MEMORIAL UNIVERSITY OF NEWFOUNDLAND
FACULTY OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Text Summarization

Computer Science 4750

Jacob House Noah Gallant

Friday, November 30th, 2018

Contents

1 Topic and Motivation

Motivation to study computerized text summarization — referred to by the term *automatic abstracting* by those in the field — stemmed from curiosity about the mechanisms used in an automatic abstraction PowerShell script found years ago on the Internet [1]. Further research has shown automatic abstraction's usefulness in many interesting fields, including but not limited to the legal and medical professions, scholarly research, and search engine result sorting and summarization.

2 Background Research

3 Implementation and Program Structure

The implementation chosen consisted primarily of the graph method. Three Python classes were written to implement this approach: the `Node`, the `Edge`, and the `Text` class. These are called by a Python program `Summarizer.py` which has the role of opening the input text file and ensuring the correct format is used before performing the summarization.

3.1 The `Text` Class

The role of the `Text` class is to perform all operations on the text. This includes any pre-processing required, the initial object creation, determining inter-sentence relationships, as well as using the determined relationship information to form a reasonably accurate and concise summary.

When first initializing the text, the text to be summarized must be passed into the constructor of the `Text` class. This text is then saved as in instance variable for future use, and proceeds to undergo pre-processing. Pre-processing is done by the `preProcessing()`

method, and is tasked with splitting the text into sentences.

3.1.1 The `preProcessing()` Method

Pre-processing is a crucial step in achieving a high quality summary. Since pre-processing is the first operation performed on the text, the method in which pre-processing is conducted can have a significant bearing on the quality of the summary. The goal of the pre-processing in this application is primarily to split the text into sentences. This is done by considering sentence terminating symbols including '.', '!', and '?'. This also considers if a sentence contains a quote, and any other situation in which a terminating symbol should be ignored.

The secondary operation of the pre-processing in our application is to replace special Unicode characters with their plain text equivalent, or, in extreme edge cases, to simply remove the character. Many common text editors use Unicode characters as opposed to the plain text characters for many text symbols. Some of these symbols include typographer's quotation marks, accents (*e.g.,* ç, ä, ñ, etc.), en dashes and em dashes, as well as non-Latin characters (*e.g.,* ø, æ, ß, etc.).

Operating on Split Sentences

Once split, the text is processed by creating a `Node` object for each sentence. This node contains the original sentence, a list of all the words in the sentence which have been passed through a lemmatizer [nltk], a list of edges which are connected to the node, as well as the sentence number to keep track of the location of the sentence in the original text. During the `Text()` constructor which parses the sentence, `Edge` objects are created to connect sentences that are adjacent to each other in the original text. These adjacency edges are the first edges to be created in the graph.

3.1.2 Creating the Dictionary

After all the sentence nodes have been created, and a list of words in each sentence node has been processed, the instance of `Text` will then proceed to loop through each sentence node to create a complete dictionary of all the words found in the text. This dictionary will not only store the available words, but will also store all the sentence nodes in which the word is contained. The goal of creating this dictionary is to decrease the computational complexity in determining the relationships between sentences.

3.1.3 Creating Edges

By creating a dictionary that contains words and the nodes they are contained within, it then suffices to loop through all the words (*i.e.,* dictionary keys) and, when a word is contained within more than one node, to link these nodes with an edge. Each sentence `Node` object will thus contain both proximity `Edges` and `Edges` associated with common words.

This step also offers the opportunity for further improvement to the summary. The more relation edges that are made, using different criteria, the better the summary. Therefore by only counting the number of word relations, we limit the quality of our summary. Some additional criteria to be added to improve the summary include quotation detection, statistics, names, negations, and modifiers. Of course, this means creating a hybrid graph-and-text-element approach in which sentences accumulate weight (*i.e.,* their importance score) from inter-sentence content relationships as well as in-sentence text elements that do not associate one sentence with another.

3.1.4 Creating a Summary

The next step is to create a summary using the nodes and edges created in the previous steps. After this processing is complete, the software prompts the user to input how many sentences the summary should contain and supplies a recommended number of sentences to choose in case the user is not completely aware of the length of the text supplied. This recommendation R is computed using Equation (1), where N is the total number of sentences in the text.

$$R := \begin{cases} \frac{1}{2} \cdot N, & \text{if } N < 10 \\ \frac{1}{3} \cdot N, & \text{if } N \geqslant 10. \end{cases} \tag{1}$$

Then, the R highest-ranking `Node` objects are sorted according to their associated sentence's position in the original text and the stored sentences are printed in bullet-point format.

3.2 Node Class

The `Node` class is the application's representation of a sentence. The `Node()` constructor takes in a sentence as a string, and will immediately split the string into words which are then saved in a list.

To reduce the time complexity of the execution, instead of performing lemmatization in the pre-processing stage as might be expected, it was decided to perform this step at the same time the words in the sentence were being divided. This was chosen to prevent accessing a word more than was required.

The final task of the `preProcessing()` [Wait... we're still talking about `preProcessing()`? I thought we were in the `Node` class now.] method is to use a lemmatizer to remove word endings. When comparing words later in the application to determine relevant sentences, this will improve the results as it will more accurately represent similar words, rather than counting the same words with identical endings as different words and therefore misrepresenting the relations in the text.

3.3 Edge Class

4 Optimization

References

[1] Prateek Singh. *PSSummary*. 2018. URL: https://github.com/PrateekKumarSingh/PSSummary.

Appendix

A Demonstration

A.1 Heterogeneous Computing

One test case was to give the application an except (included below) of a research paper on heterogeneous computing [noahpaper].

Heterogeneous Computing Test File

> The usage of the CPU as a generalized process, and the GPU as an accelerator to the CPU has been a common practice just before and continuing after 2010. This configuration is what's considered as general-purpose computing on a GPU. This configuration offers significant performance benefits in a system, particularly in data-parallel, and computationally heavy process. Heterogeneous system architecture is used to leverage the value of each unique device, The CPU being well designed for tasks were latency is critical, while the GPU is the choice processor in throughput-oriented tasks. This creates two different classifications; a latency compute unit (LUT) being a general CPU, and a throughput compute unit being a general GPU. Both processing units are capable of performing the same calculations or tasks, however the different architectures have different strengths and weaknesses. The strengths and weaknesses of each processor boils down to their internal structure. CPUs are composed of a few large, flexible, and fast clocked cores, while GPUs are built using thousands of cores that are both smaller and slower, but are highly parallelized. In addition to performance, energy efficiency is also an important factor while GPUs have considerably greater computational power than CPU's, the have approximately the same energy cost. For example, when comparing the Intel Xeon E7 CPU, 150 watts delivers around 100GFlops/s, while a small increase of energy to 250 Watts in the NVIDIA GK110 provides 1.3 TFlops/s. At the time of writing of this article in 2015, these were both state of the art processors.

For this file we requested that the application return five sentences. The sentences returned were:

1. The usage of the CPU as a generalized process, and the GPU as an accelerator to the CPU has been a common practice just before and continuing after 2010.
2. This configuration offers significant performance benefits in a system, particularly in data-parallel, and computationally heavy process.

3. Heterogeneous system architecture is used to leverage the value of each unique device, The CPU being well designed for tasks were latency is critical, while the GPU is the choice processor in throughput-oriented tasks.
4. This creates two different classifications; a latency compute unit (LUT) being a general CPU, and a throughput compute unit being a general GPU.
5. CPUs are composed of a few large, flexible, and fast clocked cores, while GPUs are built using thousands of cores that are both smaller and slower, but are highly parallelized.

[Reflect on this summary here.]