An Empirical Study of Automatic Document Extraction

by

Irene M. Wilson

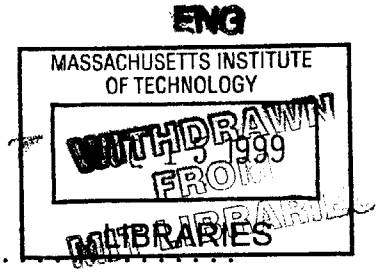
Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 1999



Author .....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by .....
Howard Shrobe
Associate Director, MIT AI Lab

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

An Empirical Study of Automatic Document Extraction

by

Irene M. Wilson

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, 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 paper presents an information extraction system designed to function on all types of textual input. It uses a combination of several statistical methods to extract a user-specified number of sentences. In addition, to accommodate for the wide variety of input types, a different "template" is used to cater to the statistical patterns of each type. This project provides a flexible framework that could be easily extended to become a trainable, multi-use extractor. Although it is not limited to any single application, this product was specifically developed to be used in conjunction with the START natural language processor. Together, they are capable of assimilating and storing large quantities of diverse information for retrieval.

Thesis Supervisor: Howard Shrobe
Title: Associate Director, MIT AI Lab

Contents

List of Figures

Chapter 1

Introduction

As time goes by, the ability to automatically summarize text has become more and more useful. However, despite the fact that automatic summarization has been attempted for decades, current techniques are far from perfect. The extractor which is presented in this thesis, the AutoExtractor, utilizes the techniques of several other systems in the attempt to create a flexible and robust method of summarization.

1.1 Why Summarize?

As the years go by and technology progresses, the amount of data available continues to increase exponentially. We are besieged by mass amounts of information that we have no hope of assimilating. It is therefore becoming increasingly valuable to be able to sift quickly though information to discover its content [7].

Consider, for instance, an organization that is interested in keeping abreast of all recent information pertaining to a particular subject, such as the import and export of a type of material. It would begin by collecting all possible documents that might contain this information. New data would have to be collected day by day, forming a continuous stream of information. Most of this information would be completely irrelevant. It might then try to narrow its search by looking for key words or phrases that could indicate the presence of a relevant document. However, it would undoubtedly find that even this narrowed search is prohibitively large; it simply does

not have sufficient resources to process the documents to determine if they are valid or false hits. The cost of labor and the amount of information is simply too large.

The example given above is only one situation in which the ability to automatically summarize a document would be extremely valuable. As the amount of available information continues to increase, such examples become more and more common.

1.2 The Utility of Summarization

There are several distinct ways that a document summary can be useful. A few examples are listed below:

- a shortened version of the real document
- a way to determine if the real document is useful
- a text over which to search automatically
- a summary of the results of the activities described

For example, one might wish to create a summary for a document because one does not have time to read the entire document. Or one might need a summary to discover quickly if the document covers relevant subject matter. There are many diverse situations in which summaries are useful or even vital. The task of summarizing is therefore very widely attempted and different implementations have been utilized for many years.

1.3 The Challenge of Summarization

Despite the decades of accumulated research on the topic, however, a truly "intelligent" document summarizer is yet to be developed. This task, which is relatively simple for an educated adult, is simply counter-intuitive to the capabilities of a computer.

To implement a summarizer that is truly reliable and accurate in every situation, it must be able to process the text of the document and somehow "understand" the content. No statistical analysis techniques are completely trustworthy. The program must be able to analyze the essence of the document in order to understand what is important and always produce a summary that humans would find understandable and relevant.

Unfortunately, understanding the content of a document is no trivial task. Humans have access to massive amounts of data about the world and its interactions that a computer does not possess. No computer, at present, is capable of reading a sentence and understanding not only its literal import, but also all the implications that do not directly follow from the text. For this reason, the task of summarization is "AI hard"; in other words, it will never be satisfactorily completed until artificial intelligence is successfully created.

1.4 Project Goals

This fact, however, does not discourage continuing efforts in the area. This thesis project, though certainly not the only work of its kind, is unique in its combination of several attributes:

- utilizes a combination of several extracting techniques
- easily alterable to place emphasis on different techniques
- functions over large array of input types
- creates summaries to user's need and specification

The AutoExtractor combines several extracting techniques because it has become clear that any one technique will not perform robustly over such diverse input. Therefore, several techniques are used so that the disadvantages of each is compensated for by the advantages of the others.

It it written in a flexible format that makes it simple to utilize a different technique or place a new emphasis on certain document characteristics. This makes the AutoExtractor extremely malleable, and also lends itself to the use of a learning algorithm to automatically find the optimal setting for a given input-output type.

It functions relatively accurately over an unusually large and diverse array of input types. Most summarizers are either specific to a certain input type or overly generalized. Though the use of document type templates, the AutoExtractor handles any number of document types while retaining the statistical accuracy that is normally lost when the input type is generalized.

Finally, the AutoExtractor also can be easily adjusted to cater to the user's specific summary needs. Previously in this chapter, several possible uses for a summary were listed. Because each summary type indicates a different optimal summary content, the AutoExtractor templates can also be used to choose sentences that are specific to the type of summary needed.

Chapter 2

Background

Many techniques have been used over the years to solve the problem of automatic document summarization. This is simply because no one technique is flexible enough to solve the problem as a whole [12]. Each approach was developed to apply to specific types of situations. In the paragraphs below, I have briefly described a few of the more prevalent summarization methods. Each of these methods will be at least partially incorporated into the structure of my implementation.

2.1 Natural Language Processing

The most obvious and elegant way to summarize a document is to process the content and use the ideas and information gained to generate a more specific representation [6]. This is, theoretically, the technique that humans use to summarize. However, this approach is extremely difficult. Currently, no language processing system has been developed that can even approach the level of understanding that a human achieves almost effortlessly. The amount of interconnected information that a human uses to parse and conceptualize a piece of text is staggering. Trying to represent this knowledge artificially is a daunting task. As a result, natural language processing, at present, is only useful when the text it is processing falls into a specific category or format. The information base needed to represent this specialized data is then manageable.

2.2 Word Occurrence Statistics

Another common technique used in summarization is word occurrence statistics. This procedure is not complex in theory or application. It attempts to capture the essence of a document by calculating the frequency of usage of significant words. This information is usually stored in a vector, which can be compared with other such vectors using a trivial computation. This technique, while quite simple and elegant, is not always effective. While there often is a correlation between word occurrence and content, this is not always the case [10].

Word occurrence methods are a rather unique technique that has its own set of advantages and disadvantages. It can be especially helpful in situations where the type of the document is unknown. The algorithm used is completely independent of the document format, because it only uses frequency of word usage.

However, a word occurrence algorithm is also capable of making drastic errors. For instance, instead of only recording the frequency of the key words used in the document, it might become sidetracked by also recording the usage of unimportant words that have nothing to do with the actual topic of the document. In addition, many words in the English language are spelled the same but actually refer to completely different objects. A word occurrence algorithm pays no attention to this fact. All words that look the same are lumped together, regardless of whether they refer to the same thing. This would create a false correlation, for instance, if one were comparing two paragraphs, one of which was about the illegal drug "coke" (an abbreviation for cocaine) and Coke, the popular soft drink [11].

2.3 Other Statistical Methods

Statistical methods assume that the target sentences usually have certain characteristics. These characteristics can include, but are not limited to:

- sentence location in document
- structure of sentence

- occurrence of certain words or phrases in sentence

The document is searched for sentences whose characteristics are similar the the characteristics that key sentences often have. These are then chosen as good candidates for key sentences. Once again, these methods cannot determine whether a sentence actually contains relevant content; they can only make an educated guess based upon previously gathered statistical data.

2.4 Information Extraction

Unlike the previous two summarization techniques, the method of information extraction (also known as "template filling") does not pull sentences directly out of a document to form the summary. Instead, one need only fill in the blanks of a summary template that was pre-generated for a specific type of document [5].

For instance, a document might be known to have a certain format or a certain type of content. Since it is known to contain this data, a template can then generated for the summary of this document before it is even processed. Then the document is searched for the specific information to plug into the template. There is no real understanding of the content; the algorithm only looks for certain words in certain locations to fill the slots in the summary.

This technique is, obviously, very specialized. Each template can only be used for one certain type of document. If there are more than one type of document in the system, a new template much be generated for each. This is a very useful and accurate method in certain cases, but it is almost useless when one is trying to summarize input that cannot be easily categorized [3].

Chapter 3

Top Level Design

In combination with a natural language processor, my code creates a system that is capable of analyzing, documenting, storing, and retrieving textual data.

3.1 Project Background

The AutoExtractor was designed for a specific use as part of a larger project. This larger project is concerned with the difficulty of processing and storing large amounts of data. Its goal is to be able to automatically assimilate data such that it can later be accessed and utilized. The principle tool that is used to accomplish this is the START natural language processor. START is currently under construction in the MIT Artificial Intelligence Laboratory. While this system is not comprehensive, it is capable of processing typical sentences and representing them as interactions between basic objects. This information can then be accessed by querying the database of objects and interactions.

Unfortunately, given the difficulty of natural language processing, the START system is limited in its ability to process input. It cannot understand sentences that have a complicated sentence structure or that use unfamiliar vocabulary. However, START is capable of assimilating a more complicated piece of data by associating it with simple annotations. For example, a long, technical paper can be summarized into a few sentences that are simple enough for the NLP to process. These sentences

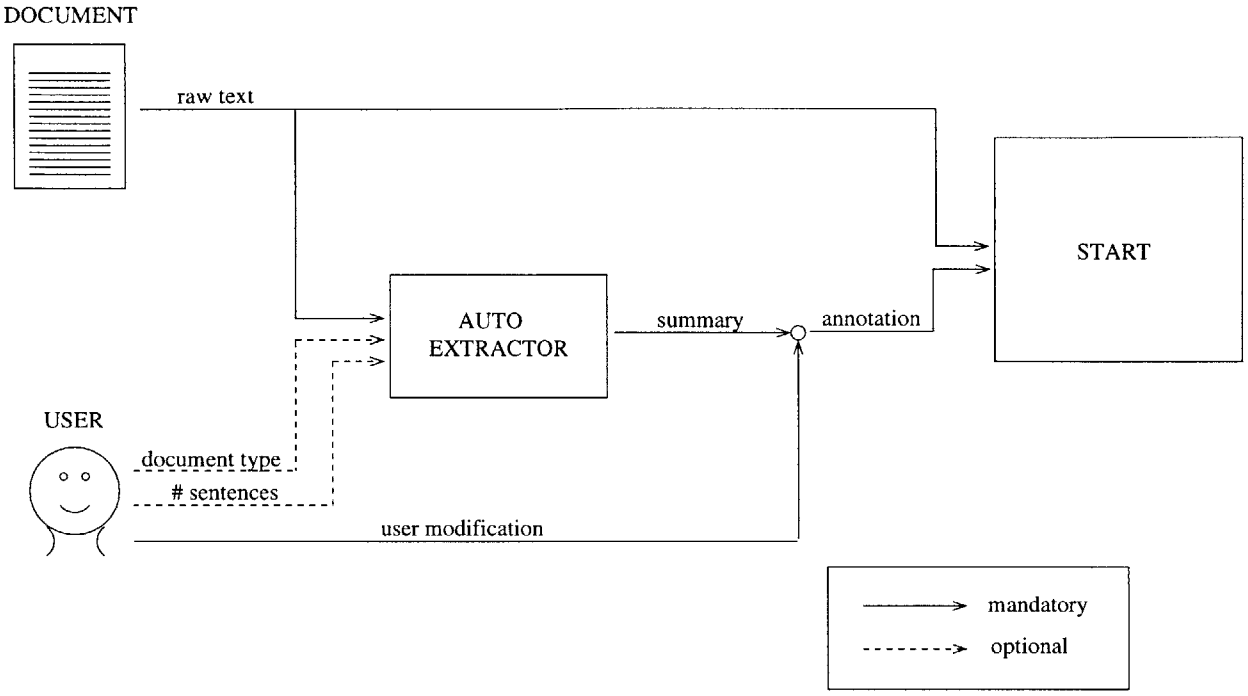


Figure 3-1: Top Level Design

are then added to the database, with the full text of the paper attached. In this way, very complicated text can be assimilated by the system. The same technique can be applied to other forms of data, such as pictures, charts, discussions, etc.

3.2 The Function of the AutoExtractor

The task of the AutoExtractor is to provide START with parseable annotations for text that is too long and/or too complicated for START to process. While there are many programs designed to construct summaries of text, the AutoExtractor is a sentence extraction algorithm. This means that instead of trying to construct sentences to form a coherent summary, the AutoExtractor chooses sentences verbatim from the document that accurately represent its content [8]. It is therefore based upon the theory that a coherent summarization can be constructed by selecting sentences or phrases from the document that are representative of the document as a whole. For instance, if one were to combine the topic sentence of each paragraph into one block of text, this text would probably be a good summary of the main ideas in the

document.

Finding the sentences which best represent a document is not always possible. In fact, there is no guarantee that such a collection of sentences exists. The alternative ways of generating a summary, however, were either too complex or too limiting for our use.

3.3 User Input Modifications

The optimal document processor would need no human aid to do its job. However, given the state of Artificial Intelligence at this time, the AutoExtractor output must be modified before it is fed to the START NLP.

3.3.1 Ensuring Sentences are Parseable by START

Although the complexity of the document assimilation has been greatly simplified by submitting annotations in place of the full document, it is still complicated by the fact that START cannot parse sentences that are overly complex in structure or content. Sentences that have several clauses, for instance, are often rejected. Also, the START program has a limited, if relatively large, lexicon. This means that if the sentence contains words that are not in the lexicon, the sentence often cannot be parsed or assimilated properly. Given the technical nature of most of the summaries my program will produce, chances are extremely high that at least some portion will be unprocessable by START because of either complex sentence structure or the usage of unknown vocabulary.

Therefore, the Autoextractor output must be altered to match the level of simplicity that the NLP requires. There are two possible ways to do this. One could either modify the code so that any sentences that are not parseable by the NLP are not selected to be in the document summary. Alternatively, one could generate the best possible summary, then modify the sentences to conform to the NLP's requirements. Ideally, the second option is more attractive because one would not be forced to discard the best summarizing sentences simply because they are too complex for

START to parse. Automatically altering the format and perhaps even the content of a sentence, however, was too complex a task to be tacked in the scope of this thesis. For this reason, the task of editing the sentences for START compatibility is at this time performed by a human operator.

3.3.2 The Optimal START Input

It was mentioned above that one must be careful that the sentences provided as input to the START system are not so complex in structure or vocabulary that START is not able to parse them. In addition, the sentences provided as annotation for larger documents must contain just the right amount and type of information to get the best results. Using certain types of input will result in greater accuracy and success in processing. In order to retrieve information about an item that has been entered into the START database, one must ask a question that matches that item of information. For example, if one might enter the following annotation into the START system:

> Bally Entertainment Corp. is seeking federal antitrust clearance to acquire a major stake in gambling rival Circus Circus Enterprises Inc.

To access this information, the user would have to enter a query about the objects created when the statement above was entered. One possible query would be:

> What corporations are seeking federal antitrust clearance?

Another possible query:

> What sort of relationship does Bally Entertainment Corp. have with Circus Circus Enterprises Inc.?

If the statement given above were part of the annotation for a larger document dealing with the business dealings of the Bally Entertainment Corp. with the Circus Circus Enterprises Inc., this would be a moderately good summarizing sentence. One of the queries given above would have a good chance of triggering the original statement and causing the entire document to be retrieved.

If the original document dealt instead with the subject of the Circus Circus Enterprises business dealings in general, however, this sentence would still trigger on both of the above queries, and START would respond to the query with a primarily irrelevant document. For this reason, it is important to prevent sentences that are too specific from being chosen for the summary.

In the same manner, it is also important not to choose sentences that are too general. In this case, a query that asks specifically about the data in the document might not trigger that document because the summary is so general that the two sentences would not match.

It is a delicate process to decide how much information to provide about a document. One must look at the problem not from the standpoint of a summarizer, but from the standpoint of a person that might wish to access this information. What queries would a person interested in that document be likely to ask? What sort of statements would trigger with those queries? Those statements are not necessarily of the same form as the sentences one would choose as the best summarizing sentences in the document.

3.3.3 Types of User Modifications

Because the AutoExtractor is not accurate enough to select only sentences that meet the above criteria, some extra modification is needed. At present, this modification is provided by the human user. After a document has been summarized, it is the responsibility of the user to choose those sentences from that summary that contain the right information and ensure that they are simple enough that the START system could process them. This usually entails:

- Eliminating extraneous clauses or sentences that add complexity or unnecessary vocabulary
- Selecting the sentences in the summary that specifically mention the main points in the document

- Eliminating the sentences or clauses that provide extraneous data about the document

While this creates undesirable overhead for the human operator, it is significantly preferable to the alternative of reading the document, finding the summarizing sentences, and then editing them for compatibility with the START system.

3.3.4 Automatic Sentence Modification

It is quite possible that the role of the human operator could be completely elimitated from the function of this system with only a slight decrease in accuracy of sentence selection. A new piece of code could be written to evaluate the sentences chosen by the AutoExtractor and alter them as necessary to conform to the START input requirements.

For instance, if a sentence contains a clause that mentions information that is not relevant to the main topic of the document, this clause can be removed automatically. A sentence may contain several clauses that cause the sentence to be very complex, but are not necessary to retain the main point of the sentence. Consider the following sentence:

> There was significant growth in manufacturing activity during the month, overtaking previous record levels, and prices were forced up as suppliers failed to meet the increase in demand.

The phrase "overtaking previous record levels" causes extra complexity, but can be removed without destroying the key content of the sentence.

The same technique could be applied to sentences that are too complex for START to parse. These sentences could be simplified by removing clauses or separating the original sentence into multiple sentences.

The problem of the limited lexicon is more simple to overcome. While START is capable of processing sentences without outside aid, it also can utilize user-supplied "hints". For instance, if the sentence contains a proper noun that is not contained in

the lexicon, START must simply be informed that the word is a noun, to be able to process the sentence. This type of information can easily be supplied automatically through any of a number of sentence parsers.

It is difficult to know with any certainty whether it would be possible to create a piece of code to seamlessly link the AutoExtractor to the START NLP. More research needs to be done to determine exactly how the output of the AutoExtractor relates to the optimal input to START, and what sorts of alterations typically need to be done. If these alterations form predictable patterns, it may be possible to create a program to perform the alterations automatically and eliminate the role of the human operator completely.

Chapter 4

Sentence Extractor Design

As this report progresses, the individual components of the AutoExtractor will be examined in more detail. Its basic technical function, however, is as follows:

1. The document type of the input must be established. This information can either be provided by the user or automatically detected in the Document Type Recognizer. See section 5.1.1 for more information.

2. The reformatter must take the raw input of the text and alter it so that it conforms with the format that the rest of the AutoExtractor is programmed to recognize. It uses the knowledge of the document type to do this. See section 5.3 for more information.

3. The Parser breaks down the document into a tree of structure objects that hold all necessary information about the document. See section 5.2 for more information.

4. The Characteristic Evaluator creates a sentence characteristic object for each sentence, which holds all the characteristics needed to compute the goodness of a sentence for extraction.

5. The Calculator uses the appropriate document template in conjunction with the sentence characteristic values to calculate a overall rating for each sentence. See section 4.3.1 for more information.

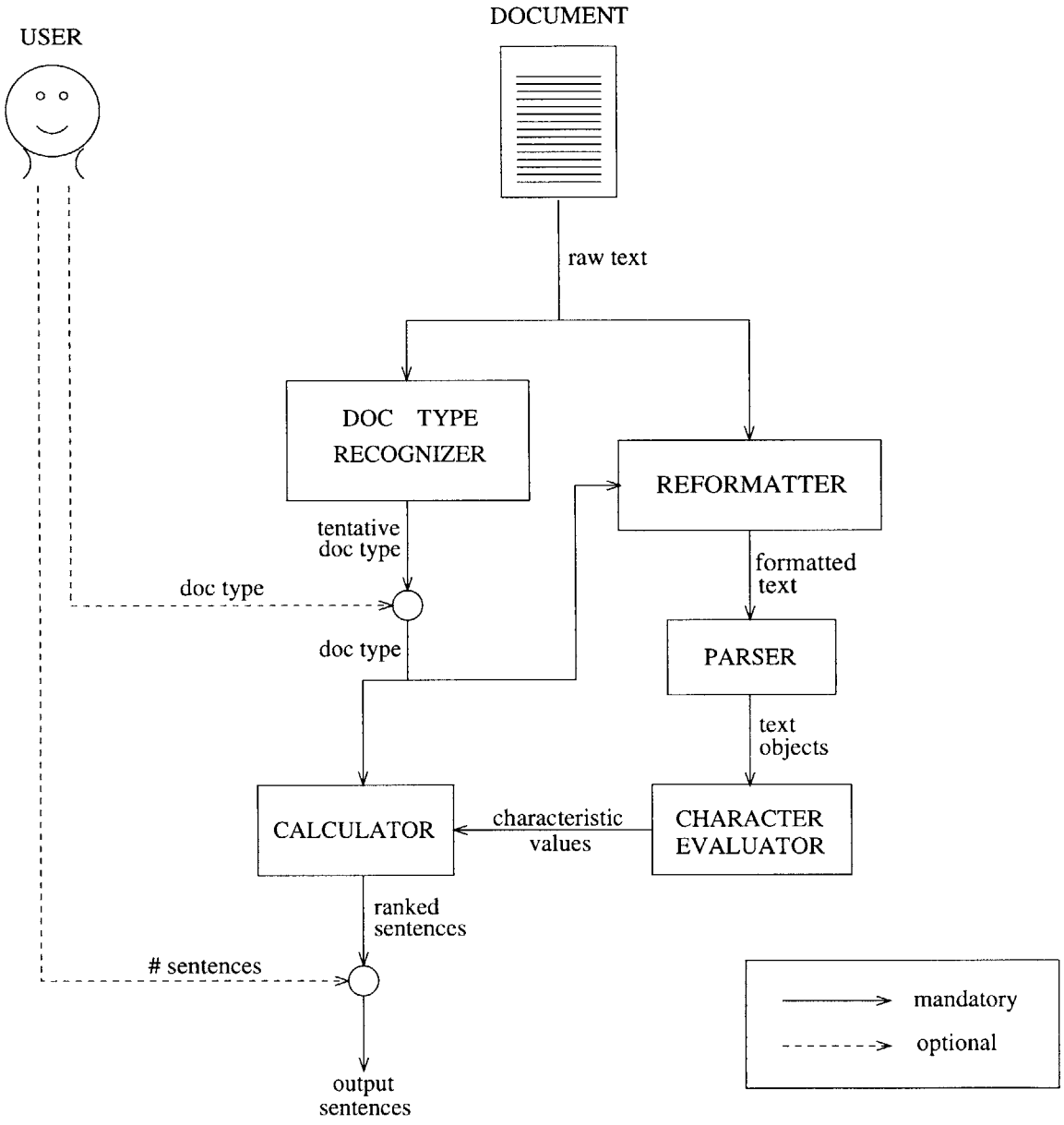


Figure 4-1: AutoExtractor Design

6. The top n sentences are returned as the output of the AutoExtractor, where n is determined by the user.

While all aspects of the technical implementation of the AutoExtractor will not be explained in detail in this chapter, there are several design decisions and techniques that will be discussed in this chapter. Some non-central design decisions are also discussed in chapter 5, "Design Issues".

Some sample input documents with their AutoExtractor generated summaries are included in Appendix C.

4.1 Design Principles

None of the popular summarization methods mentioned in the background section appear to contain all the qualities that are necessary to create a robust, adaptable way to process textual information. Therefore, the AutoExtractor is a hybrid of sevaeral techniques that combines to form an algorithm that is both accurate and adaptable. Alone or together, no method can guarantee that an accurate, coherent summary will be generated. By combining techniques, however, the flaws in one implementation can be compensated for by another. This results in an overall greater probability that an acceptable summary will be generated.

The following sections will describe how several different summarizing techniques are combined in the AutoExtractor.

4.1.1 Using Natural Language Processing

As was mentioned earlier in this paper, it is effectively impossible at this time to create a natural language processing system that can interpret and understand language as well as a human. However, systems have been developed that can process certain types of input. The START NLP, mentioned above, is one such system.

Therefore, while the technique of natural language processing is not currently utilized in the AutoExtractor to any appreciable extent, the project as a whole is ex-

tremely dependent on START to process the output of the AutoExtractor and record the facts represented there. The AutoExtractor does not have to process the text at all or make any judgements about its content. In addition, the primary weakness of natural language processing, namely, the extreme difficulty of processing complicated text, is overcome by using the other summarizing techniques in the AutoExtractor.

The task of the other summarizing techniques is now reduced to changing the content of a document into a format that can be understood by the NLP. While this is significantly easier than analyzing the document from scratch, it is still a formidable task.

4.1.2 Using Word Occurrence Statistics

Word occurrence is one of the primary methods used in the AutoExtractor. While its utility is greatly affected by the type of document being processed, in certain situations there is simply no other technique that is useful.

Studies have shown that this technique is not a particularly accurate method of extracting information from structured, predictable documents like technical reports or newspaper articles. For other types of input, like informal emails or speeches, its use can be vital. When a document has little structure, the best way to decide which sentences are key is to trigger off the words in the sentence. For this reason, word occurrence is an extremely valuable technique when dealing with such varied and unpredictable input.

4.1.3 Using Statistical Methods

There are several other statistical methods that are used in the AutoExtractor in conjunction with the word occurrence techniques. Specifically, it analyzes the location and format of each sentence and calculates a fitness number based upon its characteristics. This is a very powerful technique that has proven to be successful in other experiments [12]. While not as specialized as information retrieval techniques, statistical methods are much more useful when the document type is known. With the

help of document templates to cater to the statistical information of each document type, statistical methods provide some useful information for almost any input.

4.1.4 Using Information Extraction

The technique of information extraction is so specific to a certain type of document that it is almost useless when the input is very diverse [12]. However, some related methods can be very useful.

One example of information extraction in the AutoExtractor is the use of section and paragraph headings. The program looks for specific words and phrases in the headings to indicate that the following paragraphs are likely to contain good summarizing sentences. This is related to information extraction techniques in that the program triggers off key words and phrases to find the information it desires.

In addition, the AutoExtractor searches the text to detect words or phrases that might indicate the presence of a good summarizing sentence. Without understanding anything of the context or meaning, it gives a sentence that contains certain phrases higher probability of being selected. This also is a common technique in information extraction.

While information extraction is not as useful when the input is very diverse, it can be helpful when used in conjunction with other techniques by providing statistical clues that otherwise would be lost.

4.2 Analyzing Sentence Characteristics

There are many factors that may indicate that a sentence should be chosen as a summarizing sentence for a document. When a person tries to find key sentences for a document, but he does not have time to read it and generate his own, he will probably use this sentence extraction method. He will scan the document, looking in the most likely locations first. He will look for sentences that have a certain format and structure, and that contain words or phrases that seem relevant to the rest of the document. These words or phrases may seem important because they are mentioned

often throughout the paper or in the section headings. These are all techniques that the AutoExtractor uses. It is incapable of reading and understanding the text, but it is able to mimic the other techniques a human might use.

4.2.1 Word Occurrence Statistics

Analyzing word occurrence is the technique used to determine if a sentence contains words that are important in the document as a whole. In this way, it determines that one sentence is more likely to be a good representation of the document content than another. It is discussed in more detail in the previous chapter.

As each word in the document is read, it is analyzed to see if it is a significant word- that is, not a proposition, conjunction, etc. If it is significant, it is added to a list that keeps track of what significant words the document contains. It is also added to a similar list for the specific sentence it is in. These lists are saved in the form of a vector. Each location in the vector represents a certain word, and the value of that location is how many times that word appears in the document or sentence. When the document is completely parsed, the sentence vectors are then scaled and compared against the document vector. If a sentence vector is very similar to the document vector, it theoretically contains a good representative selection of words from the document.

In the AutoExtractor, the method of choosing keywords is quite simple. A permanent list of non-keywords is constructed, and as long as the word is not in the list, it is assumed to be a keyword. The non-keyword list consists of prepositions, determiners, and other words that are usually irrelevant to the content of the sentence. Using this technique, many unimportant words are included in the word vector, but at least very few significant words are excluded. In addition, this method takes a negligible amount of time and is trivial to implement.

4.2.2 Other Statistical Methods

In addition to word occurrence, several other types of statistical methods are extensively used in the AutoExtractor. For each sentence, certain characteristics are recorded that may have an impact on the likelihood that the sentence is a key sentence.

Sentence Location

When a human is pressed for time and wishes to quickly find the main points of a document, he will skim certain locations first. For instance, he may read the paragraph labeled "introduction" or "conclusion". He is very unlikely to pick a random paragraph in the middle of the text and expect it to give him good insight into the content of the document as a whole. This is because the good summarizing sentences are usually located in certain places in the text.

The AutoExtractor utilizes this fact by taking note of the sentences that occur in these special areas. The locational characteristics that the AutoExtractor records are:

- sentence location in the paragraph
- paragraph location in the section
- section location in document
- the paragraph or section heading

The first, second, or last sentence in the paragraph is often the topic sentence of the paragraph. Therefore, it is important to record the sentence location in the paragraph. It is often also necessary to remember information about the paragraph and section in which the sentence appears. First or last paragraphs often are summaries of the section they appear in, and so are more likely to contain key sentences. Sections at the beginning or end of a document are also often more likely to contain key sentences.

In addition to the location of the paragraph or section, it might also be important to analyze the titles or headings under which the sentence appears. For instance, one

is clearly more likely to find a key sentence under a section entitled "Abstract". For this reason, the AutoExtractor has a list of phrases that it searches for in each title or heading. If one of these "title-phrases" are found, all sentences under that title may be given an advantage.

Sentence Characteristics

The characteristics of the sentence itself can also be important when trying to find a likely key sentence. The sentence structure characteristics that the AutoExtractor records are:

- length of sentence
- occurrence of key words or phrases in sentence

The length of the sentence, for instance, can be useful information if one wishes prevent the AutoExtractor from choosing abnormally short sentences as key sentences. Also, one can decide to give an advantage to medium or longer sentences, which may be more likely to be good key sentences than short ones.

The AutoExtractor also records which sentences contain "keyphrases". A keyphrase is any phrase that might occur more frequently in good summarizing sentences. For instance, if a sentence includes the phrase "this report shows", this may indicate that it is a good sentence to include in the summary. When a sentence contains a keyword, this is recorded for later use in calculating sentence fitness.

4.3 Evaluate Sentence Keyness

Once all relevant information about the sentences of the document has been gathered, the program must then decide which sentences should be chosen. This is accomplished by calculating a total score for each sentence based upon the characteristics it has and the document type.

4.3.1 Calculations

The AutoExtractor was designed to use a very straightforward and intuitive method of calculating sentence scores. This is so that it is simple and easy to make modifications.

All the sentences in the document have certain recorded characteristics. For example, each sentence has a length, position, keyphrase content, etc. Each of these characteristic has a weight which represents how much impact that characteristic will have on the probability that the sentence is good to include in the summary. For instance, the length of a sentence may be much less important than the location of the sentence in its paragraph. Therefore, the weight of the length characteristic would be much lower than the weight of the paragraph location characteristic. Characteristic weights must be greater than or equal to zero. There is no upper limit the characteristic weights, and they do not have to sum to any total.

There are many possible values for each sentence characteristic. For example, the sentence length characteristic can have the values very short, short, medium, or long. Each of these values is assigned a score. For instance, if very short sentences are very unlikely to be good key sentences, it will have a score of zero. A short sentence might be a little more likely, so it will be given a score of .3. Medium and long sentences might be equally likely to be key sentences, so they are given a score of 1. Characteristic scores must be equal to or greater than zero, and equal to or less than 1. They do not have to sum to any value. The best the value(s) for a characteristic should be assigned the score of 1. The worst should be assigned a zero. If this is not the case, the program will still work, but the characteristic will effectively not be given the weight that was intended.

Calculating a sentence's score from its characteristic values is trivial. First, one must multiply each characteristic weight by the score of its value. The sentence score is computed by adding these numbers together.

Assume that c_n refers to sentence characteristic n. $w(c_n)$ then refers to the weight of characteristic n. $v(c_n)$ represents the value assigned to characteristic n, and $s(v(c_n))$ represents the score of the value assigned to characteristic n. Then the figure below

represents the method of calculating a sentence score:

$$\sum_i w(c_i)s(v(c_i)) \tag{4.1}$$

There are several advantages to this rather simplistic method of computing sentence fitness. First, it is trivial to add or remove sentence characteristics to the equation. No modifications need to be made to the existing characteristic weights or value scores. The addition or removal of a characteristic only increases or decreases the total possible sentence fitness. Since the comparative fitness between sentences is the only significant factor, changing the total possible fitness score is irrelevant. One can also add or remove characteristic values without altering any other characteristic. If, for instance, one wanted to make a distinction between "long" sentence length and "very long" sentence length, one need only add the "very long" sentence length and assign it a score. One might also need to alter the scores of the other values in the characteristic.

In addition to being easy to alter or extend, this method of sentence scoring is also very intuitive. The importance of a document characteristic is simply determined by its numerical weight. To compare its importance with the importance of another characteristic, one need only compare the two weights. If one weight is twice as large as the other, that characteristic is twice as significant.

4.3.2 Document templates

The characteristic weights and slot values mentioned above must be assigned numbers such that the AutoExtractor is most likely to choose good summarizing sentences. Because of the wide variety of input and output types, however, document templates are used to cater to the statistical peculiarities of each input and output combination. See appendix E for an example of document templates.

Templates for Document Types

In order to process varied input data, the AutoExtractor was created with the ability to use "templates", which make it possible to utilize the statistical data of that document type to its fullest extent.

Most document summarization projects have a limited type of input material. The AutoExtractor, however, was designed to summarize any type of input. This could include but is not limited to:

- technical reports
- documentation
- newspaper articles
- discussions
- verbal presentations
- informal discourse

The list of possibilities is long and varied, and each one of these document types is quite unique. They use different vocabulary and sentence structure, in addition to having a different document structure and formatting. The information content is in different locations and in varying concentrations. It would be extremely difficult to generalize over this entire set of documents. In fact, it may very well be impossible.

For example, one might examine a newspaper article versus a verbal presentation. In a newspaper article, the most important information is concentrated heavily in the beginning of the article. The first paragraph almost always contains all the vital information in the article. In response, one might increase the weight of the "paragraph location" characteristic and give the "first paragraph" value a score of 1. This means that the fact that a sentence is in the first paragraph would have a large positive effect on the overall sentence fitness score.

However, a verbal presentation does not have the same structure as a newspaper article. A speaker will often start the presentation with a long, roundabout introduction, and will often not address the central issue until well into the speech. For this type of document, the weight of the "paragraph location" characteristic might not be very high, since a speaker may come to the point at almost any time during the speech. Also, the "first paragraph" score of the "paragraph location" characteristic will probably not be 1, since a paragraph in the middle of the speech is more likely to contain a good summarizing sentence than the first one.

The example given above is only one of many difficulties that arise when one attempts to generalize over different types of documents. In order to have any sort of success using statistical extraction methods, different types of documents must use different statistical data. For this reason, the AutoExtractor was designed to analyze different types of documents using their own statistical data. Each document type has a "template" that contains all the characteristics that that is significant for that document type, and all the value scores for those characteristics. It is a simple matter to add a new type of document to the system; one must only create a new template file.

Templates for Summary Types

In addition to using templates to cater to the statistical information about a specific document type, templates can be used to create summaries that are customized to the needs of the user.

In the introduction to this paper, it was mentioned that there are many possible applications for a document summary. The optimal summary for one application is not necessarily the best summary for a different application. For instance, if one needed a summary to determine whether the complete document was relevant, the optimal summary would contain all the main subjects discussed. If one needed a summary of the results of the experiment discussed in the paper, however, the optimal summary would contain the basic facts of the experiment performed and the results. Therefore, two "good" summaries of the same document can have very different con-

tent based upon the type of summary needed.

The fact that there are so many different uses for a summary makes the job of the summarizer even more complex. To create a summary specific to a certain use, one must create a template that weights heavily those characteristics needed in the extracted sentences. In this way, the AutoExtractor can be modified to extract different types of sentences.

Although the summarizer was designed specifically to manufacture the type of document that is well suited as input to the START natural language processing system, the template system makes it simple to expand its capabilities to other uses.

Chapter 5

Design Issues

There were several interesting and challenging aspects of this project that were not necessarily central to the issues of document extraction. These are discussed below.

5.1 Determining Document Type

In order to use the correct template, the AutoExtractor must know what type of document it is evaluating. There are two ways to accomplish this: either the user can explicitly tell the system the type of document, or the AutoExtractor can look at the characteristics of the document and make a guess.

5.1.1 Automatically Determining Document Type

There are many document characteristics that can indicate that a document is of a certain type. For instance, certain types of documents may have specific headers or signatures. If there is a finite number of document formats that will be used as input to the AutoExtractor, one can be sure that certain characteristics, like the content of the header, will be present in order to categorize the input into these preset types. This method is both fast and accurate.

If the system is not guaranteed to receive input in a certain format, however, it must resort to other, more subjective methods. In this case, some or all of the

following indicators must be used:

- content of headings and subheadings
- formatting
  - number and size of paragraphs and section
  - presence of lists and/or figures
- document length
- writing style
  - type of vocabulary
  - sentence length
  - sentence structure

One may in fact be able to categorize an input based solely on the format or tone of the document. For instance, if the document has section and paragraph headings, and if some of these headings contain the keywords "abstract", "introduction" or "conclusion", this could indicate that the document is a formal report. On the other hand, if the document has a less rigid structure and contains simple, informal vocabulary and punctuation, it might be an informal email or speech transcript. This method of document categorization is more computationally intensive and might not always be accurate. However, any attempt at categorization is preferable to randomly choosing a template that may have no statistical relevance to the document in question.

5.1.2 Direct User Type Input

The simplest and most accurate method of categorization, of course, is to explicitly tell the system the type of the input it is receiving. This requires that the inputs are already categorized or that a human operator is available to categorize the documents as they are entered. If this is not practical, the more subjective methods mentioned above must be relied upon.

5.1.3 Necessary User Modifications

At present, the portion of code that controls document recognition is incomplete. This is because there is no information available at this time regarding the types of the documents that will be used as input. In fact, this section of code will need to be adjusted for each different environment in which the system is used. Currently, the code analyzes the document and records information that might be useful for categorization, such as the header, section titles, word frequencies, keyword usage, etc. To recognize a certain type of document, one need only check if it contains certain distinguishing characteristic(s) and return the appropriate document type. If the user, however, does not have the time or knowledge to alter the code in this fashion, it is always possible to simply choose a default categorization (resulting in some loss of accuracy) or categorize the inputs by hand.

5.2 Parsing Issues

Parsing the input documents, though not part of the experimental aspect of this project, was nevertheless quite challenging. There are several products available that can automatically parse a document, but I chose to write my own parser instead. This is because I wanted to have the flexibility of parsing exactly where and what was needed specifically for the AutoExtractor product.

For example, the parser for the AutoExtractor must be able to detect paragraph and section breaks in documents that do not contain tags to indicate these locations. The parser also must be able to record which sentences are in each paragraph and section.

Perhaps most importantly, however, one can never be sure what document characteristic might become an important piece of information for extraction. Using a commercial or pre-written parser would create a risk of being unable to modify the parser to record the information needed. Since I wrote my own parser, however, gathering new data about a document is as simple as modifying a few lines of code.

The information that the AutoExtractor parser records at this time is listed below:

- document heading
- section breaks
- paragraph breaks
- section titles
- paragraph titles
- fragment sentences
- total number of sentences
- total word content for the document
- sentence text

5.2.1 Possible Parser Modifications

There are, however, several aspects of a document that may be useful in the detection of good summarizing sentences that are not currently detected by the document parser.

One significant example of this is the comma. If the parser were able to recognize commas in the text, it could gather valuable information about the structure and complexity of a sentence. In some situations, this information could be vital in detecting or eliminating sentences of a certain structure.

Another piece of information that the AutoExtractor parser does not record is recursive section breaks. Currently, the parser can only distinguish two levels of structure in a document: the section and the paragraph. Therefore any subsections, subsubsections, etc. are simply interpreted as paragraphs. This limits the AutoExtractor's ability to gain information from the substructure of the document.

There is an almost infinite amount of information that the parser could glean from a document. As the AutoExtractor continues to be developed and honed for accuracy, the truly significant aspects of the document will be pinpointed. At that time, the parser can be modified to collect exactly the amount of data needed.

5.3 Compensating for Different Formats

In addition to causing problems with statistical variation, the wide diversity of input types also causes basic formatting difficulties. Each of the different input types has its own way of delimitating sections, paragraph breaks, headings, tables, etc. For instance, an HTML document appears quite different from a Word document or an informal email, even though all these documents can contain the same basic structures.

It would be extremely difficult to train the parser to recognize all the different formats that it may receive as input. Therefore, the AutoExtractor first runs the document through a reformatter to alter the input document so that its structure is of the format that the parser has been trained to recognize. The basic characteristics of the standard parser format are as follows:

- Fragments at the beginning of the document are headings
- Sections are separated by two or more returns
- Paragraphs are separated by one return
- Titles are fragments separated by returns
- No carriage returns at the end of lines

For example, if an informal email was used as input, it would probably be in a raw text format with carriage returns at the end of each line and large header at the beginning of the file. When entered into the reformatter, the email header would be removed and the carriage returns at the end of each line would be removed.

If an HTML document were entered as input, the formatter would examine the tags and reformat the document accordingly. Where a new section begins, it would separate the text with two returns. New paragraph tags would cause the text to be separated with one return. The title would be placed on a line separated by returns. Any HTML code or tags would be removed.

The reformatter is a portion of code that is impossible to write before the input document types are known. Documents vary widely in their format, and it is simply not possible to anticipate the changes a new input type would require. Therefore, once a new input type is known, the reformatter must be augmented to handle this new type. Its task is greatly simplified by the fact that the document type is determined by the document type recognizer. See section 5.1 for more information on document type recognition.

Chapter 6

Testing

Testing the performance of this system is not a simple task. There many interrelated variables factoring into the calculation of the best possible summarizing sentences. In addition, determining the success of a summarization is a very arbitrary and time consuming procedure.

6.1 Comparison Techniques

To test the AutoExtractor, its performance was compared against that of a human summarizer. This was accomplished by running a set of texts through the AutoExtractor and then having these same texts manually summarized. The human summarizers were instructed to choose the top $log_2(log_2 n - 2) - 1$ sentences that best summarized the text, where n is the number of sentences in the document. They were also instructed to choose the next $log_2 n - 2$ best summarizing sentences in the text. See appendix A for the specific instructions given.

To generate the document type templates for each different input type, one fourth of the test set was used. The template values were set so that they would maximize the performance on this segment of the testing set. The remainder of the testing set was then used to evaluate the actual performance.

6.2 Calculations

The summaries were scored using the following procedure:

Every sentence is a assigned a *rank*. If the AutoExtractor selects a sentence, the rank of that sentence is the order in which that sentence was selected. For example, the sentence that received the highest score from the AutoExtractor has rank one, the second best rank 2, and so on. Sentences that are not selected by the AutoExtractor are assigned rank 0. The rank of a sentence x is denoted r(x).

Assume the total set of sentences that the human underlines is u. The total set of sentences that the human circles is c. Therefore, the set $u_r > 0$ is the set of underlined sentences that have rank greater than zero; that is, the sentences that are both selected by the AutoExtractor and underlined by the human.

$|c|$ denotes the number of sentences in the set c. $|u|$ is the number of sentences in the set u. These numbers are determined as described above by the total number of sentences in the document.

The total score for a document summary is calculated using the following equation:

$$\frac{1}{|u|}\sum_{\{i:r(u_i)>0\}}\frac{2(|c|+|u|)-r(u_i)+1}{2(|c|+|u|))}+\frac{1}{|c|}\sum_{\{i:r(c_i)>0\}}\frac{2(|c|+|u|)-r(c_i)+1}{2(|c|+|u|))} \tag{6.1}$$

The rationale behind this method of scoring is as follows: The left half of the equation represents the score for the sentences that were underlined. The right half represents the sentences circled. The circled sentences represent the *best* summarizing sentences in the document. The underlined sentences indicate a larger volume of *good* summarizing sentences. These two subtotals are intended to be equally significant in the total score, indicating that choosing the best summarizing sentence is approximately as important as choosing several good sentences.

The higher the rank of an underlined or circled sentence, the larger the benefit to the summary score. Therefore, a sentence of rank c_r is assigned the value:

$$\frac{2(|c|+|u|)-r(u)+1}{2(|c|+|u|)}$$

This distributes the values of the ranked sentences from .5 to 1. The sentences with rank zero are assigned a value of zero.

The values of all underlined sentences are summed together and divided by the total number of underlined sentences. The same procedure is followed with the circled sentences. Then the two resulting numbers are added to create the final summary score.

6.3 Test Input

Several different types of documents were chosen as test input:

- 20 reports
- 20 documentation papers
- 20 transcribed speeches
- 20 newspaper articles

These four categories by no means represent the total array of possible input document types. They do, however, well illustrate the variety of input possible.

The reports consist of papers ranging from quite technical to moderately informal. All contain a title and are divided into sections and paragraphs. They were not intended to be related in any specific way; each report addresses different subject matter. They vary in length from one page to several chapters. The reports were found at the URLs listed in appendix D.

The documentation papers each describe the technical aspects of a system. Many of these papers are subsections of a larger body of documentation. The subjects of the documentation papers are quite varied. Some of the papers are taken from the same larger body of documentation. Otherwise, the content is unrelated. They are from one to several pages in length. The documentation documents were found at the URLs listed in appendix D.

The transcribed speeches are all from the same source; namely, presidential addresses to the public between the dates of March 15, 1997 and March 27, 1999. The format of the speeches are very similar, but the content varies in accordance with the current issues of the nation. Each speech is one to two pages in length.

Finally, all 20 newspaper articles are taken from the New York Times. The articles are from one to several pages in length. Their content is varied and unrelated.

6.4 Results

6.4.1 General Observations

The results of the testing were generally positive. Thirty-seven out of sixty document summaries, or 62%, contained a circled sentence; that is, a sentence that the human chose as the best summarizing sentence of the document. Only fourteen out of sixty summaries, or 23%, did not contain any of the sentences that the human selected. See appendix F for the machine and human results, as well as the actual scores of each summary.

As I personally looked through the results, I felt that most of the summaries did contain enough information to generally describe the topic of the document. However, it also became clear that many of the sentences selected are completely inappropriate as summarizing sentences. It is not difficult for a human to recognize which of these sentences are good and which are irrelevant. Perhaps there is a way to automate this process, so that less inappropriate sentences are included in the future.

6.4.2 Contrasting Input Types

The performance of the AutoExtractor varied widely across the four document types used as input. Figure 6-1 shows the average summary score and the summary standard deviation for each document type.

Figure 6-2 shows, for each document type, the percentage of AutoExtractor summaries which contained a circled sentence and the percentage of documents in which

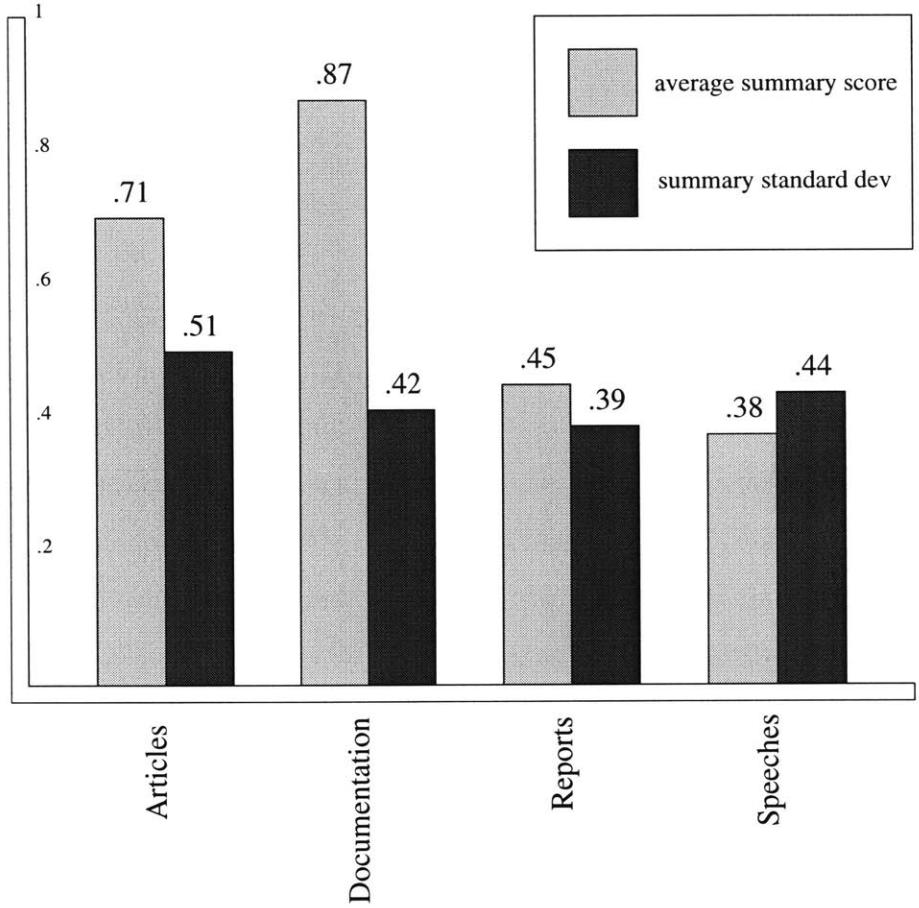


Figure 6-1: Summary Scores and Standard Deviation

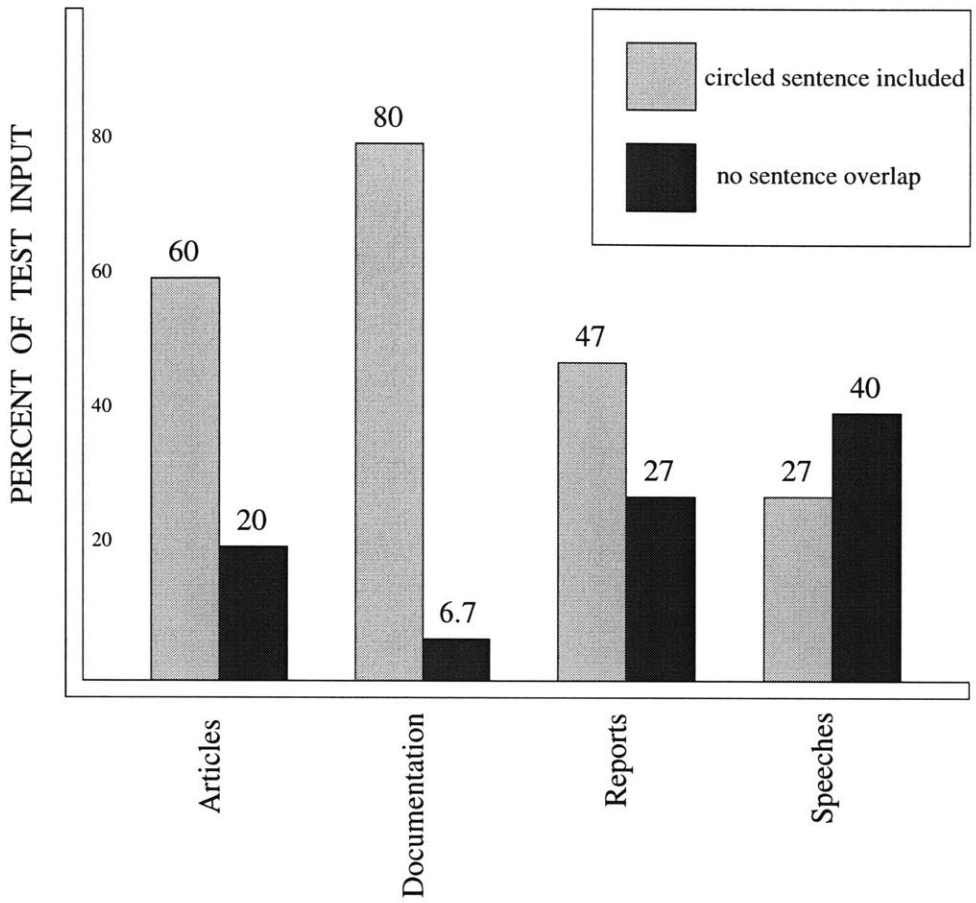


Figure 6-2: Percentages for Input Types

there was no overlap between the AutoExtractor summary and the sentences chosen by the human.

Newspaper Articles

The newspaper article was the second best summarized document type, with an average summary score of .706. This number is probably due to the fact that most summarizing sentences in an article are primarily located in the first two paragraphs. This is a simple trend for the AutoExtractor to identify and take advantage of. Identifying a summarizing sentence that is located in the body of the article, however, was more rare.

Documentation Documents

The high scores of the documentation summaries were quite surprising. The average documentation score was .873, which was significantly higher than the speech or report average. This can be attributed primarily to the fact that the AutoExtractor often chose a circled sentence from the document. This, in turn, is probably a result of the fact that a documentation document almost always gives a good summarizing sentence in the very beginning of the document, followed by very few in the rest of the document. Therefore, it is easy for the AutoExtractor to identify the primary summarizing sentence.

Reports

The results of the report scoring were rather disappointing with an average of .453. This was significantly lower than both the article and documentation scores. I fear that the performance may have been better if the parser was more accurate. I suspect that the paragraph and sections breaks and headers were not always properly identified. In addition, the low scores may have been caused by the fact that the reports had more internal variation than the other document types. The reports varied widely in content and tone, from short informal discussions to long, technical reports.

Speeches

The speeches received the lowest average score with a .383. I suspect that this was due to the lack of structure in a verbal presentation. The human summarizers did not consistently choose sentences from any one section of the document. The sentences chosen did not contain any unique structural elements that would distinguish them from the rest of the document. This is probably the result of too many good summarizing sentences. In a short address, the president felt the need to use many catch phrases and sweeping statements. For this reason, the speech documents were liberally sprinkled with good summarizing sentences. This makes it very difficult for the human and AutoExtractor to agree on only a few best sentences.

6.4.3 Further Testing

There is a great deal of additional testing that would provide further insight into the accuracy of the AutoExtractor. Unfortunately, the lack of time and resources prevented me from performing these tests.

Comparing with Other Extractors

The data gained through the tests above is somewhat unilluminating, since the performance of the AutoExtractor was analyzed in isolation. To gain information about the AutoExtractor in comparison with other extractors, one could run the same input used above through the other extractors. These results could then be scored with the same process, and the performance of the two extractors could be compared directly.

Analyzing Scoring Accuracy

The results of these tests are somewhat suspect because there is no measurement of their consistency. There are several reasons that these scores may prove to be inconsistent. The sentences selected by the human are usually not the only sentences that can create an accurate document summary. Each person has a different opinion of what constitutes a good summary, and a summary that receives a good score when graded by one user might receive a bad score when graded by another [9]. In addition, some documents might contain so many good summarizing sentences that there is not a very large overlap in sentence selection. (I suspect this to be the case with the speech documents.) Alternatively, some documents may not contain enough summarizing sentences, so that the AutoExtractor and human are forced to pick almost randomly. (This I suspect to be the case with the documentation documents.)

One way to discover the amount of precision in the testing results is to have each document analyzed by several humans. If there is a significant amount of overlap, then the testing results are valid. If the humans consistently choose different sentences, then it would be clear that the summary scoring somewhat arbitrary, and should not be taken too seriously.

Analyzing the Document Templates

Different document templates were created and used for the four different test input types, but no analysis was made of their success or failure as extraction tools. To acquire information about this subject, one might run the same documents through the extractor using different templates. For example, the same inputs can be analyzed using one general template, and then analyzed using the template that fits their document type. If performance significantly improves, this would indicate that the templates are a positive addition to the AutoExtractor.

Chapter 7

Conclusions

I began the task of creating the AutoExtractor as a straightforward path to a well defined goal. As the project comes to a close, however, it has become clear to me that the AutoExtractor is still in a stage of infancy. The list of augmentation, testing, and analysis that remains to be done grows longer and longer even as I attempt to implement it.

Luckily, the goal of this project was not to create the perfect summarizer. Engineers much greater than myself have tried this and failed. Rather, my goal was to create a working summarizer that functioned well enough for its purpose, while continuing to broaden my own knowledge and experience in the field of Artificial Intelligence. I feel that I have accomplished this goal.

7.1 Principles Discovered

In the course of my work with the AutoExtractor, I have grown in my knowledge of artificial intelligence. The following sections will discuss some of the principles that I have discovered or realized anew.

7.1.1 Variety Causes Complexity

One of the more frustrating principles that I discovered is that the tasks that are the most intuitive to a human are often the most difficult to implement.

For example, one of the first pieces of code I attempted to write was a document parser. I had decided to write one myself so that it would be very flexible and conform to my exact specifications. I was impatient to finish this subgoal, and continue on to what I considered to be the "meat" of the project. I was surprised to discover that I had begun a formidable task. Since there are no limits to the input types to the extractor, there is also no limit in the variety of formats that the parser must be able to recognize. I found it very difficult to accurately separate the text into sections and paragraphs and correctly differentiate headings and titles from headers, footers, and lists. Parsing the documents, which is an almost effortless task for a human, became one of the most challenging aspects of the project.

As this little task grew to monumentous proportions, I realized that there are some things that a computer just is not suited to do. Specifically, it is very difficult to perform a task with a great deal of variation and noise. This is a principle that continued to come back and haunt me throughout the project.

7.1.2 The Importance of Knowledge

At the outset, I viewed this project as a simple piece of code that would be complete once written. I have since learned that the code itself is merely a structure upon which the real AutoExtractor must be built.

The power of the extractor is not in the mechanics of the code that was written, but in the information that is gained through running and testing the code. Humans utilize a bed of information about documents and syntax that is vast in comparison to what is encapsulated in, for instance, an AutoExtractor document template. To create a truly versatile and accurate extractor, the knowledge that a human has must somehow be collected and stored quickly and accessibly [8].

In the case of the AutoExtractor, a great deal of knowledge could be gained

through more extensive testing and/or the implementation of an automatic learning algorithm.

7.2 Future Work

Through my studies and through the helpful suggestions of others, it has become apparent to me that there are many ways that the AutoExtractor could be augmented. Although it is not completely clear that these augmentations would have a positive effect on the system as a whole, they are at least worthy of experimentation.

7.2.1 Improving Upon Word Occurrence Statistics

One of the techniques used in the AutoExtractor to determine whether a sentence should be extracted is the similarity in word occurrence statistics between the sentence and the document (see section 4.2.1.) This technique is widely utilized both in the task of extracting and for many other tasks. Therefore, there are many permutations and additions to this technique that may improve upon the current accuracy of the AutoExtractor. A few of these improvements are listed below.

Matching Summary and Document Statistics

The test used to determine whether a sentence should be extracted is the similarity in word occurrence between the sentence and the document as a whole. In other words, this technique attempts to make each individual sentence in the summary similar in word content to the document. There is another, more complex way of extracting sentences using occurrence statistics. Instead of matching each individual sentence with the document, this other technique attempts to match the overall summary statistics with that of the document.

The implementation of this technique is straightforward. The first sentence is extracted as usual. The second sentence, however, is not simply compared to the document word occurrence statistics. Instead, the second sentence is selected by how well its word occurrence, when combined with the first sentence, matches with the

total document statistics. The third sentence is selected by how well its word occurrence, when combined with the first two sentences, matches that of the document. This process continues until the total number of sentences to be extracted have been chosen.

The advantage of this this technique is that the summary created is more likely to contain a wide variety of important topics than if the sentences were all chosen based upon the most prevalent items in the text. For example, the majority of the document might deal with one topic, but a significant portion might focus on a a different issue. If the sentences chosen are always required to match with the document as a whole, they will always be related to the most common topic in the document. They might be very repetitive, and they may not touch upon other minor, yet significant, topics in the document. When sentences are compared as an addition to the existing summary instead of independently, however, a topic that has thus far been unrepresented in the summary may be included to satisfy that portion of the document's word occurrence statistics.

The primary reason that this technique has not been implemented in the AutoExtractor is that I suspect the effect of the word occurrence aspect of the extractor is too insignifcant to cause the problems mentioned above. I have found, in the process of developing the AutoExtractor, that the word occurrence statistics have a relatively small influence on the selection of a sentence for extraction. Therefore, the slight difference in word occurrence similarity between a major topic and a minor topic in the document would not have a significant impact on the total score for that sentence.

While I could not imagine why using this improved word occurrence technique would have a negative impact on the performance of the system, it would require added complexity and computation. The accuracy of the document summary must increase significantly to justify the additional work.

Word Weighting

One of the more difficult aspects of word occurrence statistics is selecting which words are more significant that others. The key words in the document must be given high

importance, and other random words that are unrelated to the topic should have a low impact on the decision making process. Some systems use extensive semantic analysis to determine the important noun phrases used in each sentence [1]. Then those words are given a higher weight in the word array. This technique, however, is very computationally intensive, and usually does not result in significant improvements in accuracy [4].

In addition to putting more emphasis on individual words, it is also common to weight certain sections of the document that are likely to contain significant information. For instance, the first few paragraphs of a newspaper article almost always contain all vital information in the article. Therefore, the words in that portion of the article can be given a larger weight during the word occurrence calculations.

Finally, a related but different technique involves doing analysis on the entire body of documents in addition to the document in question. Words that occur frequently in the body of documents are given less significance while words that are unique to the present document are weighted heavily. This is a way of determining which words in the document are vital specifically to the document.

Taking Roots to Categorize Words

Another common way to make word occurrence techniques more accurate is to use a program to take the root of any word that is not already in root form. Therefore, the the program will acknowledge that two words that do not look exactly the same are nevertheless referring to the same concept: for instance, both "routing" and "router" would be reduced to the same root word, "rout". This technique does occasionally have undesirable effects, however. For instance, some words may appear to be in a reducible form to the program, but are not actually reducible. For instance, the last name "Carter" would be reduced to the root word "cart" [11]. For simplicity's sake, no root words are taken in this implementation. It would be possible to add this feature in the future to see if it would have a significant impact on performance.

7.2.2 Adding Additional Sentence Structure Factors

There are several aspects of sentence structure that the AutoExtractor does not process at this time. Some of these, however, might be very important in determining whether a sentence should be extracted. For example, a sentence that contains several clauses may be more likely to be a good summarizing sentence in some situations. While the parser does not detect information like commas or the word "and", these would be simple modifications that may greatly increase the ability of the extractor with certain input types.

7.2.3 Adding Negative Keywords and Phrases

One important aspect of the AutoExtractor is the detection of key words or phrases that may indicate the presence of a good summarizing sentence. In the same respect, it may be just as helpful to search for negative words or phrases that indicate that a sentence is *not* a good sentence to extract. In fact, some studies show that the negative keyphrase method can eliminate as much as 90 percent of the sentences of the document [2].

This would also be a simple augmentation to add to the AutoExtractor. It would require the addition of a new sentence characteristic that is assigned a higher value the *fewer* negative key phrases the sentence contains.

7.2.4 Altering Templates Through Automatic Learning Algorithms

At present, the AutoExtractor templates are created by hand. The user must use his or her best judgement to assign characteristic weights and value scores to conform to a certain type of desired input and output. While a user can assign these values with an often surprising amount of success, it is effectively impossible for a person to guess the exact combination of numbers that would produce the best possible output for that document type.

For this reason, altering the templates using an automatic learning algorithm

would be a very helpful addition to the AutoExtractor. Instead of assigning the template numbers, the program would be given training documents until it automatically converged on the best number combination.

In order to use learning algorithms, however, there must be a standard against which to compare the program output. If there is no way to determine how "good" a summary is, there is no feedback mechanism that the program can use to improve its performance. Unfortunately, determining the goodness of a summary is not a trivial task. There are many possible combinations of sentences that can form a good summary, and there are many different types of summaries. Evaluating the goodness of a summary is an extremely subjective task. This task must be performed automatically, however, in order for the learning algorithm to receive the volume of training data it needs.

An Automatic Learning Implementation

A related project, which used a technique very similar to the one discussed above, generated its feedback by analyzing technical reports that already contained abstracts. If the summarizer returned a sentence that was similar enough to a sentence in the abstract, that sentence was said to be a good choice.

Automatic Evaluation Disadvantages

There are a few disadvantages to this procedure. First, the correlation between the contents of the author-created abstract and the quality of extracted sentences is not concrete. Simply because a sentence is not included in the abstract does not mean that it is not a good summarizing sentence. An already difficult learning process would then be further complicated by the false evaluation of extraction results.

Secondly, the comparison of the abstract and an extracted sentence is a somewhat complex. It is possible to simply look for a sentence in the abstract that is identical to the extracted sentence, but sentences in the abstract are seldom exactly identical to sentences in the document. They are frequently either combinations or segments of the original sentences. In order to evaluate the extracted sentence with any degree

of accuracy, the comparison method must be capable of recognizing sentences that are only a part or partially contained in an abstract sentence. This is a delicate task which is somewhat subjective.

Finally, in order for this technique to be possible, there must be a testing set available with an abstract (or the equivalent) for each document. This would clearly not always be the case. For instance, in the testing set that was described in the previous chapter, there would have been no way to find a testing set for the article, documentation, or transcribed speech documents.

As inconvenient and limiting as the this automatic way of determining the quality of an extracted sentence may be, it is still preferable to the alternative. Instead, a human operator would have to hand-evaluate the fitness of the extracted sentences. A prohibitively large amount of manual labor would be necessary to process the volume of test documents needed to train the system.

Appendix A

Testing Instructions

Instructions:

You have been given four documents. Your job is to choose sentences out of these documents that you think are good summarizing sentences. This means that they would be good to include in an abstract about that document. You cannot choose parts of a sentence; you must choose the entire sentence. Please do not choose any titles or headings. Use your best judgement.

There are two numbers written on the top of each document with a slash between them: x/y. Look at the number on the left (x). CIRCLE that many sentences that you think are the BEST summarizing sentences in the document. Now look at the number on right (y). UNDERLINE that many more sentences that are also good summarizing sentences (but not as good as the circled ones). When you are finished, you should have x sentences circled and y sentences underlined. (Do not underline sentences that you have already circled.)

NOTE: Some of the documents are rather long; you do NOT have to read and understand the entire document! Feel free to skim things that you are pretty sure don't contain any significant sentences. Reading and marking a long paper should take no longer than 5-10 minutes.

Please return these to me by FRIDAY MAY 14th. You can either slip them under the door of my office (NE43-832), send them to me through interdepartmental mail (Irene Wilson, Ashdown #402A), or give them to

me directly.

Thank you thank you thank you!

Appendix B

Code

B.1 Summ.lisp

```
;; Irene Wilson
;; AI lab
;; Knowledge-Based Collaboration Project
;; created: 7-22-98  IW
;; altered:  2-4-99  IW
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Summ.lisp
;;   This file contains all the master calls for the document summarization;
;;   all functions are accessed (directly or indirectly) from this file.



(setq VERBOSE 'nil)
(setq VERBOSE-RES t)
(setq COUNT t)

;; This is the main function that runs the auto-summarization program.
(defun auto-sum (file-name &key numb-sents percent-sents user-doc-type)
  (let*
    ((percent-sentences (or percent-sents 5))

     ;; read in file (found in parse.lisp)
     (file (read-file file-name))

     ;; capture all information of the file into a bunch of objects
     ;; found in parse.lisp
     (doc-structure (parse-string file))

     ;; create a list containing the characteristics of each sentence, consed
     ;;   with the characteristics of the document
     (doc-and-sent-chars (get-doc-and-sent-chars doc-structure))

     ;; classify the document type based on the document characteristics
     (doc-type (or (intern (string-upcase user-doc-type))
                   (get-doc-type (car doc-and-sent-chars))))
```

```
     ;;  using the formula for the document type, calculate the fitness of
     ;;    each sentence as a topic sentence
     (sentence-ratings (make-sentence-rankings (cdr doc-and-sent-chars)
                                               (get-template doc-type))))

    ;; return the n percent best topic sentences from the document
    (get-best-sentences sentence-ratings
                                doc-and-sent-chars
                                percent-sentences
                                numb-sents)))



;; This function will analyze the characteristics of the document to determine
;;   the document type.  At the moment, we will just assume it is a report.
(defun get-doc-type (doc-characteristics)
  'doc)


;;returns a vector of the best rated sentences with their ratings
(defun get-best-sentences (sentence-ratings
                           doc-and-sent-chars
                           percent-sentences
                           numb-sents)
  ;; sort by highest rating
  (let ((sorted-stats (sort
                       (rank-vector sentence-ratings)
                       #'(lambda (sent1 sent2) (> (sent-rating (cdr sent1))
                                                  (sent-rating (cdr sent2))))))
        (best-sentences (make-array 0 :adjustable t :fill-pointer t))
        (number-sentences
         (if numb-sents
           numb-sents
           (* percent-sentences (/ (num-sentences (car doc-and-sent-chars))
                                   100)))))
    (when COUNT
      (let ((num-sent (num-sentences (car doc-and-sent-chars))))
        (format t "~A~A~A::~A::~A~A" #\Return #\Return num-sent
                (- (log (- (log num-sent 2) 2) 2) 1)
                (- (log num-sent 2) 2)
                #\Return)))


    ;; assemble vector of best sentences + ratings
    (loop for x from 0 to (- number-sentences 1)
          do (let* ((text (sentence-text (pointer-to-sent
                                          (aref (cdr doc-and-sent-chars)
                                                (car (svref sorted-stats x))))))
                    (sent-rating-info (cdr (svref sorted-stats x)))
                    (rating (sent-rating sent-rating-info)))
               (vector-push-extend (cons text rating) best-sentences 10)
               (when VERBOSE-RES
                 (format t "~A~A" #\Return #\Return)
                 (format t "~A:  ~A" (+ x 1) text)
                 (loop for y from 0 to (- (length (chars-used sent-rating-info)) 1)
                       do (let ((char-assgn-info
                                 (aref (chars-used sent-rating-info) y))
                                (char-val-info
                                 (aref (val-of-chars-used sent-rating-info) y)))
                            (format t "~A   ~A: ~A  ==> ~A x ~A = ~A"
                                    #\Return
                                    (car char-assgn-info)
                                    (cadr char-assgn-info)
                                    (car char-val-info)
                                    (cadr char-val-info)
                                    (caddr char-val-info)))))))

    best-sentences))
```

```
;; appends an index to each sentence-rating so we don't lose the original
;; order after sorting
(defun rank-vector (vector)
  (let ((rank -1))
    (flet ((rank-element (element)
             (setq rank (+ rank 1))
             (cons rank element)))
      (map 'vector #'rank-element vector))))
```

B.2 Parse.lisp

```
;; Irene Wilson
;; AI lab
;; Knowledge-Based Collaboration Project
;; created: 7-23-98  IW
;; altered: 2-11-99  IW
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Parse.lisp
;;   This file contains a library of functions that will take a file
;;   and parse its contents into several objects that contain all
;;   interesting information about that document


(setq kill-returns 'nil)
(setq indents 'nil)

;; This function takes a file name and returns a string of text that
;;   represents the contents of the file.  Also, if the format of the file needs
;;   to be changed, it changes it to an acceptible format.
(defun read-file (filename)
  (let ((prev-line-punc 'nil)
        (file (concatenate 'string "Grant-HD1:Users:merri:testing:documentation:formatted:" filename)))
    (labels ((reformat-with-returns (line string-stream)
               (cond ((not (position-if #'not-space line))
                      (write-line "" string-stream)
                      (if prev-line-punc
                        (write-line "" string-stream)))
                     ((and prev-line-punc (tab-p line))
                      (write-line "" string-stream)
                      (write-string (concatenate 'string line " ") string-stream))
                     (t
                      (write-string (concatenate 'string line " ") string-stream)
                      (if (and (not indents) (< (length line) 55))
                        (write-line "" string-stream)))))

             (ends-with-punct (line)
               (let ((ending-char (position-if-not #'whitespace-p line :from-end t)))
                 (if (and ending-char (find (aref line ending-char) '(#\. #\? #\!)))
                   (setq prev-line-punc t)
                   (setq prev-line-punc 'nil)))))

      (with-open-file (file-stream file)
        (with-output-to-string (string-stream)
          (loop for line = (read-line file-stream nil nil)
                while line
                do (progn (cond (kill-returns
```

```
                          (reformat-with-returns line string-stream))
                         (t
                          (write-line line string-stream)))
                   (ends-with-punct line))))))))


(defun tab-p (line)
  (or (eq (aref line 0) #\Tab)
      (and (eq (aref line 0) #\Space) (eq (aref line 1) #\Space))))


;; This function takes a string and parses it into a full document
;;   object, including sections, titles, paragraphs, and sentences.
;;   It's not particularly intelligent.
(defun parse-string (text)
  (let (;; the big overall object that represents the entire document
        ;; holds all other objects as memebers of arrays
        (my-document (make-document))
        ;; This is the index into the text string where the current word begins
        (word-start 0)
        ;; This is the index where the current word ends
        (word-end 0)
        ;; This is the index where the next item ends
        (stop-loc 0)
        ;; This is the index where the next item begins
        (next-item (or (my-position-if #'(lambda (x) (not (eq x #\Return))) text)
                       (- (length text) 1)))
        ;; This is the index where the current sentence begins
        (sentence-start 0))
    (labels ((parse-loop ()
               (loop
                 ;; while we haven't reached the end of our text string:
                 while next-item
                 ;; look at the next item in the text string and decide what it is.
                 do (case (next-chunk)
                      (word (parse-word))
                      (sentence (end-sentence))
                      (sect-title (add-sect-title))
                      (par-title (add-par-title))
                      (paragraph (finish-paragraph))
                      (section (finish-section))
                      (nothing (continue-parse))
                      (fragment (fragment-sentence))
                      (done (return))
                      (otherwise (format t "nextchunk returned illegal value") (break))))
               ;; ???
               (clean-up my-document))

             (next-chunk ()
               ;; Here I am checking to see if the next item in the text is a return.
               ;; If so, this can represent a new paragraph, a new section, a title,
               ;; or an error.
               (cond
                ((eq (aref text next-item) #\Return)
                 ;; check to see if the return was in the middle of a sentence or not.
                 (cond
                  ((beginning-of-sentence-p my-document)
                   ;; One return is a new paragraph; more than one a new section.
                   (setq stop-loc (my-position-if #'not-space text
                                                  :start (+ next-item 1)
                                                  :out-of-bounds-check 't))
                   ;; if text is finished, return
                   (if (not stop-loc)
                     'done
                     (if (eq (aref text stop-loc) #\Return)
                       'section
                       'paragraph)))
```

```
      ;; If a return is found in the middle of a sentence, this could
      ;;  either represent a title or a sentence fragement.
     (t (if (good-sect-title-loc-p my-document)
          'sect-title
          (if (good-par-title-loc-p my-document)
            'par-title
            'fragment)))))
   ;; Next item is not a return; let's see what it is!  stop-loc holds
   ;;   the next character of interest in the string.
   (t (setq stop-loc
            (position-if #'(lambda (x) (find x '(#\Space #\. #\? #\! #\Return)))
                         text
                         :start next-item))
      (cond
       (stop-loc
        (let ((stopper (aref text stop-loc)))
          ;; analyze next significant charcter
          (case stopper
            (#\Space 'word)
            ;; a period indicates the end of the sentence unless it's a
            ;;   decimal... abbreviations will screw up parser.
            (#\. (if (end-of-sent-period-p)
                   'sentence
                   'nothing))
            ((#\? #\!) 'sentence)
            (#\Return (if (good-sect-title-loc-p my-document)
                        'sect-title
                        (if (good-par-title-loc-p my-document)
                          'par-title
                          'fragment)))
            (otherwise (format t "bad stopper")))))
       ;; we have reached the end of the document without finishing the last
       ;;   sentence.
       (t
        (setq stop-loc (length text))
        'fragment)))))

;; When each of these situations has been detected, move around the
;; necessary text pointers and call the appropriate method of document
;; object.

(parse-word ()
  (setq word-end stop-loc)
  (setq next-item (position-if #'not-space text :start stop-loc))
  (add-word my-document (get-word))
  (setq word-start next-item))

(end-sentence ()
  (setq word-end stop-loc)
  (setq next-item
        (my-position-if #'(lambda (x) (not (find  x '(#\Space #\. #\? #\!))))
                    text
                    :start (+ stop-loc 1)
                    :out-of-bounds-check 't))
  (add-word my-document (get-word))
  (end-sent-doc my-document (subseq text sentence-start next-item))
  (setq word-start next-item)
  (setq sentence-start next-item))

(finish-paragraph ()
  (setq next-item stop-loc)
  (setq word-start next-item)
  (setq sentence-start next-item)
  (end-par-doc my-document))

(finish-section ()
  (setq next-item (position-if #'(lambda (x) (not (find x '(#\Space #\Return))))
                               text
```

```
                              :start stop-loc))
  (setq word-start next-item)
  (setq sentence-start next-item)
  (end-sect-doc my-document))

(fragment-sentence ()
  ;; Pick up the last word of the fragment sentence (if needed)
  (cond
   ((< next-item stop-loc)
    (setq word-end stop-loc)
    (add-word my-document (get-word))
    (setq next-item (my-position-if #'not-space text
                                    :start (+ stop-loc 1)
                                    :out-of-bounds-check 't)))
   (t
    (setq next-item (my-position-if #'not-space text
                                    :start (+ next-item 1)
                                    :out-of-bounds-check 't))))
  (fragment-sent-doc my-document (subseq text sentence-start next-item))
  (setq word-start next-item)
  (setq sentence-start next-item))

(continue-parse ()
  (setq next-item (+ next-item 1)))

(add-par-title ()
  (add-title 'par))

(add-sect-title ()
  (add-title 'sect))

(add-title (unit)
  ;; clean up last word of title if necessary
  (when (< next-item stop-loc)
    (setq word-end stop-loc)
    (add-word my-document (get-word))
    (setq next-item stop-loc))
  (setq next-item
        (my-position-if #'(lambda (x) (not (find x '(#\Space #\Return))))
                        text
                        :start (+ next-item 1)
                        :out-of-bounds-check 't))
  (if (eq unit 'sect)
    (add-sect-title-doc my-document (subseq text sentence-start next-item))
    (add-par-title-doc my-document (subseq text sentence-start next-item)))
  (setq word-start next-item)
  (setq sentence-start next-item))

(end-of-sent-period-p ()
  (not (or (digit-char-p (aref text (+ stop-loc 1)))
           (abbreviation-p
            (subseq text
                    (or (+ 1 (position-if #'(lambda (x) (find x '(#\Return
                                                                  #\Space
                                                                  #\. #\Tab)))
                                          text
                                          :end stop-loc
                                          :from-end t))
                        0)
                    stop-loc)))))

(abbreviation-p (str)
  (find str '("Mr" "Ms" "Mrs" "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L"
              "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z" "Inc"
              "St" "Ltd" "Corp" "Kan" "Mo" "1" "2" "3" "4" "5" "6" "7" "8" "9"
              "0") :test #'equal))

(get-word ()
```

```
            (string-trim '(#\, #\: #\" #\; #\- #\) #\()
                         (subseq text word-start word-end))))


      (parse-loop)
      my-document)))
```

B.3 Objects.lisp

```
;; Irene Wilson
;; AI lab
;; Knowledge-Based Collaboration Project
;; created:  7-27-98  IW
;; altered: 11-18-98  IW
;;           1-4-99  IW
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Objects.lisp
;;   This file contains all the object definitions and member functions
;;   of the automatic document summarization project.  All of the significant
;;   data about the document is stored in these objects.  Later, these objects
;;   will be analyzed to see which sentence objects are the most likely topic
;;   sentences.


;; A document object contains all interesting information about a body of text.
(defclass document ()
  (;; This vector stores all of the significant words in the document
   (hash-strings
    :accessor hash-strings
    :initform (make-array 0 :adjustable t :fill-pointer t :element-type 'string))
   ;; This vector stores how many times the word in that location of the
   ;;   hash-strings vector appears in the document
   (word-vector
    :accessor word-vector
    :initform (make-array 0 :adjustable t :fill-pointer t :element-type 'integer))
   ;; This hashtable takes in a word and returns its location in the word-vector.
   (keyword-hash
    :accessor keyword-hash
    :initform (make-hash-table :size 300 :rehash-size 100 :test 'equalp))
   ;; ??? The title of the document
   (heading
    :accessor heading
    :initform (make-array 0 :adjustable t :fill-pointer t :element-type 'sentence))
   ;; This is a list of all the keyphrases that will be searched for in the document
   (keyphrase-list :accessor keyphrase-list :initform (get-phrases "keyphrases"))
   ;; This is a list of keyphrases that have been found in the document.
   (keyphrase-status :accessor keyphrase-status)
   ;; This is an array that points to all the sections in the document
   (sections :accessor sections :initform (make-sect-array))
   ;; contains total number of sentences and fragments
   (num-of-sentences :accessor num-of-sentences :initform 0)
   ;; contians total number of words
   (num-of-words :accessor num-of-words :initform 0)))

;; This is a section object; represents a section of the document
(defclass section ()
  ((;; This is a pointer to all of the paragraph objects in the section
```

```
    paragraphs
    :accessor paragraphs
    :initform (make-par-array))
   ;; if the section has a title, it will be stored here
   (title :accessor title :initform 'nil)))

;; This is a paragraph object; represents a paragraph in the document
(defclass paragraph ()
  ((;; this is a pointer to all of the sentnece objects in the section
    sentences
    :accessor sentences
    :initform (make-sent-array))
   (title :accessor title :initform 'nil)))

;;  This is a sentence object; represents a sentence in the document
(defclass sentence ()
  (;; this string holds the exact text of the sentence.
   (sentence-text :accessor sentence-text)
   ;; this is the the number of the sentence in the document
   (sentence-num :accessor sentence-num)
   ;; this is true if this is a fragment sentence; false otherwise
   (fragment :accessor fragment-p :initform 'nil)
   ;; this vector has a location for each significant word in the sentence;
   ;;   the value of a location is the number of times that word appears
   (word-vector :accessor word-vector)
   ;; this contains the number and type of keyphrases in the sentence
   (keyphrases :accessor keyphrases)
   ;; this value is the total number of words in the sentence
   (num-of-words :accessor num-of-words :initform 0)))

;; This function creates and initializes a document object
(defun make-document ()
  (let ((doc (make-instance 'document)))
    ;; initializes the first sentence in the document
    (initialize-sent doc (get-curr-sent doc))
    ;; creates a location in the keyphrase-status vector for every element
    ;;   in the keyphrase-list vector.  This is so we can store how many
    ;;   times each keyphrase appears.
    (setf (keyphrase-status doc) (make-array (length (keyphrase-list doc))
                                             :initial-element 0
                                             :element-type 'integer))
    doc))

;; This function creates and initializes a section array; called when document
;;   first created
(defun make-sect-array ()
  (let ((sect-array (make-array 0
                                :adjustable t
                                :fill-pointer t
                                :element-type 'section)))
    (vector-push-extend (make-instance 'section) sect-array 5)
    sect-array))
;; This function creates and initializes a paragraph array; called when a section
;;   is created
(defun make-par-array ()
  (let ((par-array (make-array 0
                               :adjustable t
                               :fill-pointer t
                               :element-type 'paragraph)))
    (vector-push-extend (make-instance 'paragraph) par-array 5)
    par-array))

;; This function creates and initializes a sentence array; called when a paragraph
;;   is created.
(defun make-sent-array ()
  (let ((sent-array (make-array 0
                                :adjustable t
                                :fill-pointer t
```

```
                                :element-type 'sentence)))
    (vector-push-extend (make-instance 'sentence) sent-array 8)
    sent-array))

;; This function initializes a sentence object; sets the size of the keyphrases
;;   and word-vector object based upon the current document info.
(defmethod initialize-sent ((doc document) sent)
  (setf (keyphrases sent) (make-array (length (keyphrase-list doc))
                                      :element-type 'integer
                                      :initial-element 0))
  (setf (word-vector sent)
        (make-array (length (word-vector doc))
                    :adjustable t
                    :fill-pointer t
                    :element-type 'integer
                    :initial-element 0))
  sent)

;; These are for easy reference to all the current parts of the document
(defmethod get-curr-sent ((doc document))
  (let ((curr-par (get-curr-par doc)))
    (aref (sentences curr-par) (- (num-of-sentences curr-par) 1))))

(defmethod get-curr-par ((doc document))
  (let ((curr-sect (get-curr-sect doc)))
    (get-curr-par curr-sect)))

(defmethod get-curr-par ((sect section))
  (aref (paragraphs sect) (- (num-of-paragraphs sect) 1)))

(defmethod get-curr-sect ((doc document))
  (aref (sections doc) (- (num-of-sections doc) 1)))

(defmethod get-curr-sent ((sect section))
  (get-curr-sent (aref (paragraphs sect) (- (num-of-paragraphs sect) 1))))

(defmethod get-curr-sent ((par paragraph))
  (aref (sentences par) (- (num-of-sentences par) 1)))

(defmethod get-curr-sent ((par paragraph))
  (aref (sentences par) (- (num-of-sentences par) 1)))

;; These are for adding to the current document structure
(defmethod add-word ((doc document) word)
  (setf (num-of-words doc) (+ (num-of-words doc) 1))
  (add-word-sent (get-curr-sent doc) doc word))

(defmethod end-sent-doc ((doc document) sentence-text)
  (let ((curr-sent (get-curr-sent doc)))
    (setf (sentence-text curr-sent) sentence-text)
    (set-vector (keyphrase-status doc) 0)
    (setf (num-of-sentences doc) (+ (num-of-sentences doc) 1))
    (setf (sentence-num curr-sent) (num-of-sentences doc))
    (end-sent-par (get-curr-par doc) doc sentence-text)))

(defmethod end-par-doc ((doc document))
  (end-par-sect (get-curr-sect doc) doc))

(defmethod end-sect-doc ((doc document))
  ;; pop off the new, unused paragraph from the old section
  (vector-pop (sentences (get-curr-par doc)))
  (vector-push-extend (make-instance 'section) (sections doc) 10)
  (initialize-sent doc (get-curr-sent doc))
  (print "ended a section"))

(defmethod fragment-sent-doc ((doc document) sentence-text)
  (let ((curr-sent (get-curr-sent doc)))
    (setf (fragment-p curr-sent) 't)
```

```
    (print "(fragment)")
    (end-sent-doc doc sentence-text)))

;; add a title to the next section in the document
(defmethod add-sect-title-doc ((doc document) sentence-text)
  ;; get the next sentence object in the document
  (let ((sent (get-curr-sent doc)))
    (setf (sentence-text sent) sentence-text)
    ;; if there is only 1 section in the document so far, this "title" will
    ;; be saved as part of a "document heading" chunk.
    (if (eq (num-of-sections doc) 1)
      (vector-push-extend sent (heading doc) 5)
      (add-sect-title (get-curr-sect doc) doc))
    (when VERBOSE
      (print "added a section title")
      (format t "~A~\%" sentence-text))
    (abort-sent (get-curr-par doc) doc)))

;; add a title to the next paragraph in the document
(defmethod add-par-title-doc ((doc document) sentence-text)
  ;; get the next sentence object in the document
  (setf (sentence-text (get-curr-sent doc)) sentence-text)
  (add-par-title (get-curr-par doc) doc)
  (when VERBOSE
    (print "added a paragraph title")
    (format t "~A~\%" sentence-text))
  (abort-sent (get-curr-par doc) doc))

(defmethod add-sect-title ((sect section) (doc document))
  (setf (title sect) (get-curr-sent sect)))

(defmethod add-par-title ((par paragraph) (doc document))
  (setf (title par) (get-curr-sent par)))

(defmethod end-par-sect ((sect section) (doc document))
  ;; pop off the new, unused sentence from the old paragraph
  (vector-pop (sentences (get-curr-par sect)))
  ;; add on the new paragraph
  (vector-push-extend (make-instance 'paragraph) (paragraphs sect) 10)
  (initialize-sent doc (get-curr-sent sect))
  (print "ended a paragraph"))

(defmethod end-sent-par ((par paragraph) doc sentence-text)
  (start-sent par doc)
  (when VERBOSE
    (print "ended a sentence")
    (format t "~A~\%" sentence-text)))


(defmethod start-sent ((par paragraph) (doc document))
  (vector-push-extend (initialize-sent doc (make-instance 'sentence))
                      (sentences par)
                      20))

(defmethod add-word-sent ((sent sentence) doc word)
  (update-keyphrases doc sent word)
  (setf (num-of-words sent) (+ 1 (num-of-words sent)))
  (add-word-stats doc sent word))
;;  (format t "~A~\%" word))

(defmethod abort-sent ((par paragraph) (doc document))
  (vector-pop (sentences par))
  (start-sent par doc))

;; looks at newly added word to see if it brings us closer to identifying
;;   a keyphrase
(defmethod update-keyphrases ((doc document) (sent sentence) wrd)
  ;; for each keyphrase
```

```
  (loop for x from 0 to (- (length (keyphrase-list doc)) 1)
        do (let (;; this is the actual phrase
                 (phrase (aref (keyphrase-list doc) x))
                 ;; gets the number of words in the phrase already identified
                 (stat (aref (keyphrase-status doc) x)))
             ;; if the next word in the document equals the next word in the
             ;;   phrase
             (if (equalp wrd (aref phrase stat))
               ;; check to see if you have completed the phrase
               (cond ((eq stat (- (length phrase) 1))
                      (setf (aref (keyphrases sent) x)
                            (+ (aref (keyphrases sent) x) 1))
                      (setf (aref (keyphrase-status doc) x) 0))
                     (t (setf (aref (keyphrase-status doc) x) (+ 1 stat))))
               ;; phrase was not found.  reset its status
               (setf (aref (keyphrase-status doc) x) 0)))))

;; if new word is a "keyword", record its use in the document and sentence
;;   keyword statistics vectors
(defmethod add-word-stats ((doc document) (sent sentence) wrd)
  (when (keyword-p wrd)
    (let ((hash-val (gethash wrd (keyword-hash doc))))
      ;; if the word has not previously been found in the document
      (cond ((eq hash-val NIL)
             ;; add word to keyword vector
             (setf (gethash wrd (keyword-hash doc)) (hash-size doc))
             ;; add word to both document and sentence keyword vectors
             (vector-push-extend 1 (word-vector sent) 10)
             (vector-push-extend 1 (word-vector doc) 10)
             (vector-push-extend wrd (hash-strings doc)))
            ;; else keyword is already in document; just increment the occurances
            (t
             (setf (aref (word-vector sent) hash-val)
                   (+ 1 (aref (word-vector sent) hash-val)))
             (setf (aref (word-vector doc) hash-val)
                   (+ 1 (aref (word-vector doc) hash-val)))))))))

;; check to see if we are currently beginning a new sentence
(defmethod beginning-of-sentence-p ((doc document))
  (if (eq (num-of-words (get-curr-sent doc)) 0) 't nil))

;; check to see if this is a possible section title location
(defmethod good-sect-title-loc-p ((doc document))
  ;; we are at a good title location if there is no title already, and we are
  ;;  not in the middle of a section or a paragraph
  (if (and (not (title (get-curr-sect doc)))
           (eq (length (paragraphs (get-curr-sect doc))) 1)
           (eq (length (sentences (get-curr-par doc))) 1))
    't
    nil))


;; check to see if this is a possible paragraph title location
(defmethod good-par-title-loc-p ((doc document))
  ;; we are at a good title location if there is no title already, and we are
  ;;  not in the middle of a paragraph
  (if (and (not (title (get-curr-par doc)))
           (not (eq (length (paragraphs (get-curr-sect doc))) 1))
           (eq (length (sentences (get-curr-par doc))) 1))
    't
    nil))


;; clean up document structure
(defmethod clean-up ((doc document))
  ;; get rid of all extra sections, paragraphs, and sentences on the end of the
  ;;   document structure.
  (vector-pop (sentences (get-curr-par doc)))
```

```
  (when (eq (length (sentences (get-curr-par doc))) 0)
    (vector-pop (paragraphs (get-curr-sect doc)))
    (when (eq (length (paragraphs (get-curr-sect doc))) 0)
      (vector-pop (sections doc)))))

;; count number of unique keywords in the document
(defmethod hash-size ((doc document))
  (hash-table-count (keyword-hash doc)))

;; some simple utilities

(defmethod num-of-sentences ((par paragraph))
  (length (sentences par)))

(defmethod num-of-paragraphs ((sect section))
  (length (paragraphs sect)))

(defmethod num-of-sections ((doc document))
  (length (sections doc)))

;; These are for printing the document structure.

(defmethod print-doc ((doc document))
  (print "doc")
  (map 'vector #'print-sect (sections doc)))

(defmethod print-sect ((sect section))
  (print " sect")
  (map 'vector #'print-par (paragraphs sect)))

(defmethod print-par ((par paragraph))
  (print "  par")
  (map 'vector #'print-sent (sentences par)))

(defmethod print-sent ((sent sentence))
  (print "   sent"))
```

B.4 Char.lisp

```
;; Irene Wilson
;; AI lab
;; Knowledge-Based Collaboration Project
;; created: 8-10-98  IW
;; altered: 8-11-98  IW
;;          1-13-99  IW
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Char.lisp
;;   This file contains code that analyzes a document object to calculate
;;   and store all valuable information in document and sentence statistic
;;   objects.


(defclass document-stat ()
  ((document-structure :accessor document-structure)
   (ave-sent-doc-similarity :accessor ave-sent-doc-similarity :initform 0)
   (doc-sent-similarity-stand-dev
    :accessor doc-sent-similarity-stand-dev
    :initform 0)
```

```
   (ave-word-length :accessor ave-word-length)
   (ave-sent-length :accessor ave-sent-length)
   (number-of-titles :accessor number-of-titles)
   (num-compound-sents :accessor num-compound-sents)
   (num-sentences :accessor num-sentences)
   (title-content :accessor title-content)))

(defclass section-stat()
  ((section-loc :accessor section-loc)))

(defclass paragraph-stat()
  ((paragraph-loc :accessor paragraph-loc)))

(defclass sentence-stat ()
  ((sent-char-list :accessor sent-char-list :initform (make-charact-list))
   (sent-cont-list :accessor sent-cont-list :initform (make-contain-list))
   (pointer-to-sent :accessor pointer-to-sent)
   (title-content :accessor title-content)))

;; creates and returns a characteristics object that contains all important
;; info about the document
(defun get-doc-and-sent-chars (doc-structure)
  (let ((doc-stat (make-instance 'document-stat))
        ;; contains a "characeristic object" for each sentence in the document
        (sent-chars-array (make-array (num-of-sentences doc-structure)
                                      :element-type 'sentence-stat))
        ;; this is a place to save information about the sentences that are
        ;; not needed in the final analysis
        (scratchpad (make-array (num-of-sentences doc-structure)))
        ;; load all the key-title-phrases to search for the titles
        (key-title-phrases (get-phrases "title-phrases"))
        ;; ???
        (chars-array-size 0)
        ;; this is for the title-contents info
        (curr-sect-index 0))

    (labels
      ;; assimilate all info and objects in this section of the document
      ((analyze-section (document index curr-sent-info)
         (let ((section (aref (sections document) index)))
           ;; save this for later use in title-contents
           (setq curr-sect-index index)
           ;; add all section information into curr-sent-info (this info will
           ;; be copied into all sent-char objects in this section)
           (add-sect-info document index curr-sent-info)
           ;; analyze each paragraph in the section
           (loop for y from 0 to (- (length (paragraphs section)) 1)
                 do (analyze-paragraph section y (copy-of curr-sent-info)))))

       ;; assimilate all info and objects in this paragraph of the document
       (analyze-paragraph (section index curr-sent-info)
         (let ((paragraph (aref (paragraphs section) index)))
           ;; add all paragraph info into curr-sent-info
           (add-par-info section index curr-sent-info)
           ;; analyze all sentences in the paragraph
           (loop for z from 0 to (- (length (sentences paragraph)) 1)
                 do (analyze-sentence paragraph z (copy-of curr-sent-info)))))

       ;; assimilate all info in this sentence into curr-sent-info
       (analyze-sentence (paragraph index curr-sent-info)
         (add-sent-info paragraph index curr-sent-info))

       ;; gets all info that can be gotten directly from the document
       (add-doc-info (doc)
         (setf (ave-sent-length doc-stat) (/ (num-of-words doc)
                                             (num-of-sentences doc)))
         (setf (num-sentences doc-stat) (num-of-sentences doc))
         ;; make empty array for specials titles found during sentence analysis
```

```
  (setf (title-content doc-stat) (make-array (length (sections doc))
                                             :initial-element 'nil)))

;; gets all info that can be gotten directly from the section and adds
;; to curr-sent
(add-sect-info (doc sect-number curr-sent-info)
  (let* ((curr-sect (aref (sections doc) sect-number))
         (key-title-phrase (find-keytitle (title curr-sect))))
    (case sect-number
      (0 (set-sent-char curr-sent-info 'section-loc 'first))
      (1 (set-sent-char curr-sent-info 'section-loc 'second))
      ((length (sections doc))
       (set-sent-char curr-sent-info 'section-loc 'last))
      (otherwise (set-sent-char curr-sent-info 'section-loc 'body)))
    (set-sent-char curr-sent-info 'title-content key-title-phrase)
    (when key-title-phrase
      (setf (aref (title-content doc-stat) sect-number)
            key-title-phrase))))

;; gets all info that can be gotten directly from the paragraph and adds
;; to curr-sent
(add-par-info (section par-number curr-sent-info)
  (let ((curr-par (aref (paragraphs section) par-number)))
    (case par-number
      (0 (set-sent-char curr-sent-info 'paragraph-loc 'first))
      (1 (set-sent-char curr-sent-info 'paragraph-loc 'second))
      ((length (paragraphs section))
       (set-sent-char curr-sent-info 'paragraph-loc 'last))
      (otherwise (set-sent-char curr-sent-info 'paragraph-loc 'body)))
    ;;
    (if (not ())
      (let ((key-title-phrase (find-keytitle (title curr-par))))
        (when key-title-phrase
          (set-sent-char curr-sent-info 'title-content key-title-phrase)
          (setf (aref (title-content doc-stat) curr-sect-index)
                key-title-phrase))))))

;; gets all info from the sentence and adds it to curr-sent
(add-sent-info (paragraph sent-number curr-sent-info)
  ;; get sentence location in paragraph
  (let ((sentence (aref (sentences paragraph) sent-number)))
    (case sent-number
      (0 (set-sent-char curr-sent-info 'sentence-loc 'first))
      (1 (set-sent-char curr-sent-info 'sentence-loc 'second))
      ((length (sentences paragraph))
       (set-sent-char curr-sent-info 'sentence-loc 'last))
      (otherwise (set-sent-char curr-sent-info 'sentence-loc 'body)))
    ;; save a pointer into the sentence object (eek!)
    (setf (pointer-to-sent curr-sent-info) sentence)
    ;; get (comparative) length of sentence
    (cond ((> (num-of-words sentence) (ave-sent-length doc-stat))
           (set-sent-char curr-sent-info 'sentence-length 'long))
          ((< (num-of-words sentence) 7)
           (set-sent-char curr-sent-info 'sentence-length 'very-short))
          (t
           (set-sent-char curr-sent-info 'sentence-length 'short)))
    ;; save the doc-sentence keyword similarity in the scratchpad
    (setf (aref scratchpad chars-array-size)
          (dot-product (word-vector doc-structure)
                       (word-vector sentence)))
    ;; add the current similarity to the total document similarity
    ;; (to later calculate average)
    (setf (ave-sent-doc-similarity doc-stat)
          (+ (ave-sent-doc-similarity doc-stat)
             (aref scratchpad chars-array-size)))
    ;; add presence of keyphrases
    (let ((sent-keyphrases (make-array 0 :adjustable t :fill-pointer t)))
      (loop
```

```
        for poss-phrase from 0 to (- (length (keyphrases sentence)) 1)
        do (when (not (= (aref (keyphrases sentence) poss-phrase) 0))
             (let ((phrase-symbols
                    (map 'list
                         #'(lambda (x) (intern (string-upcase x)))
                         (aref (keyphrase-list doc-structure)
                               poss-phrase))))
               (vector-push-extend phrase-symbols sent-keyphrases 2))))
      (set-sent-cont curr-sent-info 'keyphrase-content sent-keyphrases))
    ;; add the current sent-info to the array!
    (setf (aref sent-chars-array chars-array-size) curr-sent-info)
    (setf chars-array-size (+ chars-array-size 1))))

;;
(find-keytitle (title)
  (if title
    (find-phrase title key-title-phrases)
    'nil))

;; looks to see if there are any key-title-phrases in the title

(find-phrase (sent phrases)
  (let ((found-phrases (make-array 0 :adjustable t :fill-pointer t))
        (parsed-sent (parse-line (sentence-text sent)))
        (found-one 'nil))
    (loop
      for wrd from 0 to (- (length parsed-sent) 1)
      do (loop
           for phrase-index from 0 to (- (length phrases) 1)
           do (if (eq (aref parsed-sent wrd)
                      (aref (aref phrases phrase-index) 0))
                (let ((curr-phrase (aref phrases phrase-index)))
                  (unless (< (+ wrd (length curr-phrase))
                             (length parsed-sent))
                    (loop
                      for phrase-wrd from 0 to (length curr-phrase)
                      while (eq (aref parsed-sent (+ wrd phrase-wrd))
                                (aref curr-phrase phrase-wrd))
                      finally
                      (setq found-one t)
                      (let ((phrase-symbols
                             (map 'list #'(lambda (x)
                                            (intern (string-upcase x)))
                                  curr-phrase)))
                        (vector-push-extend phrase-symbols
                                            found-phrases
                                            2))))))))
    (if found-one
      found-phrases
      'nil)))


;; finish up calculations
(do-other-calculations ()
  ;; calculate average similarity of sentences to document
  (setf (ave-sent-doc-similarity doc-stat)
        (/ (ave-sent-doc-similarity doc-stat) (length scratchpad)))
  ;; calculate std. dev. of similarity of sentences to document
  (loop for x from 0 to (- (length scratchpad) 1)
        do (setf (doc-sent-similarity-stand-dev doc-stat)
                 (+ (doc-sent-similarity-stand-dev doc-stat)
                    (expt (- (aref scratchpad x)
                             (ave-sent-doc-similarity doc-stat))
                          2))))
  (setf (doc-sent-similarity-stand-dev doc-stat)
        (if (> (length scratchpad) 1)
          (/ (doc-sent-similarity-stand-dev doc-stat)
             (- (length scratchpad) 1))
```

```
                0))
          ;; get comparative assessment of similarity of sentences with doc.
          (loop for x from 0 to (- (length scratchpad) 1)
                do (set-sent-char (aref sent-chars-array x)
                                  'sent-doc-similarity
                                  (get-similarity (aref scratchpad x))
                                  (aref scratchpad x))))

        ;; get comparative assessment of the similarity by using the ave and
        ;; std. dev.
        (get-similarity (doc-sent-sim)
          (let ((difference (- doc-sent-sim (ave-sent-doc-similarity doc-stat))))
            (if (> difference 0)
              (if (> difference (doc-sent-similarity-stand-dev doc-stat))
                (if (> difference (* 2 (doc-sent-similarity-stand-dev doc-stat)))
                  '3-dev-above
                  '2-dev-above)
                '1-dev-above)
              (if (> (- difference) (doc-sent-similarity-stand-dev doc-stat))
                '2-dev-below
                '1-dev-below)))))

       ;; begin function

       ;; add all information directly accessible from the document object
       (add-doc-info doc-structure)
       ;; calculate all information possible from analyzing the parts of the
       ;; document
       (loop for x from 0 to (- (length (sections doc-structure)) 1)
             do (analyze-section doc-structure
                                 x
                                (make-instance 'sentence-stat)))
       ;; more calculations... need a second pass to assimilate
       (do-other-calculations)
       ;; return document characteristics and sentence characteristics array
       (cons doc-stat sent-chars-array))))

;; sets a sentence characteristic to a certain value
(defmethod set-sent-char ((sent-stat sentence-stat) charact value
                          &optional data)
  (set-sent-item (sent-char-list sent-stat) charact value data))

;; sets a sentence slot to a certain value(s)
(defmethod set-sent-cont ((sent-stat sentence-stat) charact value
                          &optional data)
  (set-sent-item (sent-cont-list sent-stat) charact value data))

;; sets a sentence item to a value- used by characteristics and containers
(defun set-sent-item (char-array charact value &optional data)
  (loop for x from 0 to (- (length char-array) 1)
        do (let ((char (aref char-array x)))
             (when (eq charact (car char))
               (setf (aref char-array x) (list (car char) value data))
               (return)))
        finally (break "sentence characteristic ~A not found in sentence data"
                       charact)))

;; create the list of important items the sentence contains; fields can
;; have multiple values.  save in sent-stat object
(defun make-contain-list ()
  (let ((sent-contain-array (make-array 0 :adjustable t :fill-pointer t)))
    (add-charact 'keyphrase-content sent-contain-array)
;   (add-charact 'structure-content sent-contain-array)
    sent-contain-array))

;; create the list of characteristics needed for each sentence- save in
;; sent-stat object
(defun make-charact-list ()
```

```
  (let ((sent-charact-array (make-array 0 :adjustable t :fill-pointer t)))
    (add-charact 'section-loc sent-charact-array)
    (add-charact 'paragraph-loc sent-charact-array)
    (add-charact 'sentence-loc sent-charact-array)
    (add-charact 'key-title sent-charact-array)
    (add-charact 'sent-doc-similarity sent-charact-array)
    (add-charact 'title-content sent-charact-array)
    (add-charact 'sentence-length sent-charact-array)
    sent-charact-array))

;; returns a sentence characteristic
(defmethod get-charact ((sent-stat sentence-stat) charact)
  (let ((char-array (sent-char-list sent-stat))
        (value 'nil))
    (loop for x from 0 to (- (length char-array) 1)
          do (when (eq (car (aref char-array x)) charact)
               (setq value (cdr (aref char-array x)))
               (return))
          finally (break "characteristic ~A not found in sentence-stat object"
                         charact))
    value))

;; helper with make-charact-list; adds another characteristic to the list
(defun add-charact (symbol array)
  (vector-push-extend (cons symbol 'nil) array 15))

;; make a copy of a sent-stat object
(defmethod copy-of ((sent-stat sentence-stat))
  (let ((copy (make-instance 'sentence-stat)))
    (loop for x from 0 to (- (length (sent-char-list sent-stat)) 1)
          do (setf (aref (sent-char-list copy) x)
                   (aref (sent-char-list sent-stat) x)))
    copy))
```

B.5 Template.lisp

```
;; Irene Wilson
;; AI lab
;; Knowledge-Based Collaboration Project
;; created:       ?  IW
;; altered:  1-4-99  IW
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Template.lisp
;;
;; This file contains the code that reads in a "template" file and
;; processes it into a doc-template object.  A different template is
;; made for different types of documents.  The template stores a list of
;; sentence characteristics that are important to determine which
;; sentence is the topic sentence for that particular type of document.
;; It also stores how much weight should be given to each sentence
;; characteristic.  For each characteristic, there are several possible
;; values, each of which receives a value which represents the probability
;; that the sentence is a topic sentence given the value is true.


;; class that contains all info for a doc type
(defclass doc-template ()
  ((charact-array :accessor charact-array
```

```
                :initform (make-array 0 :adjustable t :fill-pointer t))))

;; class that contains all the info for one characteristic of the doc type
(defclass characteristic ()
  ((name :accessor name :initform 'nil)
   (weight :accessor weight :initform 0)
   ;; the slots are the different values for the characteristics and what weight
   ;; they each have
   (slots :accessor slots :initform (make-array 0 :adjustable t :fill-pointer t))))

;; returns a template object that contains all the info needed to process a
;; document of that type
(defun get-template (doc-type)
  (let ((my-template (make-instance 'doc-template)))
    ;; read in the template file of that type
    (with-open-file (template-stream (concatenate 'string
                                                  "Grant-HD1:Users:merri:input:"
                                                  (string-downcase doc-type)
                                                  "-template"))
      (loop for line = (get-line template-stream)
            while line
            ;; lines with # in front are slots; without are char. categories
            do (if (eq (aref line 0) #\#)
                   ;; check to make sure a characteristic comes before a slot
                   (if (> (length (charact-array my-template)) 0)
                     (add-slot my-template line)
                     (break "template format incorrect"))
                   (setf my-template (add-char my-template line)))))
    my-template))

;; add another characteristic to the template
;; returns the new template
(defmethod add-char ((template doc-template) line)
  (let ((word-array (parse-line line)))
    (when (< (length word-array) 2) (break t "template format incorrect"))
    ;; make new characteristic
    (let ((curr-charact (make-instance 'characteristic))
          (val))
      (setf (name curr-charact) (intern (string-upcase (aref word-array 0))))
      ;; set the weight of the new characteristic
      (unless (setq val (parse-float (aref word-array 1)))
        (break "template format incorrect: ~A" (aref word-array 1)))
      (setf (weight curr-charact) val)
      (setf (slots curr-charact) (make-array 0 :adjustable t :fill-pointer t))
      (vector-push-extend curr-charact (charact-array template)))
    template))

;; add another slot value to the current characteristic
(defmethod add-slot ((template doc-template) line)
  ;; get rid of the #s in the beginning of the line
  (let ((start-info (position-if (lambda (x) (not (eq #\# x))) line)))
    (unless start-info (break "template format incorrect: blank line"))
    (let ((word-array (parse-line (subseq line start-info))))
      ;; check that there are at least two words in the line
      (when (< (length word-array) 2)
        (break t "template format incorrect: not enough arguments"))
      (let (;; get the current characteristic to which this slot belongs
            (curr-charact (aref (charact-array template)
                                (- (length (charact-array template)) 1)))
            ;; get the value of the rating of this slot
            (val (parse-float (aref word-array (- (length word-array) 1)))))
        (unless val (break "template format incorrect: ~A" (aref word-array 1)))
        ;; add slot value + all the words in the line to the template
        ;; characteristic array location
        (let ((word-list 'nil))
          (loop for x from (- (length word-array) 2) downto 0
                do (setq word-list
                         (cons (intern (string-upcase (aref word-array x)))
```

```
                              word-list)))
          (vector-push-extend  (cons val word-list) (slots curr-charact)))))))

;; get the next line from a stream that is not all whitespace and does not
;; have a semicolon as the first non-whitespace character
(defun get-line (stream)
  (let ((first-item)
        (next-line 'nil))
    (loop for line = (read-line stream nil nil)
          while line
          do (when (and (setq first-item (position-if #'not-space line))
                        (not (eq (aref line first-item) #\;)))
               (setq next-line line)
               (return)))
    next-line))
```

B.6 Rank.lisp

```
;; Irene Wilson
;; AI lab
;; Knowledge-Based Collaboration Project
;; created:       ?  IW
;; altered: 1-20-99  IW
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Rank.lisp
;;   This file contains the code that ranks the sentences of the document.
;;   It uses the template information to calculate the impact of the data
;;   stored in the sent-statistics array


;; This object is used to keep track of the rating of a sentence.  An array
;; of these objects are created to represent the rating of all the sentences
;; in the document
(defclass sent-rating-info ()
  (;; this is the overall rating of the sentence
   (sent-rating :accessor sent-rating :initform 0)
   ;; an array of the sentence characteristics used to calculate the
   ;; rating
   (chars-used :accessor chars-used
               :initform (make-array 0 :adjustable t :fill-pointer t))
   ;; an array of the values for each of the sentence characteristics in
   ;; chars-used; these numbers summed give the sent-rating
   (val-of-chars-used :accessor val-of-chars-used
                      :initform (make-array 0 :adjustable t :fill-pointer t))))

;; calculates the "goodness" of all the sentences using the outline in the
;; template.  returns an array of ratings for each sentence.  The higher, the more
;; likely that sentence is a good summarizing sentence
(defun make-sentence-rankings (sent-statistics template)
  (let ((sentence-ratings
         (make-array (length sent-statistics) :element-type 'sent-rating-info))
        (all-characters (charact-array template)))

    (labels (;; adds the characteristic "character" to the sentence rating of
             ;; every sentence
             (add-in-character (character char-index mult-val)
               (loop
                 for y from 0 to (- (length sent-statistics) 1)
```

```
                do (let ((sent-stat (aref sent-statistics y)))
                     (if mult-val
                       (let ((sent-cont-info
                              (aref (sent-cont-list sent-stat) char-index)))
                         (calc-slot-value character sent-cont-info y))
                       (let* ((sent-char-info
                               (aref (sent-char-list sent-stat) char-index))
                              (item-val (get-item-val character
                                                      (cadr sent-char-info))))
                         (when (eq item-val -1)
                           (format t "WARNING: sentence characteristic slot
 not found: ~A:~A" (name character)
(cadr sent-char-info))
                           (setq item-val 0))
                         (add-in-slot-value character
                                            sent-char-info
                                            item-val
                                            y))))))

            ;; add the "item-val" multiplied by the characteristic weight
            ;; to the sentence-rating indicated by "index".  Also record the
            ;; characteristic, slot assignment, and intermediate values.
            (add-in-slot-value (character sent-char-info item-val index)
              (let ((prev-rating-info (aref sentence-ratings index)))
                (setf (sent-rating prev-rating-info)
                      (+ (sent-rating prev-rating-info)
                         (* item-val (weight character))))
                (vector-push-extend sent-char-info
                                    (chars-used prev-rating-info)
                                    10)
                (vector-push-extend (list item-val
                                          (weight character)
                                          (* item-val (weight character)))
                                    (val-of-chars-used prev-rating-info)
                                    10)))


            ;; this function is used by characteristic slots that can have
            ;; multiple values.  All the values of the slots are combined into
            ;; one overall value and added to the appropriate sentence's rating
            (calc-slot-value (character sent-char-info index)
              (let ((item-list (cadr sent-char-info))
                    (inverse-accum 1))
                (loop
                  for item-index from 0 to (- (length item-list) 1)
                  do (let ((item-val
                            (get-phrase-val character
                                            (aref item-list item-index))))
                       (unless (eq item-val -1)
                         (setq inverse-accum (* inverse-accum (- 1 item-val)))
                         (format t "VALUE: ~A ~A" (- 1 inverse-accum) #\Return))))

                (add-in-slot-value character
                                   sent-char-info
                                   (- 1 inverse-accum)
                                   index)))

            ;; this looks up the value of a slot assignemnt "item"
            ;; to the characteristic "character"
            (get-item-val (character item)
              (loop
                for curr-slot across (slots character)
                when (equal (cadr curr-slot) item)
                do (return (car curr-slot))
                finally (return 0)))

            ;; this looks up the value of a slot assignemnt "phrase"
            ;; to the characteristic "character"
```

```
          (get-phrase-val (character phrase)
            (loop
              for curr-slot across (slots character)
              when (equal (cdr curr-slot) phrase)
              do (return (car curr-slot))
              finally (return -1))))


    ;; initialize sentence ratings
    (loop
      for x from 0 to (- (length sentence-ratings) 1)
      do (setf (aref sentence-ratings x) (make-instance 'sent-rating-info)))

    ;; go through each characteristic in the report template and accumulate
    ;;   a fitness total for each sentence in the sent-characteristics array
    (loop
      for x from 0 to (- (length all-characters) 1)
      ;; find each template characteristic in sentence stats
      do (let* ((character (aref all-characters x))
                (char-index
                 (get-info-index (name character)
                                 (sent-char-list (aref sent-statistics 0)))))
           (if char-index
             (add-in-character character char-index 'nil)
             (let ((cont-index
                    (get-info-index (name character)
                                    (sent-cont-list (aref sent-statistics 0)))))
               (unless cont-index (break "sentence characteristic not found: ~A"
                                         (name character)))
               (add-in-character character cont-index 't)))))

    sentence-ratings)))


;; find the index of the characteristic "key" in the array of characteristics in
;; sent-stat
(defun get-info-index (key char-array)
  (let ((index 'nil))
    (loop for x from 0 to (- (length char-array) 1)
          do (when (eq key (car (aref char-array x)))
               (setq index x)
               (return)))
    index))
```

B.7 Lib.lisp

```
;; Irene Wilson
;; AI lab
;; Knowledge-Based Collaboration Project
;; created:       ?  IW
;; altered:  1-4-99  IW
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Lib.lisp
;;


;; sets all members of vector to value
```

```
(defun set-vector (vector value)
  (loop for x from 0 to (- (length vector) 1)
        do (setf (aref vector x) value)))

;; This function returns a vector that contains all the key phrases that
;;   might indicate that a sentence is a topic sentence.  Each phrase is
;;   represented by an array of strings.
(defun get-phrases (filename)
  (let ((phrase-array (make-array 0
                                  :adjustable t
                                  :fill-pointer t
                                  :element-type 'string)))
    (with-open-file (phrase-stream (concatenate 'string
                                                "Grant-HD1:Users:merri:input:"
                                                filename)
                                   :direction  :input)
      (loop for phrase = (read-line phrase-stream nil nil)
            while phrase
            do (let ((curr-phrase-array (parse-line phrase)))
                 (when (> (length curr-phrase-array) 0)
                   (vector-push-extend (parse-line phrase) phrase-array 20))))
      phrase-array)))


;; takes a line of text and returns an array of strings; one string for each
;; word.  No whitespace is preserved.
(defun parse-line (line)
  (let ((end 0))
    (loop with word-array = (make-array 0
                                        :adjustable t
                                        :fill-pointer t
                                        :element-type 'string)
          for st = (position-if #'not-space line :start end)
          while st
          do (setq end (or (position #\Space line :start st) (length line)))
             (vector-push-extend (subseq line st end) word-array 5)
          finally (return word-array))))


(defun not-space (x)
  (not (eq x #\Space)))

(defun whitespace-p (x)
  (find x '(#\Space #\Tab)))


;; Parses a string into a float number.
(defun parse-float (str)
  (let ((num 0)
        (decimal (position #\. str)))
    (cond
     (decimal
      (setq num (parse-integer
                 (concatenate 'string
                              (subseq str 0 decimal)
                              (subseq str (+ 1 decimal)))
                 :junk-allowed 't))
      (setq num (/ num (expt 10 (- (- (length str) decimal) 1)))))
     (t (parse-integer str :junk-allowed 't)))))


;; This function takes a file name as input and creates a stream to that file.
;;   (The file should be in my working directory.)
(defun get-stream (file-name)
  (open (make-pathname :directory "Grant-HD1:Users:merri:input" :name file-name)
        :direction
        :input))
```

```
;; These words are ignored when the word statistics of a portion of text is
;; computed.
(defparameter *non-keywords*
  '("a" "an" "and" "as" "at" "be" "but" "by" "for" "in" "is" "if" "it" "not"
    "of" "on" "or" "that" "the" "this" "to" "with"))

;; Returns true if wrd is a keyword, else false
(defun keyword-p (wrd)
  (not (find wrd *non-keywords* :test 'equalp)))

;; Just like position-if with an extra optional input to check that the start location
;; is in bounds.  If both out-of-bounds input is true, and if either there is no start
;; or the start is out of bounds, returns nil.
(defun  my-position-if (predicate proseq &key key from-end start end out-of-bounds-check)
  (if (and out-of-bounds-check (not start))
    'nil
    (if (and start (> start (length proseq)))
      'nil
      (position-if predicate
                   proseq
                   :key (or key 'identity)
                   :from-end (or from-end 'nil)
                   :start (or start 0)
                   :end (or end 'nil)))))

;; Takes the dot product of two vectors.  The vectors do NOT have to be of the
;; same length; if one vector is shorter that the other, the rest of the short
;; vector is effectively padded with zeros.
(defun dot-product (long-vect short-vect)
  (let ((sum 0)
        (total-long 0)
        (total-short 0))
    (loop for x from 0 to (- (length short-vect) 1)
          do (let ((val1 (aref long-vect x))
                   (val2 (aref short-vect x)))
               (setq sum (+ sum (* val1 val2)))
               (setq total-long (+ total-long (* val1 val1)))
               (setq total-short (+ total-short (* val2 val2)))))
    (loop for x from (length short-vect) to (- (length long-vect) 1)
          do (setq total-long (+ total-long
                                 (* (aref long-vect x) (aref long-vect x)))))
    (if (> total-short 0)
      (/ sum (* (sqrt total-long) (sqrt total-short)))
      0)))
```

Appendix C

Sample Output

C.1 Sample Article Output

CHICAGO -- United Airlines and parent UAL Corp. took a vital step toward completing a proposed employee buyout by issuing $1.15 billion in bonds and preferred stock, but the offerings raised less than the $1.5 billion Wall Street had expected.

While the amount fell below estimates, it still represents a crucial step in UAL's bid to persuade shareholders to approve the buyout plan at the company's annual meeting July 12. UAL recently renegotiated the terms of the buyout agreement, offering to give shareholders the proceeds from the offering rather than the securities themselves.

"It's a positive move," said Ray Neidl, an analyst with Furman Selz Inc. "Without it, stockholders would have had less incentive to vote for the deal."

With interest rates rising, investors in UAL stock had grown increasingly fearful that the securities they would have gotten would trade for less than they hoped. UAL's stock plummeted this year from its pre-deal high of around $155, although it has rebounded somewhat recently. In New York Stock Exchange composite trading yesterday, UAL's stock closed at $127 a share, down 25 cents.

Yesterday's offering priced $410.4 million of cumulative preferred stock at $25 each, and priced a split-rated debt offering valued at an additional $741.2 million. The proceeds will be used to make payments of $84.81 a share to current UAL shareholders for the 55% of the company that employees are proposing to buy-a price that is on the low end of what analysts were expecting. In addition, holders will receive one-half of a new share of UAL common.

Some analysts questioned United's ability to back up the bonds, especially since summer airline traffic normally brings lower yields. Dean Sparkman, an airline consultant in Roslyn, Va., said, "The deal's a go. But it's taking on the classic smell of a leveraged buyout. And as a guide, LBO's have not historically been successful in the airline business."

Moreover, UAL said yesterday it would use $300 million of cash reserves to compensate for the shortfall off proceeds from the offerings.

However, according to the company's prospectus for the transaction, cash churned out by UAL will increase $550 million a year on average between now and 1999. "Because of the cash savings, they are going to be able to strengthen the balance sheet in pretty short order," said Mr. Neidl. United currently has more than $1 billion in cash and another $1 billion in short-term investments.

United sold $370.2 million of 10-year notes at par to yield 10.67%. United's $371 million offering of 20-year debentures was priced at par to yield 11.21%.

The company launched a $765 million preferred stock offering at a 12% dividend yield last week, but cut the size of the offering yesterday by $355 million at pricing and increased the dividend to 12.25%. The 16.4 million shares were priced at $25 each, according to the lead manager, Merrill Lynch & Co.

```
? (auto-sum "art8" :numb-sents 3)

1: CHICAGO -- United Airlines and parent UAL Corp. took a vital step
toward completing a proposed employee buyout by issuing $1.15 billion
in bonds and preferred stock, but the offerings raised less than the
$1.5 billion Wall Street had expected.
   SECTION-LOC: FIRST  ==> 1 x 40 = 40
   PARAGRAPH-LOC: FIRST  ==> 1 x 50 = 50
   SENTENCE-LOC: FIRST  ==> 1 x 40 = 40
   SENTENCE-LENGTH: LONG  ==> 1 x 100 = 100
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 25 = 25

2: UAL recently renegotiated the terms of the buyout agreement,
offering to give shareholders the proceeds from the offering rather
than the securities themselves.
   SECTION-LOC: FIRST  ==> 1 x 40 = 40
   PARAGRAPH-LOC: SECOND  ==> 7/10 x 50 = 35
   SENTENCE-LOC: SECOND  ==> 1/2 x 40 = 20
   SENTENCE-LENGTH: LONG  ==> 1 x 100 = 100
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 25 = 25

3: While the amount fell below estimates, it still represents a
crucial step in UAL's bid to persuade shareholders to approve the
buyout plan at the company's annual meeting July 12.
   SECTION-LOC: FIRST  ==> 1 x 40 = 40
   PARAGRAPH-LOC: SECOND  ==> 7/10 x 50 = 35
   SENTENCE-LOC: FIRST  ==> 1 x 40 = 40
   SENTENCE-LENGTH: LONG  ==> 1 x 100 = 100
   SENT-DOC-SIMILARITY: 2-DEV-BELOW  ==> 0 x 25 = 0
```

C.2 Sample Documentation Output

Environment for running local transports

Local transports handle deliveries to files and pipes. (The 'autoreply' transport can be thought of as similar to a pipe.) Whenever a local transport is run, Exim forks a subprocess for it. Before running the transport code, it sets a specific uid and gid by calling 'setuid()' and 'setgid()'. It also sets a current file directory; for some transports a home directory setting is also relevant.

The values used for the uid, gid, and the directories may come from several different places. In many cases the director that handles the address associates settings with that address. However, values may also be given in the transport's own configuration, and these override anything that comes with the address. The sections below contain a summary of the possible sources of the values, and how they interact with each other.

Uids and gids

All local transports have the options 'group' and 'user'. If 'group' is set, it overrides any group that may be set in the address, even if 'user' is not set. This makes it possible, for example, to run local mail delivery under the uid of the recipient, but in a special group. For example:

```
group_delivery:

  driver = appendfile   file = /var/spool/mail/${local_part}
  group = mail
```

If 'user' is set, its value overrides what is set in the address. If 'user' is non-numeric and 'group' is not set, the gid associated with the user is used. If 'user' is numeric, then 'group' must be set.

The 'pipe' transport contains the special option 'pipe_as_creator'. If this is set and 'user' is not set, the uid of the process that called Exim to receive the message is used, and if 'group' is not set, the corresponding original gid is also used.

When the uid is taken from the transport's configuration, the 'initgroups()' function is called for the groups associated with that uid if the 'initgroups' option is set for the transport; 'pipe' is the only transport that has such an option.

When the uid is not specified by the transport, but is associated with the address by a director or router, the option for calling 'initgroups()' is taken from the director or router configuration. All directors and routers have

'group', 'user', and 'initgroups' options, which are used as follows:

For the 'aliasfile' director they specify the uid and gid for local deliveries generated directly -- that is, deliveries to pipes or files. They have no effect on generated addresses that are processed independently.

The 'forwardfile' director's 'check_local_user' option causes a password file lookup for the local part of an address. The uid and gid obtained from this lookup are used for any directly generated local deliveries, but they can be overridden by the 'group' and 'user' options of the director. As for 'aliasfile', these values are not used for generated addresses that are processed independently.

The 'localuser' director looks up local parts in the password file, and sets the uid and gid from that file for local deliveries, but these values can be overridden by the director's options.

For the 'smartuser' director and all the routers, the 'group', 'user', and 'initgroups' options are used only if the driver sets up a delivery to a local transport.

Current and home directories

The 'pipe' transport has a 'home_directory' option. If this is set, it overrides any home directory set by the director for the address. The value of the home directory is set in the environment variable HOME while running the pipe. It need not be set, in which case HOME is not defined.

The 'appendfile' transport does not have a 'home_directory' option. The only use for a home directory in this transport is if the expansion variable '$home' is used in one of its options, in which case the value set by the director is used.

The 'appendfile' and 'pipe' transports have a 'current_directory' option. If this is set, it overrides any current directory set by the director for the address. If neither the director nor the transport sets a current directory, then Exim uses the value of the home directory, if set. Otherwise it sets the current directory to '/' before running a local transport.

The 'aliasfile', 'forwardfile', and 'localuser' directors all have 'current_directory' and 'home_directory' options, which are associated with any addresses they explicitly direct to a local transport.

For 'forwardfile', if 'home_directory' is not set and there is a 'file_directory' value, that is used instead. If it too is not set, but 'check_local_user' is set, the user's home directory is used. For 'localuser', if 'home_directory' is not set, the home directory is taken from the password file entry that this director looks up. There are no defaults for 'current_directory' in the directors, because it defaults to the value of 'home_directory' if it is not set at transport time.

The 'smartuser' director and all the routers have no means of setting up home and current directory strings; consequently any local transport that they use must specify them for itself if they are required.

Expansion variables derived from the address

Normally a local delivery is handling a single address, and in that case the variables such as '$domain' and '$local_part' are set during local deliveries. However, in some circumstances more than one address may be handled at once (for example, while writing batch SMTP for onward transmission by some other means). In this case, the variables associated with the local part are never set, '$domain' is set only if all the addresses have the same domain, and '$original_domain' is never set.

```
? (auto-sum "doc8" :numb-sents 4)

1:  Local transports handle deliveries to files and pipes.
   SECTION-LOC: FIRST  ==> 1 x 15 = 15
   SENTENCE-LOC: FIRST  ==> 1 x 20 = 20
   SENTENCE-LENGTH: SHORT  ==> 1 x 30 = 30
   SENT-DOC-SIMILARITY: 2-DEV-BELOW  ==> 0 x 10 = 0

2: The values used for the uid, gid, and the directories may come from
several different places.
   SECTION-LOC: SECOND  ==> 4/5 x 15 = 12
   SENTENCE-LOC: FIRST  ==> 1 x 20 = 20
   SENTENCE-LENGTH: SHORT  ==> 1 x 30 = 30
   SENT-DOC-SIMILARITY: 2-DEV-BELOW  ==> 0 x 10 = 0

3: The 'smartuser' director and all the routers have no means of
setting up home and current directory strings; consequently any local
transport that they use must specify them for itself if they are
required.
   SECTION-LOC: BODY  ==> 0 x 15 = 0
   SENTENCE-LOC: FIRST  ==> 1 x 20 = 20
   SENTENCE-LENGTH: LONG  ==> 1 x 30 = 30
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 10 = 10

4: Normally a local delivery is handling a single address, and in that
case the variables such as '$domain' and '$local_part' are set during
local deliveries.
   SECTION-LOC: BODY  ==> 0 x 15 = 0
   SENTENCE-LOC: FIRST  ==> 1 x 20 = 20
   SENTENCE-LENGTH: LONG  ==> 1 x 30 = 30
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 10 = 10
```

C.3 Sample Report Output

December, 1994

Designing NewsMaker:
Ethnographic Methods Applied
in Elementary School

Michele Evard, Noah Breslow
MIT Media Lab
20 Ames Street, E15-320
Cambridge, MA 02139, USA
Tel: 1-617-253-0330
E-mail: mevard@media.mit.edu

L. Mark Kortekaas
NBC Desktop News
30 Rockefeller Plaza
New York, New York 10112, USA
Tel: 1-212-664-5292
E-mail: markk@media.mit.edu

ABSTRACT

This paper presents the participatory design process for an on-line communications environment which we created for children's use in school. Rather than conducting controlled lab experiments, we introduced our initial implementation of the system to children for use in their school projects. We applied ethnographic techniques, including observations, informal discussions, and interviews, to obtain the children's input on further design decisions. From these experiences we obtained a clear picture of how students in a natural setting would use the software.

KEYWORDS: Ethnography, educational applications, participatory design, communication, news.

INTRODUCTION

The application of participatory design methods to the creation of new environments for children is complicated by several factors, including the typical roles adults play in a child's life. The children must first be convinced that a designer is not trying to instruct or test them; even once trust is established, they may be unable to imagine the system being designed or the uses to which they may put it. In this paper we describe how we brought the initial implementation of a software system into a school for the children's use, and then, using ethnographic methods, engaged the children in the design

process.

We feel that this approach is particularly relevant to designs done by adults for children. Even some environments which are meant to engage children as active learners [5] have not been designed with children's aid. Focus groups have been used, but because the children sometimes have a difficult time imagining how they might use a planned system, their success has been limited. We hope that this paper will provide designers with concrete methods for involving children in design.

NEWSMAKER

NewsMaker was originally designed to be a production tool which would allow children to create personal newspapers. Children would be able to create their own articles, edit in-house as well as external news articles, and select articles to assemble into a printed paper using automatic layout [3].

A few additions to the original design were made to provide an in-house Usenet-style infrastructure. We did not include all of the functionality which we would want, preferring to allow the children to participate fully in that part of the design. Both aspects of the design were informed by prior discussions with children about their use of news [2].

ETHNOGRAPHIC METHODS APPLIED

We introduced NewsMaker to students in two fourth-grade and four fifth-grade classes in a Boston inner-city public elementary school. We emphasized to each group that the design was not complete, and that while they were doing their projects, we would like their input on how the system worked--or did not work. We followed two of the classes each time they used the system, and observed the others occasionally. The students with whom Evard worked were involved daily in the design and implementation of educational video games; they used NewsMaker at will to ask and respond to questions about game design during the entire four-month process [1]. Kortekaas assisted a class which used NewsMaker twice a week for seven weeks to create individual newspapers. Our role during these sessions was to help the children with their projects.

Observations

Some of the problems the children encountered occurred during the first sessions of use. For example, we noticed that the children had difficulties with mouse manipulation and therefore provided keyboard alternatives to double-clicking. While this type of problem could have been identified in an experimental situation, children encountered other types of difficulties during normal use over time. For example, when a large number of articles had accumulated in one group, children tried to use the keyword search as an author search to find articles which they knew had been written by certain classmates. This misunderstanding would not have occurred in an experimental setting as the children would not have been using large groups of articles by people they knew.

Discussions During Use

When one of us would see a student having difficulties or would be called over to assist a student, we would talk with the child about what he or she was attempting to do. This often illuminated their understanding of the system. For example, one girl called Evard over to ask why delete was not working. She had posted a response which she decided sounded rude, and wanted to delete it. To do so, she replied to her own post, cut out her comment, and reposted. She tried this several times before requesting help; through discussion with her it became apparent that she thought replies replaced the original message. Another girl said that she didn't like to use the reply command because if the previous students had not signed their names to their messages, it appeared as if she had written their part or tried to take credit for it. We believe that these types of issues would not have come to our attention during other types of usability studies.

Requests

On several occasions a student would initiate a conversation with one of the observers and request a change or addition to NewsMaker. The first request was for a reply command that would include the original message. The most common request was for a new group; one of these was a request for weather forecasts from an external source, but most of them were for new groups to which the children could contribute. The groups included rap music, puzzles, school news, Logo programming, ecology, Japan, video games, and book reviews. Students also requested changes to the interface, such as making a child's name visible on the same screen as his or her article, and to have particular hot keys for tasks which they did frequently.

Formal Interviews

We conducted individual interviews with students. While the students would freely discuss their opinions of what they and others used NewsMaker for, they seemed to find it difficult to point out particular design issues. Several of the children articulated problems which they had had during use, but these had generally been observed prior to the interview.

TIMING OF CHANGES

We made some modifications during these four months, but chose to do most of the changes after the school year ended to avoid disturbing the children's projects. The potential for disruption was made clear to us when we renamed one of the system's menu titles. Even though several of the children had requested this change to a more commonly used term, when it was implemented many of the children were confused. Such disruption could perhaps be avoided by discussing each change with all of the groups of children, but it was felt that this would distract students from their required activities. The changes are now being tested by children and have met with positive reactions.

IMPLICATIONS

In addition to advising us on software design, the children created activities, chose discussion topics, and set guidelines about the appropriateness of certain types of messages. Future work will focus on these areas. Additional modifications to the environment will be made if and when required by the

students.

This work has demonstrated that nine- and ten-year-old children can contribute constructively to the design of environments for their use. Our use of ethnographic methods during natural use facilitated their participation. It is our hope that other designers will be able to use these or similar methods to allow children to aid in the design of systems meant for their use.

ACKNOWLEDGMENTS

The children of Project Headlight who participated in our work are responsible for its success; any problems are certainly due to the authors and not the children. We would also like to thank the teachers as well as our advisors and other members of the Media Lab who have been helpful in this work. This research was supported by the News In The Future Consortium and the MIT Media Lab.

REFERENCES

1. Evard, M. Articulation of Design Issues: Learning Through Exchanging Questions and Answers. In Y. Kafai and M. Resnick (Eds.), Constructionism in Practice: Rethinking the Roles of Technology in Learning. MIT Media Laboratory, Cambridge, MA, 1994.

2. Evard, M. What Is "News"?: Children's Conceptions and Uses of News. Annual Meeting of the American Educational Research Association (April 1994, New Orleans, LA).

3. Kortekaas, L. M. News and Education: Creation of "The Classroom Chronicle." Master's thesis, MIT Media Laboratory, Cambridge, MA, 1994.

4. Monk, A., B. Nardi, N. Gilbert, M. Mantei & J. McCarthy. Mixing oil and water? Ethnography versus Experimental psychology in then study of computer-mediated communication. Proceedings of CHI'93, 3-6, ACM New York, 1993.

5. Papert, S. The Children's Machine. Basic Books, New York, 1993.

```
? (auto-sum "rep8" :numb-sents 4)

1: This paper presents the participatory design process for an on-line
communications environment which we created for children's use in
school.
   SECTION-LOC: SECOND  ==> 3/5 x 20 = 12
   SENTENCE-LOC: FIRST  ==> 1 x 4 = 4
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 10 = 10
   KEYPHRASE-CONTENT: #()  ==> 0 x 30 = 0
   TITLE-CONTENT: NIL  ==> 0 x 30 = 0
   SENTENCE-LENGTH: LONG  ==> 1 x 30 = 30

2:  Copyright 1994 by the Massachusetts Institute of Technology.
```

```
   SECTION-LOC: FIRST  ==> 1 x 20 = 20
   SENTENCE-LOC: FIRST  ==> 1 x 4 = 4
   SENT-DOC-SIMILARITY: 2-DEV-BELOW  ==> 0 x 10 = 0
   KEYPHRASE-CONTENT: #()  ==> 0 x 30 = 0
   TITLE-CONTENT: NIL  ==> 0 x 30 = 0
   SENTENCE-LENGTH: SHORT  ==> 1 x 30 = 30

3: From these experiences we obtained a clear picture of how students
in a natural setting would use the software.
   SECTION-LOC: SECOND  ==> 3/5 x 20 = 12
   SENTENCE-LOC: BODY  ==> 0 x 4 = 0
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 10 = 10
   KEYPHRASE-CONTENT: #()  ==> 0 x 30 = 0
   TITLE-CONTENT: NIL  ==> 0 x 30 = 0
   SENTENCE-LENGTH: LONG  ==> 1 x 30 = 30

4: Rather than conducting controlled lab experiments, we introduced
our initial implementation of the system to children for use in their
school projects.
   SECTION-LOC: SECOND  ==> 3/5 x 20 = 12
   SENTENCE-LOC: SECOND  ==> 0 x 4 = 0
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 10 = 10
   KEYPHRASE-CONTENT: #()  ==> 0 x 30 = 0
   TITLE-CONTENT: NIL  ==> 0 x 30 = 0
   SENTENCE-LENGTH: LONG  ==> 1 x 30 = 30
```

C.4 Sample Speech Output

THE WHITE HOUSE

Office of the Press Secretary
(Highfill, Arkansas)

For Immediate Release November 7, 1998

REMARKS BY THE PRESIDENT
IN RADIO ADDRESS TO THE NATION

The Oval Office

THE PRESIDENT: Good morning. This week the American people sent a clear message to Washington that we must put politics aside and take real action on the real challenges facing are nation: saving Social Security for the 21st century, passing a patients' bill of rights, strengthening our schools by finishing the job of hiring 100,000

teachers and passing my plan to build or modernize 5,000 schools across our country.

Over the past six years, we have taken real action to address another important challenge: making our communities safe for our families. For too long it seemed that rising crime was a frightening fact of life in America. In too many communities children could not play on the street or walk to school in safety, older Americans locked themselves in their homes with fear, and gangs armed with illegal guns boldly roamed our streets and schools.

I took office determined to change this, committed to a comprehensive anti-crime strategy based on more community policing, tougher penalties, and better prevention. Today our strategy is showing remarkable results. We're ahead of schedule and under budget in meeting our goal of putting 100,000 police on the street. And all across America, crime rates have fallen to a 25-year low, respect for the law is on the rise, families are beginning to feel safe in their communities again.

Keeping guns out of the hands of criminals has been at the center of our strategy, and an essential part of our success. Since I signed the Brady Law, after a big debate in Congress which was led in the House of Representatives by now Senator-elect Charles Schumer of New York, background checks have put a stop to nearly a quarter of a million handgun purchases by fugitives or felons. Law enforcement officers from around the country have told us that fewer guns on the street have made a huge difference in the lives of families they serve.

At the end of this month, we will make the Brady Law even stronger. For the first time ever, we will require background checks for the purchase of any firearm, whether purchased from a licensed gun deal or a pawn shop. But under this new Insta-Check system, as it's called, we'll be able to run nearly twice as many background checks, and most of them in just a matter of minutes.

We've spent five years working with state and local law enforcement to put this system in place, but when it comes to our families' safety, we must take another important step. Every year, an untold number of firearms are bought and sold at an estimated 5,000 gun shows around our country. I come from a state where these shows are very popular. I have visited and enjoyed them over the years. They're often the first place parents teach their children how to handle firearms safely. I know most gun dealers and owners are dedicated to promoting safe and legal gun use.

But at too many gun shows, a different, dangerous trend is emerging. Because the law permits some firearms to be sold without background checks, some of these gun shows have become illegal arms bazaars for criminals and gun traffickers looking to buy and sell guns on a cash-and-carry, no-questions-asked basis.

On Tuesday, the people of Florida voted overwhelmingly to put a stop to these tainted transactions and make it harder for criminals to buy

firearms. Under the new Florida law, communities now can take action to require background checks for the public sale of all guns. I believe this should be the law of the land: No background check, no gun, no exceptions.

Therefore, I am directing Secretary Rubin and Attorney General Reno to report back to me in 60 days with a plan to close the loophole in the law and prohibit any gun sale without a background check. We didn't fight as hard as we did to pass the Brady Law only to let a handful of unscrupulous gun dealers disrespect the law, undermine our progress, put the safety of our families at risk. With this action, we are one step closer to shutting them down.

I look forward to working together with members of both parties in the new Congress to meet this challenge and all our challenges to build a safer and stronger America for the 21st century.

Thanks for listening.

END

```
? (auto-sum "sp8" :numb-sents 3)

1: This week the American people sent a clear message to Washington
that we must put politics aside and take real action on the real
challenges facing are nation: saving Social Security for the 21st
century, passing a patients' bill of rights, strengthening our schools
by finishing the job of hiring 100,000 teachers and passing my plan to
build or modernize 5,000 schools across our country.
   SECTION-LOC: FIRST  ==> 1 x 5 = 5
   SENTENCE-LOC: SECOND  ==> 1 x 15 = 15
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 10 = 10
   SENTENCE-LENGTH: LONG  ==> 1 x 30 = 30
   KEYPHRASE-CONTENT: #()  ==> 0 x 25 = 0

2: Because the law permits some firearms to be sold without background
checks, some of these gun shows have become illegal arms bazaars for
criminals and gun traffickers looking to buy and sell guns on a
cash-and-carry, no-questions-asked basis.
   SECTION-LOC: BODY  ==> 0 x 5 = 0
   SENTENCE-LOC: SECOND  ==> 1 x 15 = 15
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 10 = 10
   SENTENCE-LENGTH: LONG  ==> 1 x 30 = 30
   KEYPHRASE-CONTENT: #()  ==> 0 x 25 = 0

3: Therefore, I am directing Secretary Rubin and Attorney General Reno
to report back to me in 60 days with a plan to close the loophole in
the law and prohibit any gun sale without a background check.
   SECTION-LOC: BODY  ==> 0 x 5 = 0
   SENTENCE-LOC: FIRST  ==> 1 x 15 = 15
   SENT-DOC-SIMILARITY: 3-DEV-ABOVE  ==> 1 x 10 = 10
   SENTENCE-LENGTH: LONG  ==> 1 x 30 = 30
   KEYPHRASE-CONTENT: #((THEREFORE))  ==> 0 x 25 = 0
```

Appendix D

Test Document Sources

D.1 Documentation Sources

alpha-bits.ai.mit.edu/projects/iiip/doc/cl-http/home-page.html
markl.tech.ftech.net/Exim/exim-html-2.10/doc/html/spec.html
ariel.usc.edu/manuals/matlab52/techdoc/basics/gettingtoc.html
zowie.metnet.navy.mil/~mundyj/METCASTClient-WebHelp/WHStart.htm
www.left-coast.com/docs/java/langspec-1.0/index.html
tahiti.salesforce.com/docs/oracle8/SERVER803/INDEX.HTM
tahiti.salesforce.com/docs/C/STL_doc/
www.comp.utas.edu.au/documentation/python/tut/

D.2 Report Sources

www.ccrl.nj.nec.com/html/publication/index.html,
elib.stanford.edu, el.www.media.mit.edu/groups/el/elpapers.html

Appendix E

Templates Used in Testing

E.1 Article Template

```
;; here is the template for a newspaper-type document.

;; explanation: each line that does not begin with a pound sign
;; represents a sentence characteristic.  After each pound sign,
;; there is a possible value for that characteristic.  Each possible
;; value is assigned a number.  The higher the number is, the more
;; likely it is that a sentence with that characteristic value is
;; a topic sentence.

section-loc 40  ;; important
#first    1
#second   0
#last     .2
#body      0

paragraph-loc 50  ;; v.important
# first   1
# second  .7
# last    0
# body    .3

sentence-loc  40
# first   1
# second  .5
```

```
# last     .1
# body     0

sentence-length 100
# very-short 0
# short     1
# long     1

sent-doc-similarity 25
# 1-dev-above  .5
# 2-dev-above  .8
# 3-dev-above   1
# 1-dev-below  .2
# 2-dev-below   0
```

E.2 Documentation Template

```
;; here is the template for a documentation-type document.

;; explanation: each line that does not begin with a pound sign
;; represents a sentence characteristic.  After each pound sign,
;; there is a possible value for that characteristic.  Each possible
;; value is assigned a number.  The higher the number is, the more
;; likely it is that a sentence with that characteristic value is
;; a topic sentence.

section-loc 15  ;; important
#first     1
#second    .8
#last      .2
#body       0

sentence-loc  20
# first    1
# second  .3
# last      .2
# body     0

sentence-length  30
# very-short  0
```

```
# short    1
# long     1


sent-doc-similarity  10
# 1-dev-above  .5
# 2-dev-above  .8
# 3-dev-above  1
# 1-dev-below  .2
# 2-dev-below  0
```

E.3 Report Template

```
;; here is the template for a report-type document.

;; explanation: each line that does not begin with a pound sign
;; represents a sentence characteristic.  After each pound sign,
;; there is a possible value for that characteristic.  Each possible
;; value is assigned a number.  The higher the number is, the more
;; likely it is that a sentence with that characteristic value is
;; a topic sentence.

section-loc 20  ;;v. important
# first  1
# second .6
# last   .8
# body   0

sentence-loc  4
# first   1
# second  0
# last    0
# body    0

sent-doc-similarity 10
# 1-dev-above  .5
# 2-dev-above  .8
# 3-dev-above   1
# 1-dev-below  .2
# 2-dev-below   0
```

```
keyphrase-content  30
# this paper      1
# will show       .4
# have shown      .4
# in summary      .5
# in conclusion   .5
# introduce       .2
# introduced      .2
# describe        .2
# described   .2
# presented   .2
# propose     .2
# proposed    .2


title-content 30
# abstract  1
# introduction  1
# conclusion  .4
# conclusions  .4

sentence-length 30
# very-short 0
# short    1
# long    1
```

E.4 Speech Template

```
;; here is the template for a speech-type document.

;; explanation: each line that does not begin with a pound sign
;; represents a sentence characteristic.  After each pound sign,
;; there is a possible value for that characteristic.  Each possible
;; value is assigned a number.  The higher the number is, the more
;; likely it is that a sentence with that characteristic value is
;; a topic sentence.

section-loc  5
#first       1
```

```
#second      0
#last        .8
#body        0

sentence-loc  15
# first   1
# second  1
# last    .8
# body    0

sent-doc-similarity  10
# 1-dev-above  .5
# 2-dev-above  .8
# 3-dev-above  1
# 1-dev-below  .2
# 2-dev-below  0

sentence-length   30
# very-short   0
# short        .7
# long         1

keyphrase-content  25
# Today           .4
# like to talk     1
# like to speak    1
# want to talk     1
# want to speak    1
# that's why       .4
# I urge            .7
# announce         .7
```

Appendix F

Summary Analysis Results

The following data represents the scoring of the test documents.

The first column is the number of the document.

The second column contains two numbers separated by a slash. This represents the number of sentences that the human testers circled and underlined, respectively.

The third column represents the overlap between the human chosen sentences and the sentences extracted by the AutoExtractor. An o represents a circled sentence, a - represents an underlined sentence, and a x represents a sentence that was neither circled nor underlined. The leftmost symbol represents human's evaluation of the first sentence the AutoExtractor selected, the next position represents the next sentence automatically selected, etc. For example, the code "x-o" would indicate that the first sentence of the three the AutoExtractor selected was neither circled or underlined. The second was underlined, and the third was circled.

The last column represents the score of the summary, calculated according to the formula given in section 4.3.1.

```
Documents:

1 1/4 oxxxx 1
2 1/4 oxxxx 1
3 1/4 oxxxx 1
4 1/3 xoxx .875
```

```
5 1/3 o-xx 1.29
6 1/3 oxxx 1
7 1/4 xoxxx .9
8 1/4 o-xxx 1.23
9 1/4 o-xxx 1.23
10 1/4 oxxx- 1.15
11 1/4 xxxx- .15
12 1/3 xxx- .208
13 1/4 o-xxx 1.225
14 1/3 xxxx 0
15 1/2 xox .833

Reports:

1 2/5 oxxxx-x .629
2 1/3 oxxx 1
3 2/5 xxxxxxx 0
4 2/5 xxoxxo- .864
5 0/3 xx- .222
6 1/3 xoxx .875
7 1/4 xxxxx 0
8 2/4 -xxxx- .396
9 1/4 oxxxx 1
10 2/4 xxxoxx .375
11 2/4 xxxxxx 0
12 1/4 xxxxx 0
13 1/4 xoxxx .9
14 1/4 xx-xx .2
15  2/5 x-xx-x  .329

Articles:

1 1/2 oxx 1
2 1/3 -xxx .333
3 1/2 oxx 1
4 1/2 oxx 1
5 1/3 xxxx 0
6 1/3 xxxx 0
7 1/3 xxxx 0
8 1/2 o-x 1.42
9 1/3 oxx- 1.21
10 1/3 xxox .75
11 1/2 o-x 1.42
12 1/2 oxx 1
13 1/3 x-xx .292
```

```
14 0/3 -xx .333
15 1/2 xox .833

Speeches

1 1/2 -xo 1.17
2 1/3 oxxx 1
3 1/2 xxx 0
4 1/2 xxx 0
5 1/2 xx- .333
6 1/2 xx- .333
7 1/2 xxx 0
8 1/2 xxx 0
9 0/3 xx- .333
10 1/3 xx-x .25
11 1/3 xxxx 0
12 1/2 xx- .333
13 1/2 oxx 1
14 1/2 xxx 0
15 1/2 oxx 1
```

Bibliography

[1] Salton G., Mitra M., Buckley C., and Singha A. Automatic analysis, theme generation, and summarization of machine readable texts. *Science*, 264, 1994.

[2] Rush J.E. Automatic abstracting and indexing. ii. production of abstracts by application of contextual inference and sytntactic coherence criteria. *Journal of the American Societny for Information Science*, 22(4), 1971.

[3] Tait J.I. Generating summaries using a script based language analyser. In Steels L. and Campbell J.A., editors, *Progress in artificial intelligence*. Ellis Horwood, Chichester, 1985.

[4] Hasida K., Ishizaki S., and Isahara H. A connectionist approach to the generation of abstracts. In G. Kempen, editor, *Natural Language Generation: New Results in Artificial Intelligence, Psychology and Lunguistics*. Martinus Nijhoff, Dordrecht, the Netherlands, 1987.

[5] McKeown K.R. and D.R. Radev. Generating summaries of multiple news articles. In *Proceedings of the eighteenth Annual International ACM SIGIR Conference on Research and Development in IR*, 1995.

[6] Alterman R. and Bookman L. Some computational experiments in summarization. *Discourse Processes*, 13, 1990.

[7] Fung R. Applying bayesian networds to information retrieval. *Comms of the ACM*, 38, 1995.

[8] Mitkov R., Le Roux D., and Descles J.P. Knowledge-based automatic abstracting: Experiments in the sublanguage of elementary geometry. In C. Martin-Vide, editor, *Current Issues in Mathematical Linguistics*. North-Holland, The Netherlands, 1994.

[9] Resnick A. Rath G.J. and Savage R. The formation of abstracts by the selection of sentences: Part 1: sentence selection by man and machines. *American Documentation*, 12(2), 1961.

[10] Gerard Salton. Historical note: the past thirty years in information retrieval. Technical Report TR-87(827):16, Cornell University, April 1987.

[11] Gerard Salton. Term weighting approaches in automatic text retrieval. Technical Report TR-87(881):21, Cornell University, November 1987.

[12] Simone Teufel and Marc Moens. Sentence extraction as a classification task. http://www.cogsci.ed.ac.uk/simone/ac197/teufel-moens97.html, 1997.