Building an Advanced NLP Text Summarizer with Streamlit

Header image showing text summarization concept

Introduction

In today's information-saturated world, the ability to quickly extract key insights from large volumes of text is increasingly valuable. Text summarization is a fascinating application of Natural Language Processing (NLP) that addresses this need by condensing documents while preserving their essential meaning.

In this tutorial, we'll build a powerful text summarization application using Python and Streamlit. Our summarizer will support both extractive and abstractive summarization techniques, and handle multiple input types including raw text, PDFs, and web articles.

By the end of this tutorial, you'll have:

- A solid understanding of extractive vs. abstractive summarization
- Experience integrating transformer-based models (BART) with traditional NLP techniques
- A fully functional web application for text summarization

Let's dive in!

Prerequisites

Before starting, ensure you have the following installed:

- Python 3.7+
- pip (Python package manager)

You'll also need to install the following packages:

```
pip install streamlit spacy transformers torch pdfplumber newspaper3k
python -m spacy download en_core_web_sm
```

Understanding Text Summarization Approaches

Our application will implement two main summarization techniques:

1. **Extractive Summarization**: Identifies and extracts key sentences from the original text. This approach preserves the original wording but rearranges content for brevity.

2. **Abstractive Summarization**: Generates new sentences that capture the essence of the original text. This approach can produce more concise and natural-sounding summaries but requires more computational power.

Project Structure

We'll build our application as a single Python file that integrates all the necessary components:

- A `TextSummarizer` class that handles the NLP processing
- A Streamlit UI for user interaction
- Support for multiple input types (text, PDF, URL)

Creating the TextSummarizer Class

Let's start by creating our main class that will handle the summarization logic:

```
import spacy
import heapq
import torch
from transformers import BartTokenizer, BartForConditionalGeneration
import pdfplumber
from newspaper import Article

# Load spaCy model ONCE
nlp = spacy.load("en_core_web_sm")

class TextSummarizer:
    def __init__(self, model_name="facebook/bart-large-cnn"):
        """Initialize Summarizer with BART transformer model."""
        self.model_name = model_name
        self.tokenizer = BartTokenizer.from_pretrained(model_name)
        self.model = BartForConditionalGeneration.from_pretrained(model_name)
```

Here, we initialize our class with the BART model from Facebook/Meta, which is specifically fine-tuned for text summarization tasks. We load the tokenizer and model during initialization to avoid reloading them for each summarization request.

Implementing Extractive Summarization

Next, let's implement our extractive summarization method:

```
def extractive_summary(self, text, num_sentences=5):
    """Extractive summarization with small text handling."""
    doc = nlp(text)
    sentences = [sent.text for sent in doc.sents if sent.text.strip()]

    # Handle short texts
    if len(sentences) <= num_sentences:
        return " ".join(sentences)

    # Frequency-based sentence scoring
    word_frequencies = {}
    for token in doc:
        if not token.is_stop and not token.is_punct:
            lemma = token.lemma_.lower()
            word_frequencies[lemma] = word_frequencies.get(lemma, 0) + 1
    max_freq = max(word_frequencies.values()) if word_frequencies else 1
    for word in word_frequencies:
        word_frequencies[word] /= max_freq

    sentence_scores = {}
    for sent in sentences:
        sent_doc = nlp(sent.lower())
        for token in sent_doc:
            if token.lemma_.lower() in word_frequencies:
                sentence_scores[sent] = sentence_scores.get(sent, 0) +
word_frequencies[token.lemma_.lower()]

    summary_sentences = heapq.nlargest(min(num_sentences, len(sentences)),
sentence_scores, key=sentence_scores.get)
    return " ".join(summary_sentences)
```

Our extractive summarization approach:

1. Breaks text into sentences using spaCy
2. Calculates word frequencies (ignoring stop words and punctuation)
3. Normalizes these frequencies
4. Scores each sentence based on the frequency of its component words
5. Selects the highest-scoring sentences to form our summary

Implementing Abstractive Summarization

Now let's add our abstractive summarization method using the BART transformer model:

```
def abstractive_summary(self, text, max_length=512, min_length=100):
    """Abstractive summarization with small text handling."""
    # Handle short texts by dynamically adjusting length parameters
    text_length = len(text.split())

    if text_length < 50:
        min_length = 30
        max_length = 100
    elif text_length < 100:
        min_length = 50
        max_length = 200
    else:
        min_length = 100
        max_length = 512

    inputs = self.tokenizer.encode(
        "summarize: " + text,
        return_tensors="pt",
        max_length=1024,
        truncation=True
    )
    summary_ids = self.model.generate(
        inputs,
        max_length=max_length,
        min_length=min_length,
        length_penalty=2.0,
        num_beams=4,
        early_stopping=True
    )
    return self.tokenizer.decode(summary_ids[0], skip_special_tokens=True)
```

In this method:

1. We dynamically adjust the summary length based on input text length
2. Encode the input text with the "summarize:" prefix to guide the model
3. Generate the summary using beam search (considering multiple possibilities at each step)
4. Decode the resulting token IDs back to text

The parameters like `length_penalty` and `num_beams` help control the quality and length of the generated summary.

Adding PDF and URL Support

To make our summarizer more versatile, let's add methods to handle PDFs and web articles:

```
def summarize_pdf(self, pdf_path):
    """Extract and summarize PDF text."""
    text = ""
    with pdfplumber.open(pdf_path) as pdf:
        for page in pdf.pages:
            page_text = page.extract_text()
            if page_text:
                text += page_text + "\n"

    if not text.strip():
        return "No text found in PDF."

    max_input_length = 1024
    chunks = [text[i:i+max_input_length] for i in range(0, len(text),
max_input_length)]

    summaries = [self.abstractive_summary(chunk) for chunk in chunks]
    return " ".join(summaries)

def summarize_url(self, url):
    """Extract and summarize URL content."""
    article = Article(url)
    try:
        article.download()
        article.parse()
    except Exception as e:
        return f"Failed to download/parse the URL: {e}"
```

```python
        text = article.text.strip()
        if not text:
            return "No main article content found."

        return self.abstractive_summary(text)
```

The PDF method uses pdfplumber to extract text from PDF files, then breaks it into manageable chunks before summarizing each part.

The URL method leverages the newspaper3k library to extract the main content from web articles, filtering out menus, ads, and other irrelevant elements.

Building the Streamlit UI

Finally, let's create a user-friendly interface with Streamlit:

```python
import streamlit as st

# Streamlit UI
st.title("NLP Text Summarizer")

option = st.selectbox("Choose an option", ["Text", "PDF", "URL"])
summarizer = TextSummarizer()

if option == "Text":
    user_input = st.text_area("Enter text to summarize:")
    if st.button("Summarize Text"):
        if len(user_input.split()) < 10:
            st.warning("Text is too short for meaningful summarization.")
        else:
            extractive = summarizer.extractive_summary(user_input)
            abstractive = summarizer.abstractive_summary(user_input)

            st.subheader("Extractive Summary")
            st.write(extractive)

            st.subheader("Abstractive Summary")
            st.write(abstractive)

elif option == "PDF":
    uploaded_file = st.file_uploader("Upload a PDF", type=["pdf"])
    if uploaded_file and st.button("Summarize PDF"):
        with open("temp.pdf", "wb") as f:
            f.write(uploaded_file.read())
        pdf_summary = summarizer.summarize_pdf("temp.pdf")
        st.subheader("PDF Summary")
        st.write(pdf_summary)

elif option == "URL":
    url_input = st.text_input("Enter a URL to summarize:")
    if st.button("Summarize URL"):
        url_summary = summarizer.summarize_url(url_input)
        st.subheader("URL Summary")
        st.write(url_summary)
```

Our Streamlit interface provides:

1. A dropdown to select between text, PDF, or URL input
2. Appropriate input fields for each selection
3. Summary display areas that show both extractive and abstractive summaries for text input

Running the Application

Save the entire code to a file (e.g., `text_summarizer.py`) and run it with:

```
streamlit run text_summarizer.py
```

This will launch your application in a web browser, typically at http://localhost:8501.

Testing the Summarizer

To test your application:

1. **Text Summarization**: Paste a lengthy article or document into the text area and click "Summarize Text"
2. **PDF Summarization**: Upload a PDF document and click "Summarize PDF"
3. **URL Summarization**: Enter a URL to a news article or blog post and click "Summarize URL"

Compare the extractive and abstractive summaries to see how they differ in approach and output.

Enhancing the Summarizer

Once you have the basic functionality working, here are some ways to enhance your summarizer:

- **Add language detection** to support multiple languages
- **Implement summary length controls** to let users adjust summary length
- **Add keyword extraction** to highlight key terms in the summarized text
- **Implement sentiment analysis** to capture the emotional tone of the text
- **Add visualization** components to show word frequencies or sentence importance

Conclusion

In this tutorial, we've built a versatile text summarization application that combines classical NLP techniques with state-of-the-art transformer models. This hybrid approach allows for both extractive summaries (preserving original wording) and abstractive summaries (generating new content that captures key ideas).

By integrating our summarizer with Streamlit, we've created an accessible user interface that makes NLP technology available to users without programming experience. The ability to process text, PDFs, and web content makes this tool a valuable asset for researchers, students, and professionals dealing with information overload.

As NLP continues to advance, the possibilities for text summarization will only expand. By understanding the foundations built in this tutorial, you'll be well-prepared to incorporate future innovations into your own summarization tools.

About the Author

[Your Name] is a data scientist and NLP enthusiast specializing in text analytics and machine learning applications. Follow me for more tutorials on practical applications of AI and natural language processing.