title sentence must also come before this tag in a document.
BR	standard	none	Treated as an end-of-paragraph marker (*i.e.*, the EOP marker).
CAPTION	standard	none	The tag itself is ignored, but the text in between is read as if it were regular text.
CENTER	standard	required	This tag is treated as a heading tag, similar to <H1>. It is therefore used in the HTML heading heuristic.
CITE	standard	required	Treated in the same way as <I>.
CODE	standard	required	Considered a junktag because the GDS has no provision to handle programming languages.
COMMENT	Explorer	required	Considered a junktag because the comment in between does not fit in the flow of the document text.
DD	standard	optional	The tag itself is ignored and the text is read as regular text.
DFN	Explorer	required	Treated in the same way as <DD>.
DIR	standard	required	Considered a junktag because the text in between is not summarizable as regular text.
DIV	Netscape	optional	Treated as an end-of-paragraph marker
DL	standard	required	The tag is used in the HTML list heuristic.
DT	standard	required	The tag is used in the HTML list heuristic. In the future, these terms might want to be considered for keywords/keyphrases.
EM	standard	required	Used in the HTML emphasis heuristic
FONT	extension	required	Used to keep track of font size information.
FORM	standard	required	Considered a junktag. Everything in between <FORM> and </FORM> is ignored.
FRAME	Netscape	optional	This tag defines a frame but provides a pointer to the document instead of the actual document. Therefore, the GDS cannot make use of this tag.

Tag	Support	Ending	Comments
FRAMESET	Netscape	required	This tag defines a set of frames for a web page. See <FRAME> above.
H1 H2 H3 H4 H5 H6	standard	required	Used in the HTML heading heuristic.
HEAD	standard	optional	Used to figure out the start-of-text.
HR	standard	none	Treated as an EOP marker.
HTML	standard	optional	The tag is ignored when encountered. It conveys no useful information.
I	standard	required	Used in the HTML emphasis heuristic. Preferred by most over the <EM> tag.
IMG	standard	none	This tag is ignored when encountered. It contains no textual information.
ISINDEX	standard	none	This tag is ignored when encountered. It contains no useful text.
KBD	standard	required	Considered a junktag. Furthermore, all text in between <KBD> and </KBD> are ignored, since they are most likely just computer inputs.
LI	standard	optional	This tag is ignored when it is encountered.
LINK	standard	none	Ignore this tag because it is used to define relationships between web documents.
LISTING	deprecated	required	Used to keep track of the other formatting tags (*e.g.*, <B>).
MAP	extension	required	Considered a junktag since it contains no text.
MARQUEE	Explorer	required	Considered a junktag since the text included here is most likely irrelevant to the rest of the document.
MENU	standard	required	Used in the HTML list heuristic.
META	standard	none	Considered a junktag since the tag does not include useful textual information.
NEXTID	standard	none	Considered a junktag since the tag does not include useful textual information.
NOBR	standard	required	The tag is ignored while the text in between is processed.

Tag	Support	Ending	Comments
NOFRAMES	Netscape	optional	This part of the <FRAME> definition provides the alternative document to browsers that do not support frames. The GDS will read the text in between <NOFRAMES> and </NOFRAMES> as if it were part of the HTML document.
OL	standard	optional	Used in the HTML list heuristic.
OPTION	standard	optional	This tag and any enclosed text is ignored because it does not contain any useful textual information.
P	standard	optional	This tag is treated as an EOP mark.
PARAM	Netscape	none	Considered a junktag because it supplies parameters to Java applets and not useful for summarization.
PLAINTEXT	standard	none	Used to keep track of the other formatting tags (*e.g.*, <B>).
PRE	standard	required	Used to keep track of the other formatting tags (*e.g.*, <B>). See <PLAINTEXT>.
S	Explorer	required	This tag and the enclosed text are ignored because it marks text for "strike-through."
SAMP	standard	required	Not implemented. The tokenizer has not been written to handle this tag.
SELECT	standard	required	Ignored because it contains no text.
SMALL	Netscape	required	Used to keep track of font size information.
STRIKE	standard	required	This tag and enclosed text are both ignored. See <S>.
STRONG	standard	required	Used in the HTML bold heuristic.
SUB	standard	required	This tag and the enclosed text are ignored because subscript text is usually insubstantial.
SUP	standard	required	This tag and the enclosed text are ignored. See <SUB>.
TABLE	standard	required	All table-related tags are ignored, but the text is processed as regular text.
TD	standard	optional	Ignored. See <TABLE>.
TEXTAREA	standard	required	Ignored. See <FORM>.
TH	standard	optional	Ignored. See <TABLE>.
TITLE	standard	required	Used to detect the document's title sentence. Note that the sentence or text in between the <TITLE> pair may not be the best title sentence. The GDS actually looks at other sentences too.

Tag	Support	Ending	Comments
TR	standard	optional	Ignored. See <TABLE>.
TT	standard	required	Ignore the tag and treat the text as normal.
U	standard	required	Used in the HTML emphasis heuristic.
UL	standard	required	Used in the HTML list heuristic.
VAR	standard	required	Ignore the tag and treat the text as normal (even though it is supposed to be variable names).
WBR	standard	none	Ignore the tag since it provides no information for the GDS.
XMP	deprecated	required	Used to keep track of the other formatting tags (*e.g.*, <B>).
!-- --	standard	none	This comment tag is ignored.

Bibliography

[1] Francine Chen, Marti Hearst, Julian Kupiec, Jan Pedersen, and Lynn Wilcox. Mixed Media Access. Via the world-wide web at `http://www.csdl.tamu.edu/DL94/position/marti.html`.

[2] Robert Cole, Ed. A Survey of the State of the Art in Human Language Technology. Via the world-wide web at `http://www.cse.ogi.edu/CSLU/HLTsurvey/HLTsurvey.html`.

[3] Jack I. Fu. Internal Summarizer 2.0 Overview. Internal Xerox XSoft Document, October 1996.

[4] P. S. Jacobs and L. F. Rau. Scisor: Extracting Information from On-Line News. *Communications of the ACM*, 33(11):88–97, 1990.

[5] Julian Kupiec, Jan Pedersen, and Francine Chen. A Trainable Document Summarizer. In *Proceedings of the 18th Annual International ACM/SIGIR Conference*, pages 68–73, Pittsburgh, Pennsylvania, June 1995. ACM Press.

[6] Robert F. Lorch, Jr. and Elizabeth Pugzles Lorch. Effects of Headings on Text Recall and Summarization. *Contemporary Educational Psychology*, 21:261–278, 1996.

[7] Kathleen McKeown and Dragomir R. Radev. Generating Summaries of Multiple News Articles. In *Proceedings of the 18th Annual International ACM/SIGIR Conference*, pages 74–82, Pittsburgh, Pennsylvania, June 1995. ACM Press.

[8] Chuck Musciano and Bill Kennedy. *HTML: The Definitive Guide.* O'Reilly & Associates, Inc., Sebastopol, California, 1996.

[9] Chris D. Paice and Paul A. Jones. The Identification of Important Concepts in Highly Structured Technical Papers. In *Proceedings of the 16th Annual International ACM/SIGIR Conference*, pages 69–78, Pittsburgh, Pennsylvania, June 1993. ACM Press.

[10] Ellen Riloff. Little Words Can Make a Big Difference for Text Classification. In *Proceedings of the 18th Annual International ACM/SIGIR Conference*, pages 130–136, Pittsburgh, Pennsylvania, June 1995. ACM Press.

[11] Gerard Salton, James Allan, Chris Buckley, and Amit Singhal. Automatic Analysis, Theme Generation, and Summarization of Machine-Readable Texts. *Science*, 264:1421–1426, June 1994.

[12] LinguistX Technology Reference, Version 2.0. Internal Xerox XSoft Document, October 1996.