Improved Summary Routine Using MEAD

Austin Almond
University of Washington
aalmond@uw.edu

Corey Moore
University of Washington
cogmoore@uw.edu

Arun Sen
University of Washington
arunsen@uw.edu

Abstract

Our team designed and built an extractive summarizer in Python 3. We used MEAD for content selection, chronological and topic similarity for sentence ordering, and CLASSY sentence compression for sentence realization. The quality of the output summaries were then analyzed using ROUGE metrics.

1 System overview

Our goal for the initial state of the project was to create a system using a proven architectural model and pre-written software packages to establish a baseline level of performance. This baseline model then served as the platform by which later attempts at improvement could be judged. Using the simple three-part structure of extraction, ordering, and realization, we assigned our individual tasks for the project, and then designed our software architecture with a main method (Fig 1) that branched off into its separate components, so we could each work independently on them.

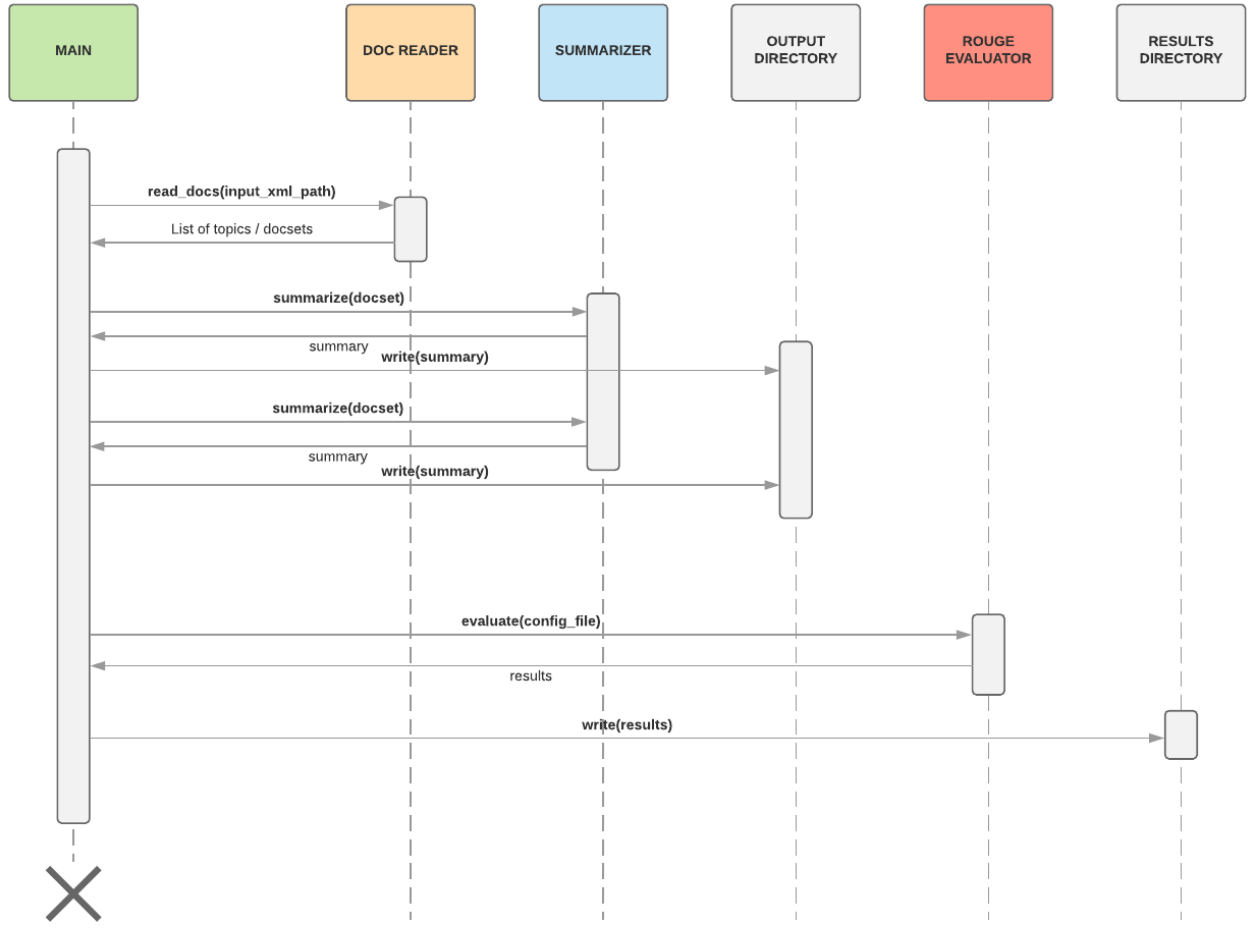

Fig. 1. Main method overview.

2 Approach

We decided to write summarizer in Python 3, and used the GitHub environment as the main repository for our code, which enabled us to work on separate components and establish separate development branches as the components came together. The primary sentence extraction module was MEAD, a Perl script package, and we supplemented this with various NLTK tokenizers (Radev 2002).

Since the members of our team were all working remotely and in different time zones, real-time one-on-one communication channels would not be sufficient. Our first line of communication was a channel that we formed in Slack, where comments left by one person would persist for many days afterward, and conversations could occur in slow motion as and when we were available. The GitHub platform also enabled some of our team communication, giving us the ability to comment on modules and even on specific lines as our code progressed.

2.1 Document reader

The DocumentReader class's constructor accept base file paths for the three document formats: AQUAINT, AQUAINT-2, and ENG-GW The read_docs method accepts an XML filename, and returns a data structure mapping topic IDs to document sets. The XML filename points to file that lists all of the necessary document IDs for each topic.

For each set of document IDs, the document reader resolves the path of the XML file containing the document. It then parses it into a Python dictionary, keeping a consistent dictionary format regardless of the document format.

The text of the document is parsed into plain text as best as possible, using either **lxml** for XML files or **html.parser** for SGML (Behnel 2018, Python Software Foundation 2018). AQUAINT-2 files use XML, while AQUAINT and ENG-GW use SGML (Linguistic Data Consortium 2002).

Each document's data is parsed into a dict with the keys *type*, *keywords*, *headlines*, *datelines*, and *paragraphs*. Each of these dicts is combined into an array. This array is placed at key *docset*, which is placed alongside a topic's *id*, *title*, and *category*. The method then returns an array with this data combined for all topics.

2.2 Content selection

Preprocessing begins with abbreviation removal using a short rule-based script containing a list of sample abbreviations that are usually marked with additional punctuation (such as Dr., Mr., Mt., et al). The script makes the naive assumption that any abbreviation in the sentence that is found on the list does not end the sentence. The next stage is coreference resolution, which was done using the Stanford NLP package (Stanford CoreNLP, 2018), leveraged in our code through a Python wrapper. The coreference resolution makes only one replacement (first mention) per sentence to avoid redundancy. These two elements were ordered because the reverse sequence would take co-references that included punctuation (such as *Dr. Smith*) and alter them beyond recognition.

Further in our preprocessing regime, sentence separation and tokenization are handled using Stanford NLP. The target sentences from each cluster are then selected using MEAD. As for post-processing, stemming was done using NLTK Snowball Stemmer, a list of English-language stop words from the NLTK corpus was used to filter the text, and all remaining punctuation was removed.

2.3 Information ordering

Information ordering uses a two different methods--chronological and topical closeness. This approach of consulting various experts and weighting the results was inspired by Bollegala et al, 2012. The chronological expert assembles the sentences in chronological order based upon the order of the documents in the directory, and the order of the sentences within each document. The topical closeness expert uses the Wordnet package was used to calculate the similarity of two sentences (Bogdani, 2016). In this approach,

first the two most similar sentences were selected. Then the remaining sentences were added to the list of ordered sentences based on their similarity to the first or last sentence already in the list.

2.4 Content realization

Content realization is now fully implemented, using the CLASSY sentence compression approach discussed in Conroy, et. al, 2006. The Dunlavy et. al, 2003 paper was also consulted in the course of designing this component.

Non-essential phrases, such as datelines and leading conjunctions, are dropped from the sentences within the *sentence_realization* module. Extraneous spaces and punctuation are also removed. This module is run before the sentences are fed into the MEAD content selector, and it is run again just before the ordered sentences are written to the system's output. This ensures a well-formatted set of sentences in all cases.

3 Updated ROUGE Results

The system was evaluated with the ROUGE metric. We found that D4 presented a consistent improvement over D3 for both ROUGE-1 and ROUGE-2 in both datasets.

ROUGE test	D3	D4
ROUGE-1	0.19117	0.22656
ROUGE-2	0.04327	0.06229

Table 1. Comparison of D3 and D4 ROUGE scores (Average_R) on the **devtest** dataset.

ROUGE test	D3	D4
ROUGE-1	0.14382	0.29362
ROUGE-2	0.03803	0.08335

Table 2. Comparison of D3 and D4 ROUGE scores (Average_R) on the **evaltest** dataset.

4 Discussion

4.1 Error Analysis

Sentence segmentation errors of an as-yet unknown origin persist in this iteration. For example, in docset 1101, we find this so-called single sentence: *Charles Carl Roberts IV, 32 husband is loving, supportive, thoughtful, all the things you'd always want and more. Charles Carl Roberts IV, 32, held a steady job working nights driving a truck that picked up milk from area dairy farms.* This is a run-on sentence caused by segmentation failure. The tokens to either side of the full stop are an adjective and a replaced co-reference, which suggests the co-reference resolution may be interfering with sentence segmentation.

On closer inspection, the summarizer also produces this "sentence": *A new study by British researchers reveals that those women in the study group who slept less than or equal to 5 hours a night health is much more at risk from sleep deprivation than men's. The study, carried out by researchers at Warwick Medical School at the University of Warwick, Britain, involved 5,766 volunteers 4,199 men and 1,567 women from 20 London-based civil service departments, according to a press release from the university on Friday.* To either side of the full stop is a possessive pronoun and a noun phrase not associated with a pronoun coreference replacement. It may be that the NLTK segmentation module fails to recognize the token [`men's.`] as the end of a sentence, possibly due to the proximity of the apostrophe.

4.2 Future Enhancements

The Info Ordering module uses a two-expert model, defining each of the Chronological and Similarity based experts at 50%. We intended to optimize this weighting, however there was not enough time to experiment with other values besides 50/50. Further experimentation may yield better information ordering results.

Only some parts of the CLASSY 2006 sentence trimming algorithm were implemented. The paper described several other ways of trimming sentences, which were too complex for us to implement in a reasonable time frame.

Implementing more of these would likely have improved our scores.

4.3 Lessons Learned

We learned some valuable lessons about summarization during the construction and assembly of our baseline summarizer, and also about our collaborative working process. When we started work on the project, there was less emphasis on linguistic analysis and more about finding the appropriate way to stitch together different elements that were pre-written to accomplish different tasks. As work on our system progressed, linguistic analysis came into play considerably more.

We discovered during our strategic planning sessions that it was important to confirm the precise format of the inputs and outputs for each component: what data it required in order to operate, and how it would deliver its results for the next summarization step. Some of our code, originally written in the hopes of using a Python dictionary structure called a JSON, had to be redesigned when it became clear that MEAD's desired input format was a specially-formatted data file. The JSON approach had to be abandoned.

5 Conclusion

We first achieved a baseline level of performance from our summarizer, successfully welding together our code in approximately the same structural sequence that we had initially intended. The finished system produced summaries with considerably higher ROUGE scores compared to the initial baseline, which is evidence of the fine tuning and enhancements that were implemented. While time did not allow us implement all the features we had hoped to incorporate, we found that our additions to the initial MEAD structure did eventually pay dividends.

At least some of the improvements we added, in particular our attempt at sentence segmentation, were not unalloyed successes. Examining the final produced summaries showed that some sentence segmentation errors remained. Determining the source of these errors will take a systematic comparison of our summary results with, and without, the various added modules that could have precipitated those segmentation errors (such as abbreviation removal and coreference resolution).

6 References

Behnel, Stefan. "Lxml - XML and HTML with Python." Lxml - XML and HTML with Python, 21 Mar. 2018, http://lxml.de/.

Bogdani. "Compute sentence similarity using Wordnet." NLP for Hackers, 25 August 2016. https://nlpforhackers.io/wordnet-sentence-similarity/.

Bollegala, D., Okazaki, N., and Ishizuka, M. (2012) A preference learning approach to sentence ordering for multi-document summarization.

Conroy, J. M., Schlesinger, J. D., and Goldstein, J. (2006). Classy tasked based summarization: Back to basics.

Linguistic Data Consortium. "The AQUAINT Corpus of English News Text." AQUAINT Text Corpus, 2002, http://catalog.ldc.upenn.edu/docs/LDC2002T31/

M Dunlavy, Daniel & Conroy, John & Schlesinger, Judith & A Goodman, Sarah & Ellen Okurowski, Mary & O'leary, Dianne & Halteren, Hans. (2003). Performance of a three-stage system for multi-document summarization. IEEE Intelligent Systems & Their Applications - IEEE INTELL SYST APPL.

Python Software Foundation. "20.2. Html.parser - Simple HTML and XHTML Parser." Python 3.4.8 Documentation, 4 Feb. 2018, http://docs.python.org/3.4/library/html.parser.html.

Radev, D. et al. 2002. MEAD - a platform for multidocument multilingual text summarization. Web. Accessed 4/26/2018. http://clair.si.umich.edu/~radev/papers/lrec-mead04.pdf.

Stanford CoreNLP. "CoreNLP version 3.9.1 Stanford CoreNLP– Natural language software." Stanford CoreNLP– Natural language software, 2018. https://stanfordnlp.github.io/CoreNLP/