INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

BANGALORE

NATURAL LANGUAGE PROCESSING

AI 829

Story Summarization

Under the Guidance of:

Prof. Srinath Srinivasa

Submitted by:

Kartik Satish Gawande

MT2023045

kartik.gawande@iiitb.ac.in

Contents

1 Dataset

As a starting point I chose 'Xsum' dataset to get the stories and there summaries. This dataset contain around 227k News article stories. Out of these I took 100k Samples as training set. Throughout this report I will be documenting what Lexical processing I did on these 100k samples. Following is a brief about what Xsum dataset is:

1.1 Xsum Dataset

1. **What is Xsum dataset?**

   XSum was created to facilitate research in abstractive text summarization, where the goal is to produce a concise, informative summary that captures the main points of an article using new sentences. This requires models to have a deeper understanding of the text and the ability to generate coherent and relevant summaries.

2. **Content of this dataset?**

   The dataset consists of BBC articles covering a wide range of topics, including news stories from politics, sports, and entertainment. Each article in the dataset is paired with a professionally written, single-sentence summary that serves as the gold standard for summarization. These summaries are intended to capture the most salient point or the main event described in the article.

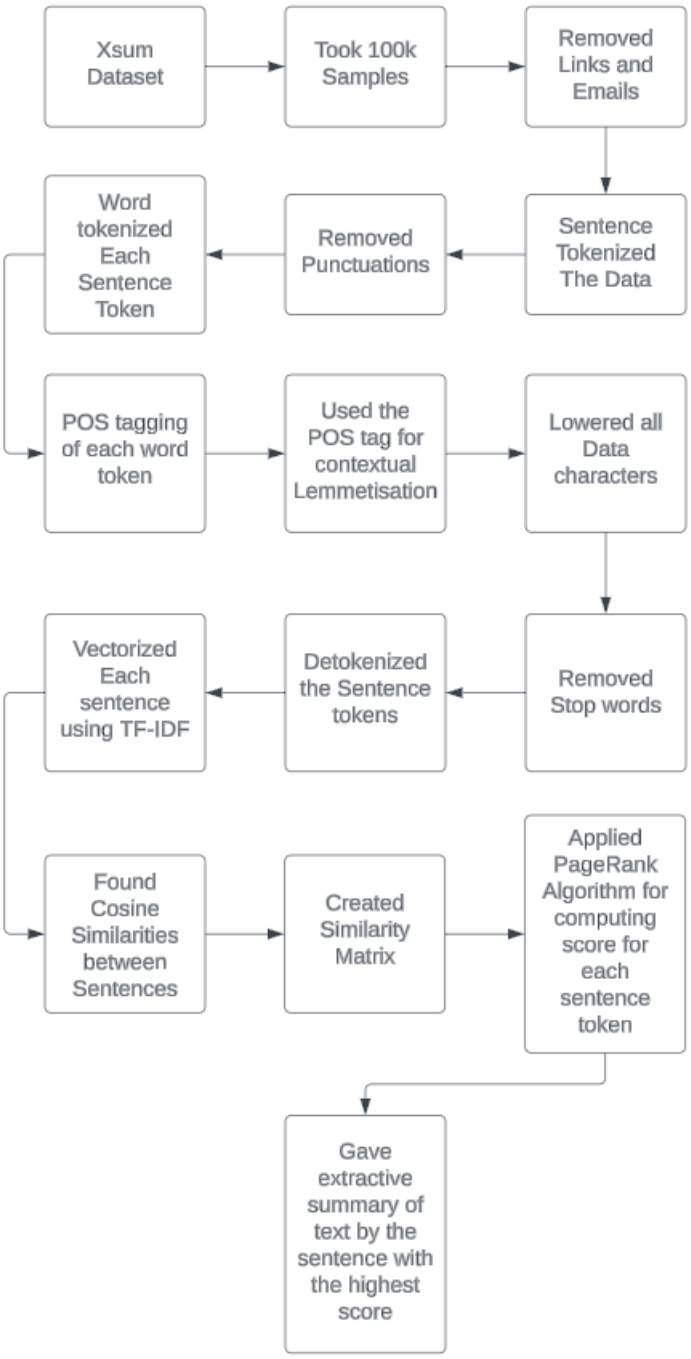
3. **Size of this Dataset?**

   XSum contains 227 thousand document-summary pairs, making it a comprehensive resource for training and evaluating text summarization models.

4. **Why I Chose Xsum Dataset?**

   The summaries in XSum are notably more abstractive than those in other datasets like CNN/Daily Mail. This means that the summaries often include information synthesized from across the article and are not merely extracted sentences. This challenges models to understand and condense articles in a more human-like manner, often requiring inference and a grasp of overall narrative structures.

2 Process flow



Each of these steps and their details are explained in the sections that follows.

3 Lexical Analysis

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A token is a string with an assigned and thus identified meaning. That is, Lexical analysis simplifies the input by converting it into a stream of tokens, which are easier for subsequent processes to handle. It acts as a preprocessing step that abstracts away the character-level details of the input.

Following are the steps I followed in reprocessing the Xsum Dataset:

3.1 Removed Emails and Links:

Emails and links don't contribute substantially in the meaning of the text. Hence in summarization it becomes unwanted data that we don't want our models to put any attention to. So I removed them in the preprocessing itself.

```python
import re

def remove_links_and_emails(example):
    # Define regex patterns for links and emails
    link_pattern = r'https?://\S+|www\.\S+'
    email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

    # Compile regex for efficiency if applying this to many documents
    link_regex = re.compile(link_pattern)
    email_regex = re.compile(email_pattern)

    # Remove links and emails from 'document'
    example['document'] = link_regex.sub('', example['document'])
    example['document'] = email_regex.sub('', example['document'])

    # Remove links and emails from 'summary'
    example['summary'] = link_regex.sub('', example['summary'])
    example['summary'] = email_regex.sub('', example['summary'])

    return example
```

Figure 1: Function for removing Emails and Links from data

3.2 'Sentence Tokenized' the Text:

I wanted to do extractive summarization as part of the first mandate other than other lexical processing. For this we need to sentence tokenize the text.

```
import nltk
from nltk.tokenize import sent_tokenize
nltk.download('punkt')

def sent_tokenize_example(example):
    example['document'] = sent_tokenize(example['document'])
    example['summary'] = sent_tokenize(example['summary'])
    return example
```

Figure 2: Function to sentence tokenize the data

3.3 Removed Punctuations:

Removing punctuation can simplify the dataset by reducing the number of unique tokens the model needs to handle. This can be particularly important for models with a limited vocabulary size. Punctuation can introduce variability that doesn't add meaning. For example, "stop!" and "stop" convey the same command, but they would be treated as two different tokens if punctuation were not removed. So I removed Punctuations.

```
import string
string.punctuation
def remove_punctuations(example):
    table = str.maketrans('', '', string.punctuation)
    example['document'] = [word.translate(table) for word in example['document'] if word not in string.punctuation]
    example['summary'] = [word.translate(table) for word in example['summary'] if word not in string.punctuation]

    # Remove empty strings that may result from removing punctuation
    example['document'] = [token for token in example['document'] if token]
    example['summary'] = [token for token in example['summary'] if token]
    return example
```

Figure 3: Function to remove Punctuations from Data

3.4 'Word Tokenized' each Sentence token:

For Lemmetisation, I used POS tagging. POS tagging needs tokenization at the level of words. So I word tokenized the sentence tokens further.

```
#word tokenization
from nltk.tokenize import word_tokenize

def word_tokenize_sent_tokens(example):
    example['document']=[word_tokenize(sentence) for sentence in example['document']]
    example['summary']=[word_tokenize(sentence) for sentence in example['summary']]
    return example
```

Figure 4: Function to word tokenize each sentence tokens

3.5 POS tagging and Lemmetisation:

· Part-of-Speech (POS) tagging:

Part-of-Speech (POS) tagging is the process of assigning a part of speech to each word in a sentence, such as noun, verb, adjective, adverb, etc. POS tagging is essential because many words can represent more than one part of speech at different times and POS determines the role that a word plays in the sentence.

For example, in the sentence "The bear will bear the weight", the first "bear" is a noun, while the second "bear" is a verb. A POS tagger would annotate this as

"The/DT bear/NN will/MD bear/VB the/DT weight/NN"

where DT denotes a determiner, NN a noun, and VB a verb.

Since the WordNet lemmatizer in NLTK expects POS tags that are compatible with the WordNet tag set. These are '**n**' for noun, '**v**' for verb, '**a**' for adjective, and '**r**' for adverb. However, the NLTK POS tagger outputs tags like '**NN**' for noun, '**VB**' for verb, '**JJ**' for adjective, and '**RB**' for adverb. Hence I used the following function to convert NLTK's POS Tags to the tags expected by WordNet lemmatizer using the following function:

```
# Function to convert NLTK's POS tags to the format expected by the WordNet lemmatizer
def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN
```

Figure 5: Function to convert NLTK's POS tag to WordNet Tag set

- **Lemmatization:**

Lemmatization takes into consideration the morphological analysis of the words. To do so, it requires detailed dictionaries which the algorithm can look through to link the form back to its lemma. Moreover, lemmatization depends on correctly identifying the intended POS of a word, as many words will have different lemmas based on their POS (e.g., the word "saw" as a noun has the lemma "saw", but as a verb, it has the lemma "see").
So, for lemmatization I used the following function:

```
def pos_tag_and_lemmatize_example(example):
    temp = {'document':[],'summary':[],'id':''}
    temp['id']=example['id']
    for sentence in example['document']:
      tagged_tokens = pos_tag(sentence)
      temp['document'].append([lemmatizer.lemmatize(wordl, tagl) for wordl, tagl in [(word, get_wordnet_pos(tag)) for word, tag in tagged_tokens]])

    for sentence in example['summary']:
      tagged_tokens = pos_tag(sentence)
      temp['summary'].append([lemmatizer.lemmatize(wordl, tagl) for wordl, tagl in [(word, get_wordnet_pos(tag)) for word, tag in tagged_tokens]])
    return temp
```

Figure 6: Function to lemmatize the Text using the POS tags

For this the following imports were used:

```
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
from nltk import pos_tag
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
lemmatizer = WordNetLemmatizer()
```

Figure 7: Imports for POS tagging and Lemmatization

3.6 Lowered all characters:

To reduce the overall size of the vocabulary or the number of unique tokens that the model in the upcoming mandates will need to understand I lowered all the characters in the text. For example, "House" and "house" would be considered two different tokens if case distinctions were preserved.

```
def lower(example):
    temp = {'document':[],'summary':[],'id':''}
    temp['id']=example['id']
    for sentence in example['document']:
      temp['document'].append([word.lower() for word in sentence])

    for sentence in example['summary']:
      temp['summary'].append([word.lower() for word in sentence])
    return temp
```

Figure 8: Function to lower text characters

3.7 Removed Stop words:

Stop words are typically common words in a language that carry very little meaningful information about the content of the text. So, it is best to remove them. They include words such as "the", "is", "in", "on", "and", and "but".

```
from nltk.corpus import stopwords
nltk.download('stopwords')  # Download the list of stopwords
stop_words = set(stopwords.words('english'))

def remove_stopwords(example):
    temp = {'document':[],'summary':[],'id':''}
    temp['id']=example['id']
    for sentence in example['document']:
      temp['document'].append([word for word in sentence if word not in stop_words])

    for sentence in example['summary']:
      temp['summary'].append([word for word in sentence if word not in stop_words])
    return temp
```

Figure 9: Function to remove Stop words from the text

4 PageRank Algorithm

Other than the lexical analysis, since the extractive summarization using Graph algorithms like PageRank algorithm requires very little outside the lexical analysis except for a bunch of concepts like TF-IDF, Similarity formulae, PageRank Algorithm etc I included this as part of mandate 2 itself.

```python
import numpy as np
import networkx as nx
from sklearn.metrics.pairwise import cosine_similarity
from nltk.tokenize.treebank import TreebankWordDetokenizer
from sklearn.feature_extraction.text import TfidfVectorizer

def summarize_by_page_rank(sentences):
  # Preprocess: Convert tokenized sentences to strings
  detokenizer = TreebankWordDetokenizer()
  sentences = [detokenizer.detokenize(sentence) for sentence in sentences]

  # Vectorize the sentences using TF-IDF
  vectorizer = TfidfVectorizer()
  sentence_vectors = vectorizer.fit_transform(sentences).toarray()

  # Compute similarity matrix
  similarity_matrix = cosine_similarity(sentence_vectors)

  # Build the graph
  nx_graph = nx.from_numpy_array(similarity_matrix)
  # Apply PageRank
  scores = nx.pagerank(nx_graph)

  # Rank sentences based on scores
  ranked_sentences = sorted(((scores[i],s) for i,s in enumerate(sentences)), reverse=True)

  # Extract the top N sentences as the summary
  N = 1  # Number of sentences in the summary
  summary = ' '.join([ranked_sentences[i][1] for i in range(N)])
  return summary
```

Figure 10: PageRank Function to extract summary sentence from text

4.1 **Detokenization**: The function first takes a list of tokenized sentences as input. Each tokenized sentence (a list of words) is converted back into a regular sentence string using the **TreebankWordDetokenizer**. This step prepares the data for TF-IDF vectorization.

```
# Preprocess: Convert tokenized sentences to strings
detokenizer = TreebankWordDetokenizer()
sentences = [detokenizer.detokenize(sentence) for sentence in sentences]
```

Figure 10: Detokenize word tokens in the sentence tokens

4.2 **TF-IDF Vectorization**: The sentences are then vectorized using Term Frequency-Inverse Document Frequency (TF-IDF). This method transforms the text into a numerical form that reflects not just the frequency of each word in the sentence (Term Frequency), but also how unique a word is to the sentences in the corpus (Inverse Document Frequency). The TF-IDF vectorizer converts the sentences to a matrix of TF-IDF features.

- **Term Frequency (TF)**: The number of times a term occurs in a document. The assumption is that the importance of a term increases proportionally to its frequency.
- **Inverse Document Frequency (IDF)**: This down-scales terms that appear a lot across documents. It is computed as the logarithm of the number of documents divided by the number of documents that contain the term.

Mathematically, for a term **t** in document **d**, TF-IDF is calculated as:

$$TFIDF(t,d) = TF(t,d) \times IDF(t)$$
$$IDF(t) = \log \frac{1+n}{1+df(t)} + 1$$

Equation 1: TF-IDF formula

where:

- **n** is the total number of documents
- **df(t)** is the number of documents that contain term **t**.

```
# Vectorize the sentences using TF-IDF
vectorizer = TfidfVectorizer()
sentence_vectors = vectorizer.fit_transform(sentences).toarray()
```

Figure 11: Code to vectorize text with TF-IDF

4.3 **Cosine Similarity**: The cosine similarity is a measure used to calculate how similar two vectors are. It is defined as the cosine of the angle between two vectors, which is calculated by taking the dot product of the vectors and dividing it by the product of their magnitudes (lengths).

Mathematically, for two vectors **A** and **B**, the cosine similarity **cos(θ)** is:

$$cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

Equation 2: Cosine Similarity function

I used the **cosine similarity** function from **sklearn.metrics.pairwise** to calculate the similarity matrix between all pairs of sentence vectors. The result is a square matrix where each element **(i, j)** represents the cosine similarity between sentence **i** and sentence **j**.

```
# Compute similarity matrix
similarity_matrix = cosine_similarity(sentence_vectors)
```

Figure 12: Code to find cosine similarity between sentences

4.4 **Graph Creation**: I then used this similarity matrix to create a graph with sentences as nodes and the similarities as edge weights. NetworkX library was used to create a graph from the similarity matrix using numpy array, which interprets the matrix as an adjacency matrix.

```
# Build the graph
import networkx as nx
nx_graph = nx.from_numpy_array(similarity_matrix)
# Apply PageRank
scores = nx.pagerank(nx_graph)
```

Figure 13: Code to create graph from Similarity Matrix

4.5 **PageRank Score**: The PageRank algorithm is then applied to the graph to rank the sentences. PageRank assigns a score to each sentence based on the scores of the sentences linking to it and the number of links they each have. The **pagerank** function from NetworkX library computes these scores.

The PageRank formula is:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

Equation 3: PageRank Algorithm Formula

where:

- $PR(p_i)$ is the PageRank of page p_i.
- d is the damping factor (usually set to 0.85).
- N is the total number of pages(sentences in our case).
- $M(p_i)$ is the set of pages that link to p_i.
- $L(p_j)$ is the number of outbound links on page p_j.

```
# Apply PageRank
scores = nx.pagerank(nx_graph)
```

Figure 14: apply PageRank to find scores of each sentence

4.6 **Sentence Ranking and Summary Extraction**: Then I sorted sentences based on their PageRank scores in descending order. The top **N** sentences (in this case, **N** is set to 1) are chosen to form the summary.

```
# Rank sentences based on scores
ranked_sentences = sorted(((scores[i],s) for i,s in enumerate(sentences)), reverse=True)

# Extract the top N sentences as the summary
N = 1  # Number of sentences in the summary
summary = ' '.join([ranked_sentences[i][1] for i in range(N)])
return summary
```

Figure 15: Sorting sentences as per their PageRank score and choosing sentence with Highest Score for the extractive summary

```
text = train_sent_word_tokens['document'][0] #sentence tokenized text
summarize_by_page_rank(text) #returns summary as per page rank algorithm using cosine similarity of tf idf
```

```
'affect flood dumfries galloway borders say important immediate step take protect area vulnerable clear timetable put place flood prevention plan'
```

Figure 4: Example output using PageRank Algorithm

5 Collab Notebook

Link

6 Future Mandates' Plan

I am planning to try various LLM models and fine tune them for creating abstractive summarization.