Text Summarization

Authors: John Frazier and Jonathan Perrier

Abstract

For any piece of text, someone may need to know the information the piece provides, but does not have the inclination or time required to read the whole piece. Using some sort of text summarization algorithm can remedy this problem by reducing the amount needed to be read while keeping the information intact. The following paper will implement the LexRank algorithm, Luhn's Auto-Abstract algorithm, and a very naïve brute force algorithm. In addition we will evaluate the summaries given by the three algorithms using the Rouge-1 metric with one "gold standard" summary.

Introduction and Context

Automatic text summarization is the idea that using an algorithm, one can take an article, paper, etc. and create a summary of the piece that retains information and message that the piece is trying to convey. Work on solving this problem come fall into the field of natural language processing (NLP). Natural language processing is the field concerning using computers to successfully process the natural speaking and writing of humans. Some other problems of the field include speech recognition, translation, and natural language generation. Text summarization falls into NLP because of the need to correctly identify the keywords and patterns used in natural text in order to create a summary that correctly gives the proper information that the original text conveys.

After processing a document, there are two main schools of thought for how the summary is generated. The first is that the summary is extractive. That is, the program reads a document and pulls the entirety of what it thinks are the most important sentences. The program extracts the summary directly from the text without trying to simplify sentences or ideas into fewer words the same way a human might. One of the earliest forms of this type of summarization comes from a 1958 paper by Hans Peter Luhn. The idea of his paper is that authors will repeat important words heavily throughout a paper. This allows him to choose sentences with more repetition of the keywords and extract them to create a summary. Since Luhn's paper, there have been many attempts at extractive summarization, with the only main difference being in how each algorithm ranks sentences.

The other school of thought is returning an abstractive summary. An abstractive summary is the idea that after a text is processed, the algorithm can intelligently pick out the main ideas of the paper and generate a summary that condenses text in a natural way. This is attempting to mimic the way that humans naturally summarize a text when read. Using an abstractive summary method requires the algorithm to first process an extreme number of human created summaries for text in order to properly train the algorithm to attempt a natural summarization. One attempt at extractive summarization was TextSum by Google using TensorFlow. Google's research created headlines for news articles based on the first two sentences of the article. Their algorithm trained on 4 million pairs from the Gigaword dataset and TensorFlow's authors recommended

that it is only sufficiently trained after a million time steps, which Google achieved using roughly the equivalent of 7000 GPU hours. Overall abstractive text summaries are still in their infancy because of the immense time and hardware requirements needed for proper training in addition to the fact that it utilizes natural language generation, which is still an emerging field.

Formal Problem Statement and Solution

Given a document of arbitrary length, we wish to create a summary that extracts sentences from the document that sufficiently convey the message the original text intended to convey.

To do this we first partition the document into sentences where we let

$$N = \{n_1, n_2, \dots, n_m\}$$

be a text document with $n_1, n_2, \dots, n_m$ be sentences of the document in the order they appear. Next we let

$$W_m = \{w_1, w_2, \dots, w_i\}$$

be the set of words in each sentence with W_m being the set of words of the *m*th sentence and w_i being the *i*th word of the *m*th sentence. We can then say $w_{m,i}$ is the *i*th word of the *m*th sentence. Given a document *N*, we then extract a proper subset *S* of *N* where

$$S = \{n_j, \dots, n_k\}$$

and score the summary using the Rouge-1 metric.

The Rouge-1 metric is a similarity comparison that returns a score from 0 to 1 inclusive. For the Rouge-1 metric we get a recall and precision score by comparing the summaries generated by the algorithms to a "gold-standard" summary that was written by us. We can define recall as

$$Recall = \frac{\#\ unigrams\ occuring\ in\ both\ model\ and\ gold\ standard\ summary}{\#\ unigrams\ in\ the\ gold\ summary}$$

and define precision as

$$Precision = \frac{\#\ unigrams\ occuring\ in\ both\ model\ and\ gold\ standard\ summary}{\#\ unigrams\ in\ the\ model\ summary}$$

For both the recall and precision, the unigrams are the individual words in each summary as defined before.

Algorithm Implementations

Pre-processing

In order to properly analyze any given text, we must first pre-process the text by breaking it into sentences as well as words for the algorithm to use. In addition to that, we also remove stop-words (such as "the") in the cases of Luhn's algorithm and LexRank. Stop-word removal is necessary because they are words that tend to lack significant importance for conveying information while arbitrarily adding weight to a sentence by being so common. While it would be beneficial to also remove stop-words when pre-processing for our naïve algorithm, we choose not to because we are taking a naïve approach, and it is something one may easily overlook when they do not think too critically about how to solve the problem.

LexRank

LexRank was created around 2004 by Güneş Erkan and Dragomir R. Radev at the University of Michigan. The algorithm computes sentence importance using the concept of eigenvector centrality in a graph representation of sentences. Specifically, it uses a connectivity matrix based on intra-sentence cosine similarity for the adjacency matrix of the graph representation of sentences.

For a summary to be given, a text is first preprocessed into sentences with stop words removed. Then we create a graph where each sentences is a vertex of the graph. Edges are now created by comparing sentence similarity using an idf-modified-cosine equation. The equation will then be

$$\text{idf-modified-cosine}(x, y) = \frac{\sum_{w \in x,y} \text{tf}_{w,x}\text{tf}_{w,y}(\text{idf}_w)^2}{\sqrt{\sum_{x_i \in x}\left(\text{tf}_{x_i,x}\text{idf}_{x_i}\right)^2} * \sqrt{\sum_{x_i \in x}\left(\text{tf}_{y_i,y}\text{idf}_{y_i}\right)^2}}$$

where $\text{tf}_{w,s}$ is the number of occurrences of the word *w* in the sentence *s* and

$$\text{idf}_w = \log(\frac{N}{n_w})$$

where *N* is the number of documents in the collection and n_w is the number of documents in which the word *w* occurs. After all vertices and edges are created, Google's PageRank algorithm is applied to the graph. The idea of applying PageRank is that edges between sentences are votes for the vertices. This creates the idea that highly ranked sentences are similar to many other sentences and many other sentences are similar to a highly ranked sentence. We then create a summary by choosing the highest rated *x* sentences where *x* is defined by the user as the number of sentences wanted in the summary.

Luhn's Auto-Summarization Algorithm

Luhn's algorithm was first proposed in a 1958 paper written by Hans Peter Luhn. As stated before, Luhn's algorithm is based on the fact that humans are creatures of habit and will repeat keywords throughout a document. More importantly, he believes that the keywords an author uses is well defined and represents a single concept or notion. Even if an author tries to use reasonable synonyms for his or her keyword, they will eventually run out and fall back to using the best word that defines the notion, which will be the keyword that is repeated the most.

Running with the notion that an author will be repetitive with using a limited number of keywords to convey meaning, we can begin to rank sentences based on keyword frequency and proximity within a sentence. To determine sentence weight, we first look for significant words in a sentence, then take a subset of words in the sentence with the first and last word in the subset being a significant word. A subset is closed when four or five insignificant words are present before the next use of a significant word. Within the subset, we now count the number of times the significant word is present then divide by the number of total words in the subset. This number will be the weight given to that sentence. If a given sentence is long enough to contain multiple such subsets of significant words, we simply take the higher subset score as the weight of the sentence. To generate the auto-extraction, we only need to take the highest *x* sentences where *x* is a user defined number of sentences for summary length and putting the sentences back in the order they first appear. Besides just taking the highest rated sentences, it is also possible to break the text down into paragraphs and take the highest *y* sentences of each paragraph where *y* is *x* divided by the number of paragraphs. We could use this system since paragraphs are logical divisions of information specified by the author of the text.

Brute Force / Naïve Algorithm

This algorithm is one of the most naïve approaches to the problem. It also uses the idea that more important words will appear more frequently throughout the text, however it is very naïve in its implementation. Naivety comes from the fact that this algorithm does not address the issue of stop words and it makes use of no complex methods to determine meaning within a document.

Following the preprocessing of simply breaking the text into sentences, sentence weight is given by the summation of word score divided by sentence length. Words are scored across the whole document by taking a counter of every time each unique word in the document is repeated. The equation for sentence weight is

$$S_{weight} = \frac{\sum_1^i score_{w_i}}{|S|}$$

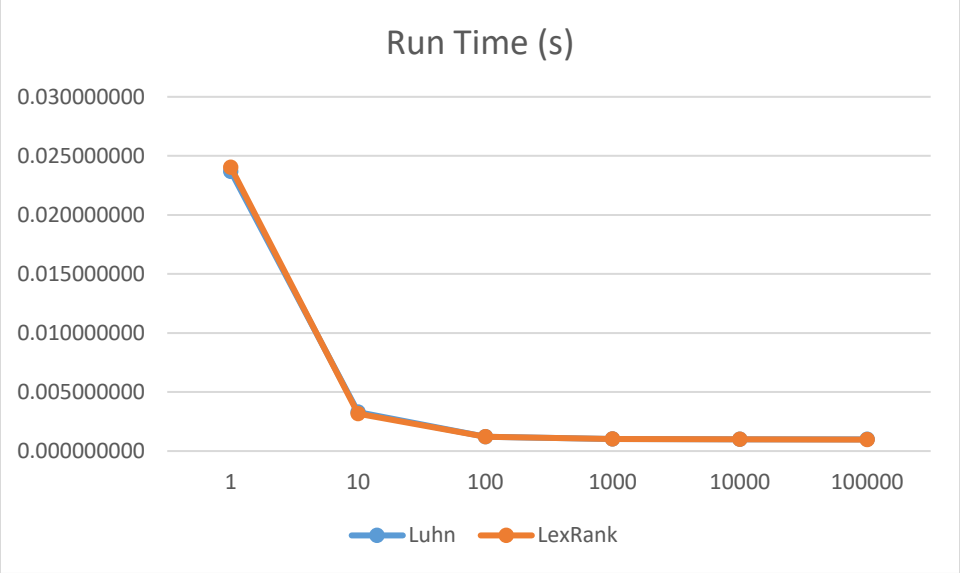
where w_i is the *i*th word of the sentence and $|S|$ is the cardinality of the sentence. *Here*, dividing by sentence length is a normalizing factor used to prevent sentences being chosen simply because they are much longer than others, rather than choosing sentences with more important

words. After calculating the weight of each sentence, the summary is given by choosing the *x* sentences with the highest weight and putting them in correct order.

Experimental Procedure & Evaluation of Algorithms

Run Time Procedure

To address the problem of inaccurate run times due to program overhead, inaccuracy of the time.clock() function, and excessive standard deviation, we decided to measure run time by looping the summarization portion of our code and dividing the result by the number of loops. To determine an appropriate loop count we ran both the Lexrank and Luhn's algorithms on one of our documents and collected the total time needed to loop the summarization code 1, 10, 100, 1000, 10000, and 100000 times. The time for an individual loop was then calculated for each. This was completed five times for each and the results were averaged. Based on this information we determined that running the summarization loop 1000 times would produce accurate results with very little benefit from increasing the number of loops any further.



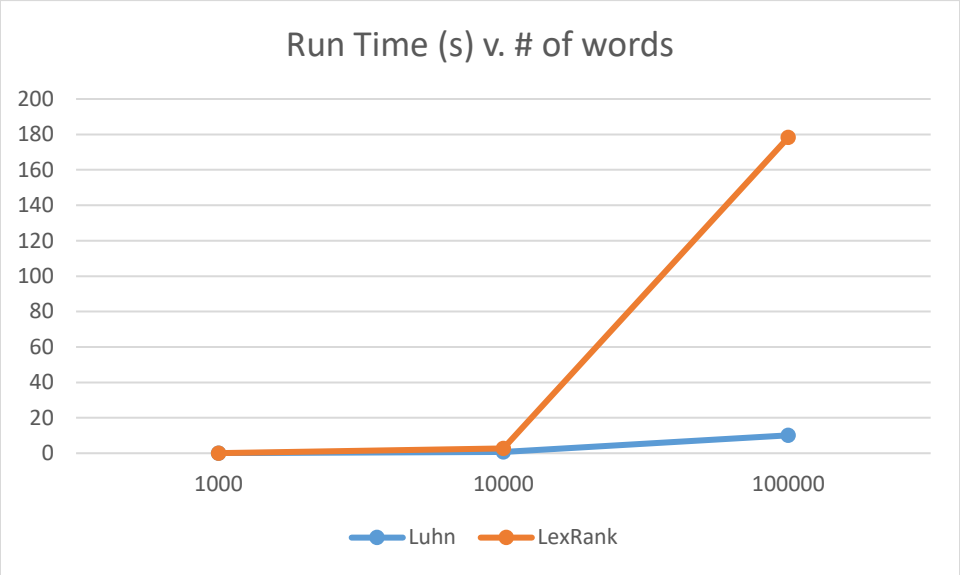
	Luhn	LexRank
1	0.023674194	0.024030157
10	0.003301801	0.003155208
100	0.001212647	0.001207262
1000	0.001027008	0.001013137
10000	0.000992096	0.000987296
100000	0.000988913	0.000970261

*all results are in seconds

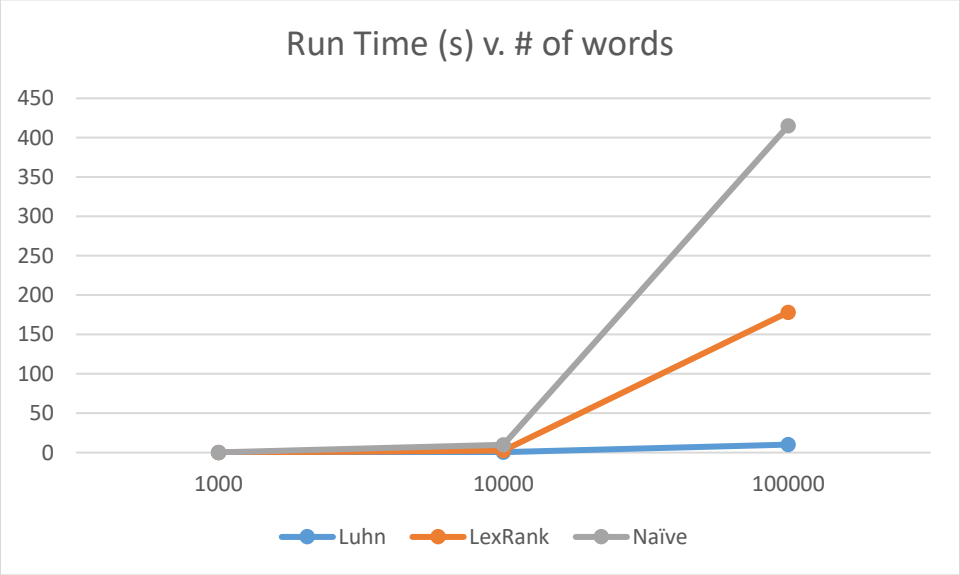
Run Time Evaluation

We then collected data on the amount of time it took each algorithm to generate a summary for articles of varying numbers of words. We used documents of 1000, 10000, and 100000 words to collect this data. We had originally planned to run this test on a one million word document as well but we had issues with overflowing the node array for Lexrank and the test would've taken an excessive amount of time to complete even a single run of the loop for Lexrank and our naïve approach.

Time needed for a single run of the summarization loop:



*Naïve approach left out to improve scaling for Luhn v. Lexrank



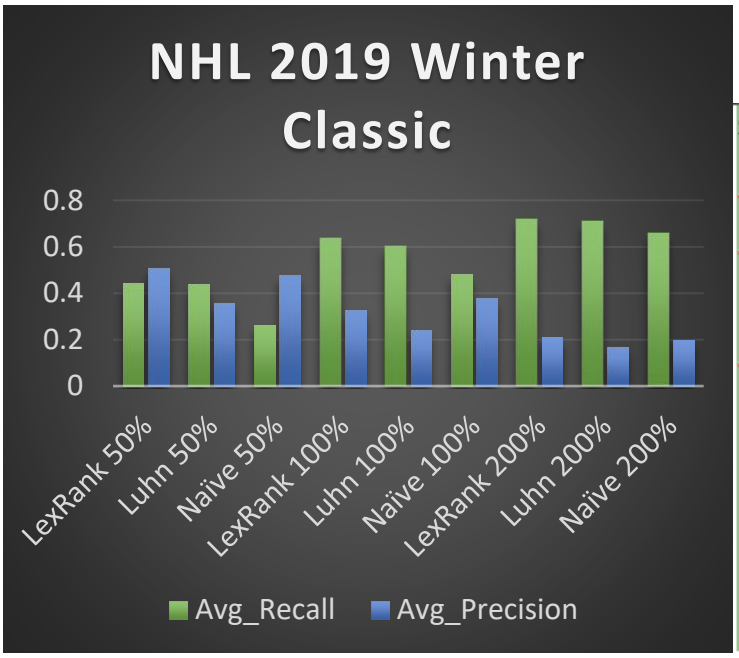
	Luhn	LexRank	Naïve
1000 words	0.048422	0.060678	0.205036
10000 words	0.659455	2.752696	9.679826
100000 words	10.1094	178.2651	415.1705

*all run times in seconds

Luhn's algorithm scales upward at a rate of around 12-15x increase in time for every 10x increase in the number of words being processed. This linear behavior is consistent with the theoretical expectation of Luhn's algorithm, where run time scales with complexity O(n). Lexrank exhibits an exponential increase in amount of time needed to run based on the length of the text being processed and greatly exceeds the amount of time needed to run Luhn's algorithm for the same text. This is due to every sentence being compared with every other sentence, which has a complexity of $O(n^2)$. Our naïve program scales upward at a rate of around 42-47x increase in time for every 10x increase in number of words in the text. We expect that this scaling would drop a little bit further for longer documents. The program compares the words in each sentence to a list of unique words used throughout the entire text. This list of unique words will, in most cases, increase at a decreasing rate as documents become very long. Even though our program exhibits linear growth, it is very greedy, taking even longer to run than Lexrank for all of the texts we tested. Eventually Lexrank would surpass our program's run time but the document being evaluated would have to be extremely long. At one million words Lexrank had issues with the array type overflowing but its theoretical run time should start to converge with our algorithm around this point.

Evaluation of Rouge-1 Scores

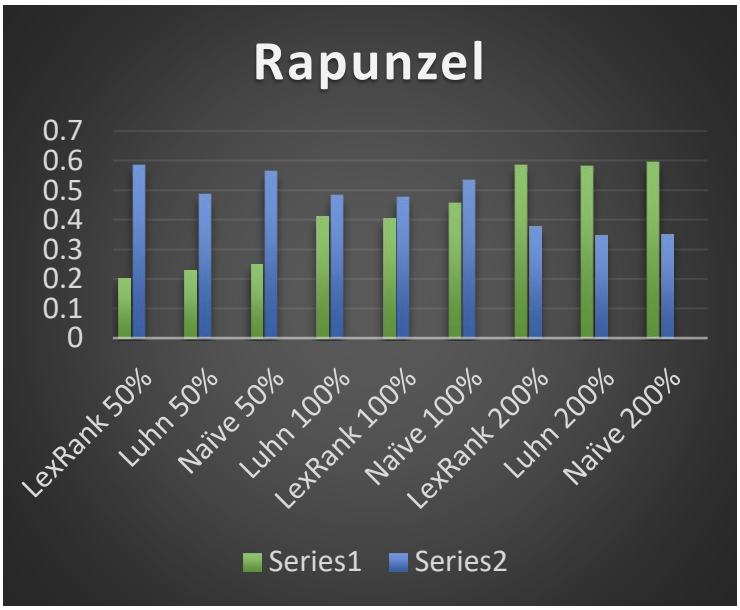
For our evaluation of the generated summaries we created a gold summary of each article, each having an equal number of sentences. We then generated a total of nine summaries for each article. For each program we generated summarizations half the length, equal to, and double the length of our gold summary. A high recall score suggests that the model summary has thorough coverage of words that would be included in the gold summary. However, a high recall score could also be caused by an excessively large generated summary. A high precision score suggests that the model summary accurately covers words used in the gold summary. Using a smaller generated summary typically will increase precision scores, but can also result in greater standard deviation between tests. For these reasons, we have included both the recall and precision scores in our results.



System Name	Avg. Recall	Avg. Precision
LexRank 50%	0.44444	0.50847
Luhn 50%	0.43704	0.35758
Naïve 50%	0.25926	0.47945
LexRank 100%	0.63704	0.32453
Luhn 100%	0.6	0.24179
Naïve 100%	0.48148	0.38012
LexRank 200%	0.71852	0.20815
Luhn 200%	0.71111	0.16901
Naïve 200%	0.65926	0.19911

2019 NHL Winter Classic:

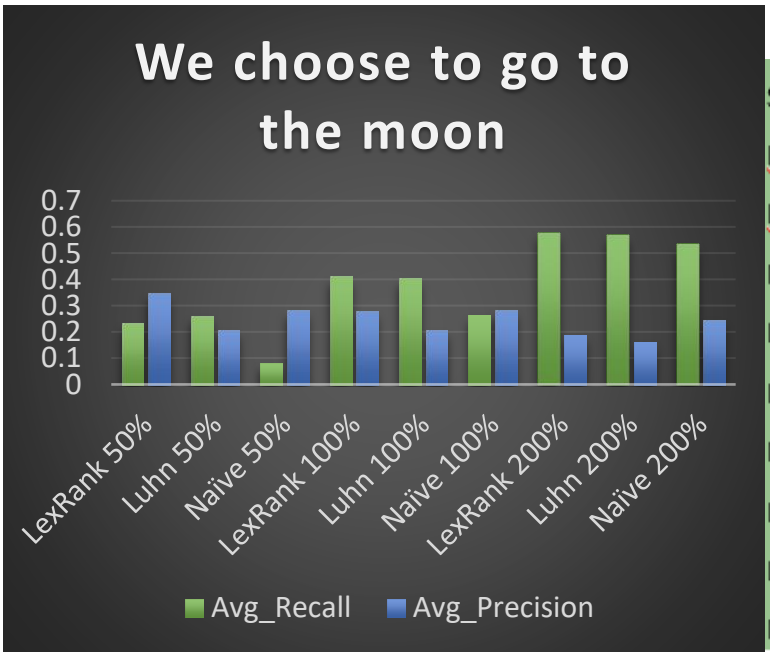
Luhn and Lexrank both have very similar recall scores for this article. Our naïve program has a significantly lower recall on the half-length generated summary but starts to approach the scores achieved by Luhn's and Lexrank as the generated summary gets longer. Lexrank and our naïve program stayed very close together on precision scores while Luhn's algorithm demonstrated the least precision in every test. Taking both recall and precision into account, Lexrank generated the most desirable summaries.



System Name	Avg_Recall	Avg_Precision
LexRank 50%	0.20096	0.58333
Luhn 50%	0.22967	0.48485
Naïve 50%	0.2488	0.56522
Luhn 100%	0.41148	0.48315
LexRank 100%	0.4067	0.47753
Naïve 100%	0.45455	0.53371
LexRank 200%	0.58373	0.37888
Luhn 200%	0.57895	0.3467
Naïve 200%	0.5933	0.35028

A short story – Rapunzel:

All three of our algorithms have very similar recall and precision scores for this article. Compared to the NHL article, the recall scores decreased and the precision scores increased across the board. Both of these trends may be in part due to the repetition of specific sentences in short stories. As expected, all recall scores increased and all precision scores decreased as our generated summary length increased. Taking both recall and precision into account, Lexrank generated the most desirable summaries.



System Name	Avg_Recall	Avg_Precision
LexRank 50%	0.23243	0.34677
Luhn 50%	0.25946	0.2069
Naïve 50%	0.08108	0.28302
LexRank 100%	0.41081	0.27839
Luhn 100%	0.40541	0.20776
Naïve 100%	0.26486	0.28324
LexRank 200%	0.57838	0.18838
Luhn 200%	0.56757	0.16006
Naïve 200%	0.53514	0.24627

John F. Kennedy – We choose to go to the moon:

For this article, Luhn's algorithm and Lexrank both have very similar recall scores. Our naïve program consistently produced the lowest recall scores but approaches the scores for Luhn's and Lexrank for longer generated summaries, as expected. The precision of our naïve program was surprisingly good compared to Luhn's and Lexrank for this article. Luhn's algorithm consistently resulted in the lowest accuracy for this article. Taking both recall and precision into account, Lexrank generated the most desirable summaries. Compared to Rapunzel and the NHL article, we had low accuracy from this text. We attribute this to John F. Kennedy's use of metaphors in his speech, which would convey an idea without using the same words.

Future Work

If we continued to work on this project there are a few areas we would like to further investigate.

For each algorithm we used we would like to collect a larger data set. We would like to run our programs on more documents and compare the results to multiple gold summaries per article. We would also like to collect more efficiency data at even intervals instead of at multiples. This should result in a more comprehensive view of how efficiency scales with word count for each program.

For the Lexrank algorithm we would like to investigate changing the data type used to store each word so we do not cause an overflow. Lexrank and our naïve program will eventually have intersecting run times but we are not able to investigate it unless Lexrank is adjusted.

For our naïve program we would like to bring the pre-processing up to par with Luhn's algorithm and Lexrank so that we can more accurately compare the summaries generated by each. We would also like to address its long run times and try to decrease these as much as possible. We would like to explore options such as implementing a hash table to store our word count information since it is commonly checked and the current implementation could be contributing to the long run times.

Questions

1. Why should we have pre-processed the text for stop-words in our naïve algorithm?
    a. Because stop-words are so common and lack significant meaning, they skew sentence weight in favor of stop-words rather than more meaningful words.
2. What is the difference between extractive and abstractive summarization?
    a. Extraction pulls full sentences directly from the text while abstraction uses machine learning to condense text in a heuristic manner.
3. What is the difference between recall and precision?
    a. Recall is the ratio between shared unigrams and a gold standard.
    b. Precision is the ratio between shared unigrams and the summarized model.
4. What does PageRank do in the LexRank summary?
    a. PageRank determines sentence weight by measuring the number of sentences that reference a given sentence
5. Why does Luhn's Algorithm only have $O(w)$ complexity?
    a. Because it only counts repetition within each sentence rather than compared to the document as a whole.

Works Cited

1. https://en.wikipedia.org/wiki/Natural_language_processing
2. https://en.wikipedia.org/wiki/Automatic_summarization
3. https://rare-technologies.com/text-summarization-in-python-extractive-vs-abstractive-techniques-revisited/
4. http://courses.ischool.berkeley.edu/i256/f06/papers/luhn58.pdf
5. https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume22/erkan04a-html/erkan04a.html