TEXT SUMMARIZATION: NATURAL LANGUAGE PROCESSING

CS60075

Group 20: Members

Pradipto Mondal, 20EC30032

Madiha Hanifa, 20MF10018

Pratyush Garg, 20IE10023

Zaid Irfan Ali, 20EE30034

Indian Institute of Technology, Kharagpur

AUTUMN 2021-22

November 2021

Contents

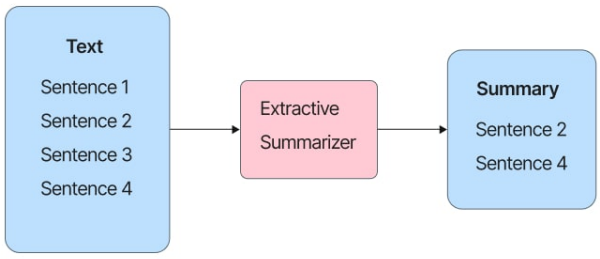
1 Introduction

Document or text summarization is one of the most common tasks in Natural Language Processing. With the amount of new content generated by billions of people and the amount of pre-existing data available, we are inundated with an increasing amount of data every day. Humans can only consume a finite amount of information. Natural Language Processing makes it easier to filter out the wheat from the chaff and find the information that matters. In this project we discuss and implement two main types of summarization i.e. extractive and abstractive summarization. We also created a simple web-application which implements 3 different methods of extractive text summarization.

Repository for the summarizers used in this project: Github repository

2 Extractive Summarizers

We identify the important sentences or phrases from the original text and extract only those from the text. Those extracted sentences would be our summary. It takes the text, ranks all the sentences according to the understanding and relevance of the text, and presents you with the most important sentences. This method does not create new words or phrases, it just takes the already existing words and phrases and presents only that. Here, we discuss and implement four different extractive summarization methods namely TextRank, Luhn, TF-IDF & KL-sum.



2.1 TextRank Summarizer

Text Rank is a kind of graph-based ranking algorithm used for recommendation purposes. TextRank is used in various applications where text sentences are involved. It worked on the ranking of text sentences and recursively computed based on information available in the entire text. It is based on the concept that words which occur more frequently are significant. Hence, the sentences containing highly frequent words are important.

The following steps are performed in a TextRank summarizer:

1. The document is preprocessed and sentence-tokenized.
2. All sentences are further word-tokenized and stopwords are removed.
3. The frequency of every word is calculated, and normalized.
4. Every sentence is assigned a score by adding the normalised frequency values of the words which compose it.
5. High scoring sentences constitute the summary.

```
t = text.lower()
t = t.replace(',','')
t = t.replace('"','')
filtered_sentences = sent_tokenize(t)
for i in range(len(filtered_sentences)):
  filtered_sentences[i] = filtered_sentences[i].replace('.','')

t = t.replace('.','')
filtered_words = word_tokenize(t)
word_freq = {}
for word in filtered_words:
  if word not in stopwords.words('english'):
    if word not in word_freq:
      word_freq[word] = 1
    else:
      word_freq[word] += 1

max_freq = max(word_freq.values())

for word in word_freq:
  word_freq[word] /= max_freq
```

```
sent_scores = []
for sent in filtered_sentences:
  sent_scores.append(0)
  for word in word_tokenize(sent):
    if word not in stopwords.words('english'):
      sent_scores[-1] += word_freq[word]

sent_scores = np.asarray(sent_scores)

summary_ratio = 0.2
summary_len = int(summary_ratio*len(sentences))
summary = ''
summary_idx = np.argsort(sent_scores)[-summary_len:]
summary_idx = summary_idx[::-1]
for i in summary_idx:
  print(sentences[i])
```

Google Colaboratory Notebook for TextRank summarizer.

2.2 Luhn Summarizer

Luhn's method is a simple technique in order to generate a summary from given words. The algorithm can be implemented in two stages.

In the first stage, we try to determine which words are more significant towards the meaning of document. Luhn states that this is first done by doing a frequency analysis, then finding words which are significant, but not unimportant English words.

In the second phase, we find out the most common words in the document, and then take a subset of those that are not these most common english words, but are still important. It usually consists of following three steps:

1. It begins with transforming the content of sentences into a mathematical expression, or vector. Here we use a bag of words , which ignores all the filler words. Filler words are usually the supporting words that do not have any impact on our document meaning. Then we count all the valuable words left to us.

2. In this step, we use evaluate sentences using the sentence scoring technique.

$$Score = \frac{No.\ of\ meaningful\ words^2}{Span\ of\ meaningful\ words}$$

3. Once the sentence scoring is complete, the last step is simply to select those sentences with the highest overall scores.

```
luhn_score = []
for sent in filtered_sentences:
  words = word_tokenize(sent)
  begin = -1
  end = -1
  meaningful_word_count = 0
  for i in range(len(words)):
    word = words[i]
    if word not in stopwords.words('english'):
      meaningful_word_count += 1
      if begin != -1:
        begin = i
      end = i
  span = end-begin+1
  luhn_score.append((meaningful_word_count**2)/span)
luhn_score = np.asarray(luhn_score)
```

```
summary_ratio = 0.2
summary_len = int(summary_ratio*len(sentences))
summary = ''
summary_idx = np.argsort(luhn_score)[-summary_len:]
summary_idx = summary_idx[::-1]
for i in summary_idx:
  summary += sentences[i]+'\n'
print(summary)
```

Google Colaboratory Notebook for Luhn summarizer.

2.3 TF-IDF based Summarizer

TFIDF, short for term frequency–inverse document frequency, is a numeric measure that is use to score the importance of a word in a document based on how often did it appear in that document and a given collection of documents.

The intuition behind this measure is : If a word appears frequently in a document, then it should be important and we should give that word a high score. But if a word appears in too many other documents, it's probably not a unique identifier, therefore we should assign a lower score to that word.

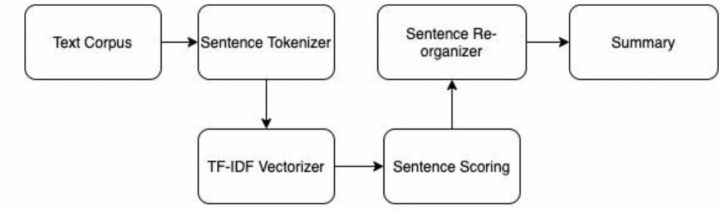
Formula for calculating tf and idf:

$$TF(w) = \frac{No.\ of\ times\ w\ appears\ in\ a\ document}{Total\ no.\ of\ terms\ in\ the\ document}$$

$$IDF(w) = log_e(\frac{Total\ no.\ of\ documents}{No.\ of\ documents\ with\ term\ w\ in\ it})$$

Hence TF-IDF score for a word can be calculated as:

$$w_{i,j} = tf_{i,j} \times log_e(\frac{N}{df_i})$$



```
def summarizer(text, tokenizer, max_sent_in_summary=3):
    # Create spacy document for further sentence level tokenization
    doc = nlp(text_corpus.replace("\n", ""))
    sentences = [sent.string.strip() for sent in doc.sents]
    # Let's create an organizer which will store the sentence ordering to later reorganize the
    # scored sentences in their correct order
    sentence_organizer = {k:v for v,k in enumerate(sentences)}
    # Let's now create a tf-idf (Term frequnecy Inverse Document Frequency) model
    tf_idf_vectorizer = TfidfVectorizer(min_df=2,  max_features=None,
                                        strip_accents='unicode',
                                        analyzer='word',
                                        token_pattern=r'\w{1,}',
                                        ngram_range=(1, 3),
                                        use_idf=1,smooth_idf=1,
                                        sublinear_tf=1,
                                        stop_words = 'english')
    # Passing our sentences treating each as one document to TF-IDF vectorizer
    tf_idf_vectorizer.fit(sentences)
    # Transforming our sentences to TF-IDF vectors
    sentence_vectors = tf_idf_vectorizer.transform(sentences)
    # Getting sentence scores for each sentences
    sentence_scores = np.array(sentence_vectors.sum(axis=1)).ravel()
    # Getting top-n sentences
    N = max_sent_in_summary
    top_n_sentences = [sentences[ind] for ind in np.argsort(sentence_scores, axis=0)[::-1][:N]]
    # Let's now do the sentence ordering using our prebaked sentence_organizer
    # Let's map the scored sentences with their indexes
    mapped_top_n_sentences = [(sentence,sentence_organizer[sentence]) for sentence in top_n_sentences]
    # Ordering our top-n sentences in their original ordering
    mapped_top_n_sentences = sorted(mapped_top_n_sentences, key = lambda x: x[1])
    ordered_scored_sentences = [element[0] for element in mapped_top_n_sentences]
    # Our final summary
    summary = " ".join(ordered_scored_sentences)
    return summary
```

Google Colaboratory Notebook for TF-IDF based summarizer.

2.4 KL-Sum Summarizer

It is a sentence selection algorithm where a target length for the summary is fixed (L words). It is a method that greedily adds sentences to a summary so long as it decreases the KL Divergence. It selects sentences based on similarity of word distribution as the original text. It aims to lower the KL-divergence criteria. It uses a greedy optimization approach and keeps adding sentences till the KL-divergence decreases.

K-L sum algorithm introduces criteria for selecting a summary S from given document D.

$$S^* = min(KL(P_D||P_S))$$

$$S : [words(S) \leq L$$

Where P_s is the empirical unigram distribution of the candidate summary-S and $KL(P||Q)$ represents Kullback-Leiber(KL) divergence given by:

$$\Sigma P(w) log_e(\frac{P(w)}{Q(w)})$$

This quantity represents the divergence between true distribution P and the approximating distribution Q (the summary S distribution).

KL-Divergence calculates the difference between two probability distributions. Probability distribution P to an arbitrary probability distribution Q. It measures between the unigram probability distributions learned from seen document set p(w/R) and new document set q(w/N).

$$D_{KL}(P||Q) = \sum_{i \in W} P(i) log_e \frac{P(i)}{Q(i)}$$

This summarization method finds a set of summary sentences that closely match the document set unigram distribution.

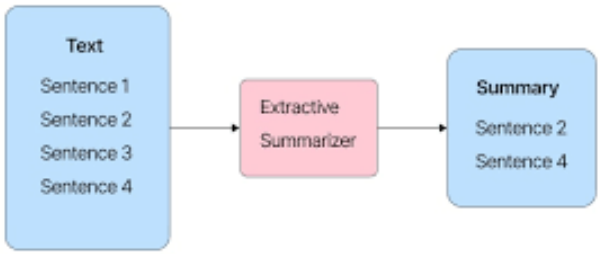
```
from sumy.parsers.plaintext import PlaintextParser
from sumy.nlp.tokenizers import Tokenizer
from sumy.summarizers.kl import KLSummarizer

parser=PlaintextParser.from_string(text,Tokenizer("english"))
# Using KL
summarizer = KLSummarizer()
#Summarize the document with 4 sentences
summary = summarizer(parser.document,4)

for sentence in summary:
    print(sentence)
```

```
They generally ask for the junk food daily because they have been trend so by their parents from the childhood.
It is found according to the Centres for Disease Control and Prevention that Kids and children eating junk food are more prone to the type-2 diabetes.
High sodium and bad cholesterol diet increases blood pressure and overloads the heart functioning.
Junk food is the easiest way to gain unhealthy weight.
```

Google Colaboratory Notebook for KL-Sum summarizer.

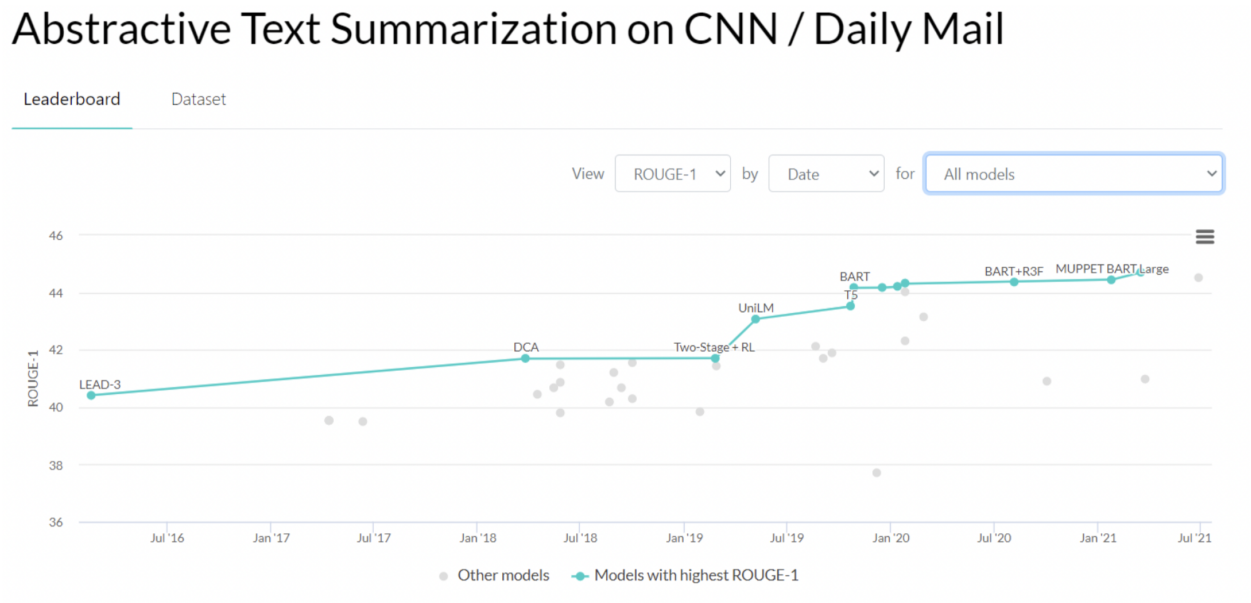
3 Abstractive Text Summarizer

Abstractive summarization, on the other hand, tries to guess the meaning of the whole text and presents the meaning to you. It creates words and phrases, puts them together in a meaningful way, and along with that, adds the most important facts found in the text. This way, abstractive summarization techniques are more complex than extractive summarization techniques and are also computationally more expensive.



However, auto-summarization used to be an impossible task. Specifically, abstractive summarization is very challenging. Differing from extractive summarization (which extracts important sentences from a document and combines them to form a "summary"), abstractive summarization involves paraphrasing words and hence, is more difficult but can potentially give a more coherent and polished summary.

It was not until the development of techniques like **seq2seq learning** and unsupervised language models (e.g., **ELMo** and **BERT**) that abstractive summarization becomes more feasible.



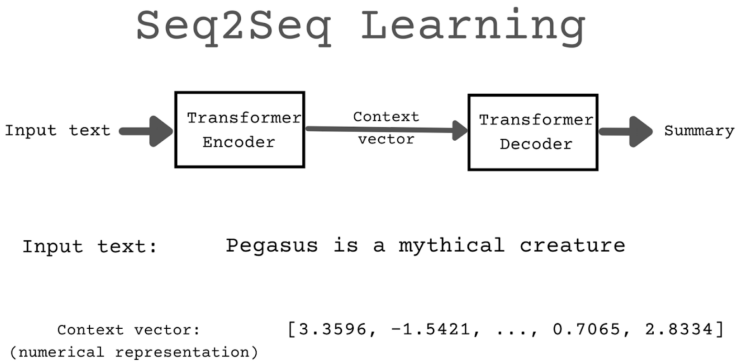
Building upon earlier breakthroughs in natural language processing (NLP) field, Google's PEGASUS further improved the **state-of-the-art (SOTA)** results for abstractive summarization, in particular with low resources. To be more specific, unlike previous models,**PEGASUS** enables us to achieve close to SOTA results with 1,000 examples, rather than tens of thousands of training data.

1. Pegasus' pretraining task is intentionally similar to summarization: important sentences are removed/masked from an input document and are generated together as one output sequence from the remaining sentences, similar to an extractive summary.
2. Pegasus achieves SOTA summarization performance on all 12 downstream tasks, as measured by ROUGE and human eval.

3.1 How does PEGASUS work?

3.1.1 Architecture

On a high level, PEGASUS uses an encoder-decoder model for sequence-to-sequence learning. In such a model, the encoder will first take into consideration the context of the whole input text and encode the input text into something called context vector, which is basically a numerical representation of the input text. This numerical representation will then be fed to the decoder whose job is decode the context vector to produce the summary.



3.1.2 Pre-training of model

GSG(Gap Sentences generation) training was done on several public datasets such as XSum, CNN/Daily mail, Newsroom, Wikihow, Reddit TIFU etc. In contrast to BART or T5, Pegasus masks whole sentences instead of small spans of text. With the pre-trained model, we can then perform fine-tuning of the model on the actual data which is of a much smaller quantity. In fact, evaluation results on various datasets showed that with just 1,000 training data, the model achieved comparable results to previous SOTA models. This has important practical implications as most of us will not have the resources to collect tens of thousands of document-summary pairs.

3.1.3 Implementation

Here, we show particularly the implementation of the **xsum** model of PEGASUS.

```
model_name = 'google/pegasus-xsum'
torch_device = 'cuda' if torch.cuda.is_available() else 'cpu'
tokenizer = PegasusTokenizer.from_pretrained(model_name)
model = PegasusForConditionalGeneration.from_pretrained(model_name).to(torch_device)
```

```
src_text = text

batch = tokenizer.prepare_seq2seq_batch(src_text, truncation=True, padding='longest', return_tensors="pt").to(torch_device)
translated = model.generate(**batch)
tgt_text = tokenizer.batch_decode(translated, skip_special_tokens=True)
print(tgt_text)
```

```
['Junk food is one of the most popular foods in the market today.']
```

Google Colaboratory Notebook for Google PEGASUS Summarizer.

4 Building the Website

In our attempt to build something useful, we have integrated four extractive summarization techniques mentioned above through a flask web app and deployed it through heroku. The web-app requires you too enter the text to be summarized, choose a particular method and also mention the number of sentences required in the summary. On submitting the form, the summary is generated instantly.

The webapp is hosted at `https://summarizeyourtext.herokuapp.com`

Github repository for the web-app

5 Conclusions

Natural language processing has various applications and automatic text summarization is one of the popular and great techniques. There are generally two approaches to achieve text summarization such as extractive summarization and abstractive summarization. Research in the field of automatic text summarization is an ancient challenge and the focus is shifting from extractive summarization to abstractive summarization. The abstractive summary technique generates relevant, exact, content full, and less repetitive summaries. Since the abstractive summarization technique focuses on generating a summary that is nearer to human intelligence it is a challenging field. Therefore, this study exercises the techniques for abstractive summarization along with the pros and cons of various approaches. All the techniques are discussed and compared.

6 Roles & Responsibilities

1. **KL-sum Summarizer** - Madiha & Zaid
2. **TextRank Summarizer** - Pradipto
3. **Luhn Summarizer** - Pratyush & Madiha
4. **TF-IDF based Summarizer** - Pratyush
5. **Google PEGASUS summarizer** - Zaid & Pradipto
6. **Building the website** - Pradipto & Pratyush
7. **Compilation of Project Report** - Madiha & Zaid

7 References

1. Article: Text summarization approaches in NLP
2. International Journal of Computer Applications (0975 – 8887) Volume 102– No.12, September 2014
3. PEGASUS usage example
4. PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization, Zhang et. al.
5. Abstractive Text Summarization on CNN / Daily Mail
6. Article: NLP Basics: Abstractive and Extractive Text Summarization