Published in Towards Data Science

Andrew Udell Follow
Nov 23, 2020 · 5 min read · · Listen

Save

Simple Text Summarization in Python

A Quick Guide on Writing your Own tl;dr

Photo by Aaron Burden on Unsplash

Project Gutenberg offers over 60,000 full length books. Wikipedia contains over 55 million surface.

The ocean of written material creates a paradoxical problem: because there's an overabundance of information, finding relevant information becomes more difficult.

Automatically generating text summarizations may help the problem. Instead of leaving users to skim through large walls of text, presenting a brief summary provides the user with key pieces of information so they can make a more informed decision to continue reading without wasting the time to parse the text themselves.

Approaches

There are two main approaches to text summarization: extraction-based and abstraction-based.

- **Extraction-Based:** This approach searches the documents for key sentences and phrases and presents them as a summary. The simplicity of extraction-based summarizations make them easy to implement, but risk the danger of taking quotes out of context.
- **Abstraction-Based:** This approach attempts to synthesize a completely new sentence or phrase to summarize a text. While abstraction-based provides a more dynamic and flexible method for summarization, it necessarily must deal with the complexity of grammar and syntax, which often makes it more difficult to implement.

Given that extraction-based models aim to take key sentences out of the text based on some criteria, they're more dynamic in the scope of languages they can handle. For example, an extraction-based method in English may be used to provide summarizations in Spanish with very little modification.

An abstraction-based model, however, would lack the same portability, because it can only handle grammar in one language at a time.

Extraction-Based Summarization in Python

To introduce a practical demonstration of extraction-based text summarization, a simple algorithm will be created in Python. To evaluate its success, it will provide a summary of this article, generating its own "tl;dr" at the bottom of the page.

Broadly, the algorithm will go through all the words in a text, rank them based on importance, and return the sentences with the most relevant words in them.

The first few lines cover basic packages that need to be imported. The most important of these is NLTK, which is Python's most famous natural language processing library.

The string library will help deal with string operations and heapq provides a quick function for finding the n largest number in a set.

```
# Declare a variable for text
text = ""

# If the length of the text is greater than 20, take a 10th of the sentences
if text.count(". ") > 20:
  length = int(round(text.count(". ")/10, 0))

# Otherwise return five sentences
else:
  length = 1
```

The text variable will store the actual text to be summarized. In this example, it's left blank for brevity.

The next lines introduce the variable length, which defines the length of the summary in number of sentences provided. For example, if a length of 3 was selected, the algorithm would provide the 3 most relevant sentences as a summary. If the length of the text isn't know ahead of time, choosing a dynamic approach provides a lot of flexibility.

To count the number of sentences in the text, the number of periods followed by a space is counted. This works well in grammatically correct sentences, but a more informal body of text (such as comments or reviews) may require a different method.

If there are more than 20 sentences, then only approximately 10% of the sentences are returned as the summary. If there are fewer than 20 sentences, a single sentence is returned.

```
# Remove punctuation
nopunc = [char for char in text if char not in string.punctuation]
nopunc = ''.join(nopunc)

# Remove stopwords
processed_text =[word for word in nopunc.split() if word.lower() not in
nltk.corpus.stopwords.words('english')]
```

The text is then processed to focus on the key symbols. The first two lines remove any punctuation in the text

```
# Create a dictionary to store word frequency
word_freq = {}

# Enter each word and its number of occurrences
for word in processed_text:
  if word not in word_freq:
    word_freq[word] = 1
  else:
    word_freq[word] = word_freq[word] + 1
```

Next, a dictionary is created. For every word in the text, the word is stored in the dictionary as a key value and its number of occurrences as its value.

This step functionary provides a list of words in the text and the number of times they appear.

```
# Divide all frequencies by max frequency to give store of (0, 1]

max_freq = max(word_freq.values())

for word in word_freq.keys():
  word_freq[word] = (word_freq[word]/max_freq)
```

To make the number of occurrences unitless, all occurrences are divided by the maximum number of occurrences.

This gives all the words a score between 0 and 1, which ranks the importance of the word within the text. Words which appear more frequently are given a higher score and are presumed more important.

```
# Create a list of the sentences in the text
sent_list = nltk.sent_tokenize(text)

# Create an empty dictionary to store sentence scores
sent_score = {}
for sent in sent_list:
  for word in nltk.word_tokenize(sent.lower()):
    if word in word_freq.keys():
      if sent not in sent_score.keys():
        sent_score[sent] = word_freq[word]
      else:
        sent_score[sent] = sent_score[sent] + word_freq[word]
```

the word's previously defined importance is referenced and then added to the sentence's score.

The more important words a sentence has, the higher the score that's assigned to that sentence.

Note that this necessarily creates a bias for longer sentences, which isn't inherently a bad thing. In fact, returning longer sentences usually provides a little more context than shorter sentence, making the overall summary more comprehensible.

```
summary_sents = nlargest(length, sent_score, key = sent_score.get)

summary = ' '.join(summary_sents)
print(summary)
```

Finally use nlargest to return the n highest scored sentences as defined by the length variable. This returns the most important sentences based on word frequency as a summary.

tl;dr Conclusions

An extraction-based text summary is relatively easy to implement. Using the relative importance of words, as measured by the number of times they appear in a text, a summary may be generated from the most important sentences.

To evaluate this method, the below is the generated summary of this article:

```
Instead of leaving users to skim through large walls of text, presenting a
brief summary provides the user with key pieces of information so they can
make a more informed decision to continue reading without wasting the time
to parse the text themselves. Broadly, the algorithm will go through all the
words in a text, rank them based on importance, and return the sentences
with the most relevant words in them.
```

Not too bad.