Summarizing the Storm of Mass Media

By: James Kitti
Advisor: Ana Stanescu
Department of Computer Science
University of West Georgia

Research Question

Problem:

In modern society, new articles, reports, journals, and research papers are written every day and distributed en-masse thanks to the World Wide Web. The information from these articles can be invaluable to scientists, doctors, journalists, marketers, and other people across a wide spectrum of domains. It is essential for these professionals to able to quickly and efficiently process the vast amounts of information contained across these documents. The goal of our research project was to explore the problem of automated text summarization of documents into a concise, legible, and informative summary.

Image source: dawn media group https://www.dawn.com/news/733766

Proposed Solution:

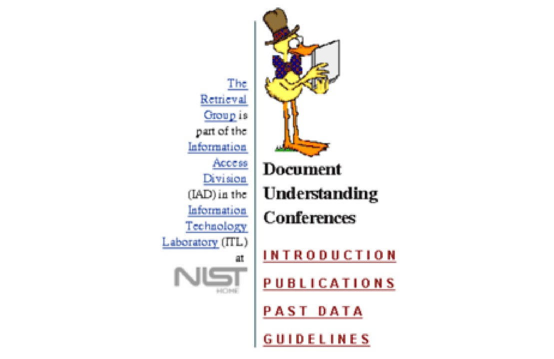
Judging by the daily expansion of documents, it can reasonably be concluded that a program or algorithm that can effectively process and summarize the text contained in these documents would be a boon to data-seeking professionals.

This can best be accomplished via automatic summarization through machine learning and data mining techniques. The goal of this project was the development of a software solution that could effectively summarize text using sentence extraction and graph-based models.

Modern-day text summarization is primarily performed through sentence extraction. Most summarization programs parse a text while calculating the TF (Term Frequency) * IDF (Inverse Document Frequency) values for each word. Each sentence is then parsed and their importance is calculated based on the TF*IDF values of the various words contained in the sentence. These sentences are then proposed as a summary of the text.

Dataset:

We chose the Document Understanding Conference (DUC) 2001 dataset. This dataset consisted of 304 news articles, each having attached corresponding human generated abstract. This dataset fit well with our intent of condensing down the mass amount of articles released onto the internet daily.

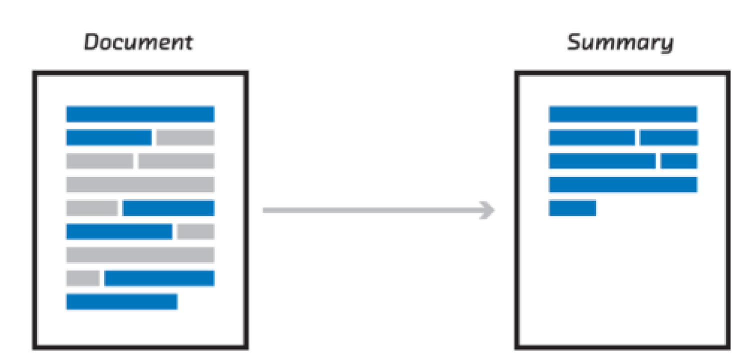


Methodology and Experimental Setup

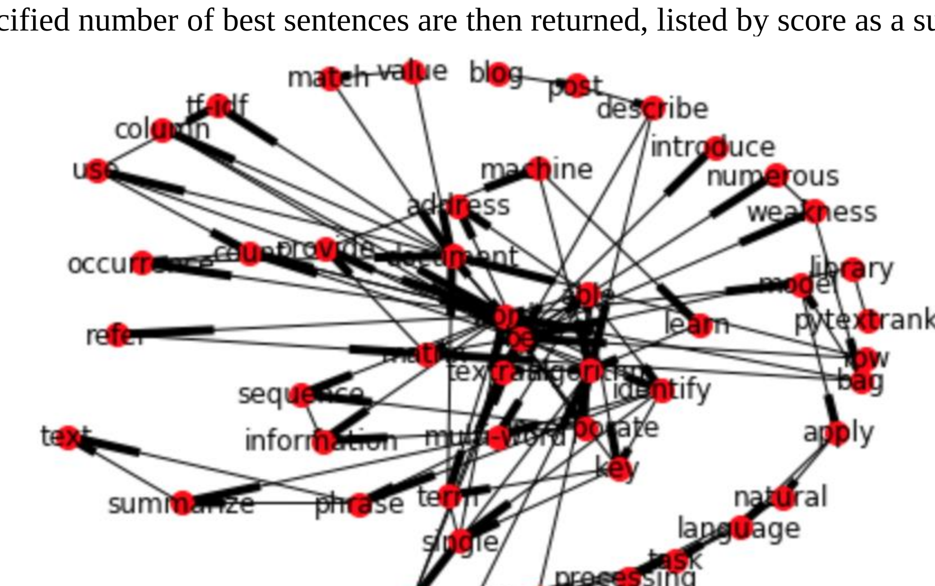
Implementations:

Using Java as our language, we made use of Apache's OpenNLP library to process the corpus. To start, we process the entire document into a string, then split it into paragraphs using the Windows system property line.separator \r\n. Next we make use of OpenNLP's sentence detector to identify the individual sentences within each paragraph, and then tokenize each word present within the sentence.

With our text processed, we proceed to deciding which sentences would be the best to extract in order to create our summary. Since paragraphs usually contain an individual theme or point, a good summary would naturally be the most relevant sentence of each significant paragraph. To start we need to decide how paragraphs would be judged for relevance, with the most obvious choice being based off length, so we decided any paragraph under two sentences long probably didn't contain summary-relevant information. After removing the smaller paragraphs, we began judging each sentence of each paragraph against every other sentence in that paragraph. Using the word tokens of each sentence, we looked for word intersections between the two sentences. For each individual word present in both sentences we increased the score/rank of that sentence by one. Other criteria used to determine the highest score was sentence size as sentence position relative to the rest of the paragraph. The highest scoring sentence is then extracted and added to the summary.



Our final approach was a Python implementation of **TextRank** known as pyTextRank. TextRank is a graph based ranking algorithm where individually tokenized words are vertices, and links are either repeated instances of the same root or based on skip-grams. TextRank notably finds part-of-speech annotations and lemmas, the base forms of each word, before ranking its sentences. Stochastic link analysis also identifies nouns which have a high probability of inbound references. TextRank's feature vector is the ranked key phrases extracted from the original text. This leads to more weights that help determine the score of each sentence. In order to rank the best sentences, each sentence is compared to the feature vector by semantic similarity. The specified number of best sentences are then returned, listed by score as a summary.



Illustrated example of a TextRank Graph

Results

Evaluation:

To assess the effectiveness of our algorithms we use the **ROUGE**, or **Recall-Oriented Understudy for Gisting Evaluation** metric. This metric is used to judge the overlap between automatically produced summaries and reference/human summaries. By separating the automatic summary into multi-word grams (Unigram, Bigram, etc.) it gauges the gauges how well the automatic summary matches the reference. The two results given by the Rouge metric are Recall and Precision.

- **Recall** measures the amount of words/N-Grams present in the automatic summary and the reference summary (*i.e.*, the human generated summarized text)
- **Precision** measures the word count of automatic summary versus the reference.

Implementation	ROUGE Unigram (ROUGE-1)	ROUGE Bigram (ROUGE-2)	ROUGE Trigram (ROUGE-3)
Java Paragraph Summarizer	*Average Recall*		
	0.5479	**0.2637**	**0.1784**
	Average Precision		
	0.2404	0.1100	0.0689
PyTextRank	*Average Recall*		
	0.3578	0.1379	0.0827
	Average Precision		
	0.3848	**0.1430**	**0.0824**

Results in terms of average recall and average precision of the two implementations, our Java-based approach and PyTextRank, on the DUC 2001 dataset.

Conclusions:

Interestingly, our Java implementation achieved a higher recall than the PyTextRank implementation but had a lower precision for all three N-Grams. This shows that while recall is slightly increased, the produced summary contains a lot of excess in terms of word count. This is likely due to the proposed Java-based implementation extracting more sentences than the PyTextRank algorithm, which in turn shows the limits of an extractive implementation.

Due to nature of sentence extraction, information is already limited to a preset length and relevance. As you pull more sentences for a summary, you can gain a higher recall rate, but at the cost of possibly pulling less relevant details along with it. Depending on the document, a summary can be shortened down to a single sentence, however an *Extractive* summarizer doesn't have the discretion to form new, more relevant sentences and is thus limited by the document itself.

This issue is addressed by *Abstractive* text summarizers, which have the ability to create new sentences by training on vast datasets. Using tools such as simile matching, the abstract summarizer can recognize individual parts of speech, allowing for a more nuanced view of the document. These implementations are extremely resource heavy and require large-scale computing environments.