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 initially 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. The latest release features some enhancements, namely coreference resolution and similarity-based information ordering.

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 (Fig 1) that branched off into its separate components so we could each work independently on them.

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.

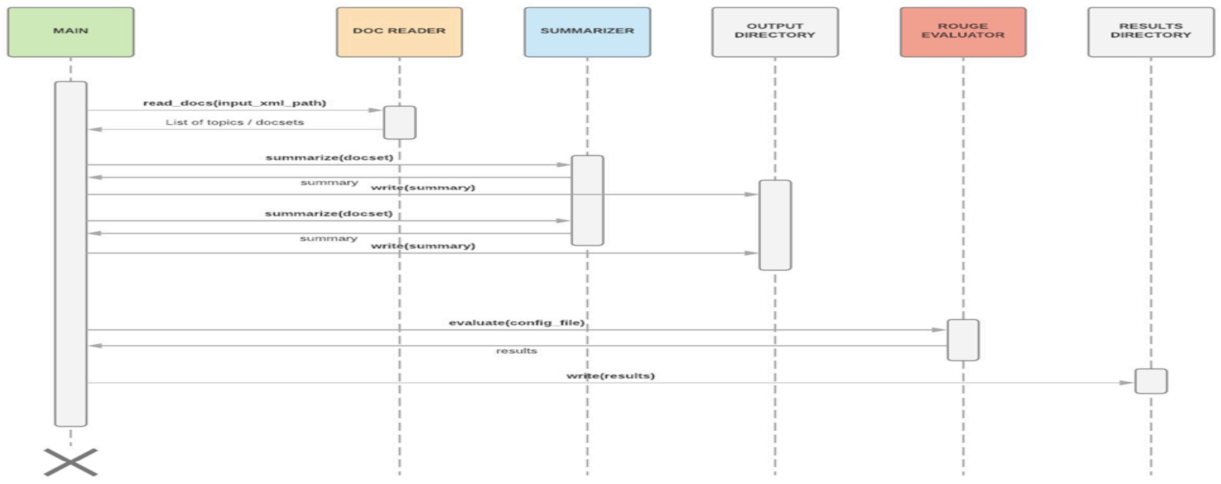

Fig 1. Main method overview.

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

The latest iteration now features a new preprocessing step. The most significant enhancement is coreference resolution, which was done using the Stanford NLP package (Stanford CoreNLP, 2018), leveraged in our code through a Python wrapper. Additionally, 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, and a list of English-language stop words from the NLTK corpus was used to filter the text, and punctuation was removed. 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. The latest iteration now features a second reordering step based on sentence similarity. Instead of using cosine similarity measures, however, the Wordnet package was used to calculate the similarity of two sentences (Bogdani, 2016). To order the sentences, 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

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. We plan to develop this module for the next release.

3 Base results

The system was evaluated with the ROUGE metric. The initial F-score for ROUGE-1, compared to the gold standard summary, is only 0.149 (Table 1); an improvement was made in the latest release, bringing that score up to 0.197 (Table 2). However, the increase in scores appears to be related to fixing several bugs which were preventing some summaries from being generated.

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

Table 1. Baseline results for our system using the ROUGE evaluation metric (overall).

ROUGE test	Recall	Precision	F-score
ROUGE-1	0.191	0.204	0.197
ROUGE-2	0.043	0.046	0.045
ROUGE-3	0.013	0.013	0.013
ROUGE-4	0.005	0.005	0.005

Table 2. Results for our updated system using the ROUGE evaluation metric (overall).

	D2 R	D2 P	D2 F	D3 R	D3 P	D3 F
ROUGE-1	0.1448	0.1500	0.1470	0.1197	0.1275	0.1230
ROUGE-2	0.0403	0.0413	0.0407	0.0292	0.0309	0.0299
ROUGE-3	0.0139	0.0141	0.0140	0.0096	0.0101	0.0098

Table 3. Comparison of D2/D3 ROUGE scores

To better understand the results in relation to our changes in content selection and information ordering, we also compare scores from summaries generated in both the initial and latest releases (Table 3). This comparison actually shows a drop in ROUGE scores. After our error analysis, we suspect this decline is related to the coreference resolution for two reasons. First, the coreference resolution introduces redundancy in the text, which is not yet handled because the sentence realization module is still under development. Second, we found a sentence segmentation issue after coreference resolution is performed. These issues will be addressed in our final release.

4 Discussion

We learned some valuable lessons about summarization during the construction and assembly of our baseline summarizer, and also about our collaborative working process. In many respects, the initial release 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. In the latest release, 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. As planned, our latest release features a preprocessing stage which includes coreference resolution. Information ordering was also updated to include a Wordnet-based reordering in addition to the chronological approach. Our next release will focus on removing remaining bugs in the system, implementing the content realization module, and final tuning of the overall system.

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/.

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.

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