Baseline Summary Routine Using MEAD

Austin Almond
aalmond@uw.edu

Corey Moore
cogmoore@uw.edu

Arun Sen
arunsen@uw.edu

Abstract

Our team designed and built a baseline extractive summarizer in Python 3 using MEAD for content selection, basic chronological sentence ordering, and by realizing the target sentences as written. It will serve as the foundation for future attempted improvements.

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 could then serve 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 that branched off into its separate components so we could each work independently on them.

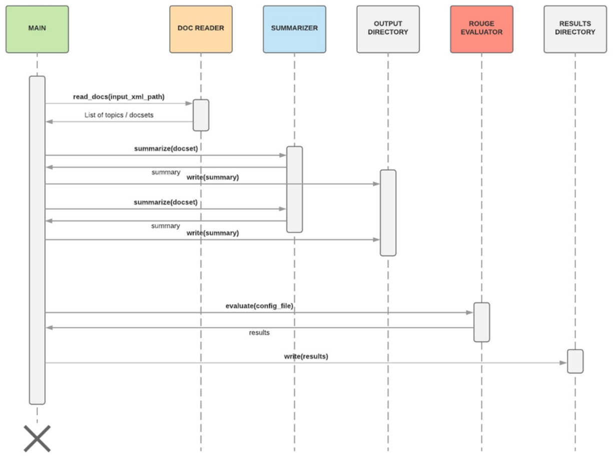

Fig 1. Main method overview.

2 Approach

We decided to write summarizer in Python 3, because it was a language with which we were all familiar, and composed together in the GitHub environment, 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

Selecting the target sentences from each cluster was done through MEAD. Our initial intent was to conduct preprocessing on the input data using NLTK sentence segmentation, tokenization, and stemming. A list of English-language stop words from the NLTK corpus was used to filter the text, and punctuation was removed. This would have allowed us to tinker with the preprocessing methods, adding components in the next deliverable stage (such as coreference identification and named entity recognition) to improve performance.
In the current iteration, however, these modules are applied as post-processing to the extracted sentences. Two versions of the output for each cluster are maintained — one of the text in a sentence-segmented state, and another of each sentence in a simplified state — and passed along to the info-ordering and sentence-realizing routines.

2.3 Information ordering

For our initial version of the summarizer, we decided to use simple chronological ordering as a baseline. The routine 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. In future deliverables we will be able to expand upon this process to see whether performance improvements can be found.

2.4 Content realization

At this stage, the content realization module simply takes the sentences verbatim from the information ordering stage, using the segmented sentence data. The simplified sentence data is not used. Because the first stage of our design does not do any named entity recognition or coreference resolution, no rewriting of the sentences is necessary. This is our baseline level of performance, upon which we hope to expand in future stages.

3 Base results

The system was evaluated with the ROUGE metric. The F-score for ROUGE-1, compared to the gold standard summary, is only 0.149, which gives us room for improvement in our second iteration of the summarizer.

ROUGE test	Recall	Precision	F-score
ROUGE-1	0.146	0.152	0.149
ROUGE-2	0.040	0.041	0.040
ROUGE-3	0.014	0.014	0.014
ROUGE-4	0.005	0.005	0.005

Fig 3. Baseline results for our system using the ROUGE evaluation metric.

4 Discussion

We learned some things about summarization during the construction and assembly of our baseline summarizer, and also about our collaborative working process. In many respects, the exercise was less about linguistic analysis and more about finding the appropriate way to stitch together different elements that were pre-written to accomplish different tasks.

As an example, 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.
Although we discussed our system architecture, we did not discuss in detail the precise order of our components or our strategies for measuring future improvements. The content selection extracts the sentences using MEAD as its first stage, without any attempt at preprocessing. A preprocessing step would have enabled us to make comparisons, by systematically altering MEAD's input stream to see which configuration of preprocessing steps was most efficacious.

5 Conclusion

We obtained a baseline level of performance from our summarizer, successfully welding together our code in approximately the same structural sequence that we had initially intended. Future steps for our project are likely to include introducing a preprocessing stage, attempting disambiguation of pronouns and coreferences, sorting out named entities, and engaging in a more robust form of information ordering and sentence realization. Now that the initial structure is established and functional, it should be a matter of adding additional computational steps and comparing our results.

6 References

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

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

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.