Project Overview: PubMed Article Summarizer

My project is a web application that summarizes scientific articles from PubMed. It uses natural language processing techniques to create concise summaries of longer texts, making it easier for users to quickly grasp the main points of scientific papers.

1. Data Preprocessing (preprocessing.ipynb):

In this phase, I prepared the PubMed dataset for use in my summarization task.

a) Dataset Loading:

- I used the Hugging Face 'datasets' library to load the PubMed dataset.
- Hugging Face is a popular platform that provides datasets and models for machine learning tasks.
- The PubMed dataset contains scientific articles and their abstracts.

b) Text Preprocessing:

- I implemented several text cleaning steps:
    1. Tokenization: Breaking down the text into individual words or sentences.
    2. Lowercasing: Converting all text to lowercase for consistency.
    3. Removing digits: Eliminating numbers from the text.
    4. Removing special characters: Cleaning the text of non-alphabetic characters.
    5. Removing stopwords: Eliminating common words that don't carry much meaning.
- Stopwords Explanation:
    - Stopwords are common words in a language that don't contribute much to the overall meaning (e.g., "the", "is", "at", "which").
    - Removing stopwords helps focus on the important content words in the text.
    - I used NLTK (Natural Language Toolkit) to get a list of English stopwords.

c) Dataset Saving:

- After preprocessing, I saved the cleaned dataset for later use in my application.

2. Flask Web Application (app.py):

This is the main backend of my web application, handling user requests and text summarization.

a) Setup:

- I used Flask, a lightweight web framework for Python.
- I imported necessary libraries for summarization (sumy) and PDF handling (PyPDF2, reportlab).

b) Text Summarization:

- I implemented the summarize_text function using the LSA (Latent Semantic Analysis) algorithm from the sumy library.
- LSA is a technique that analyzes relationships between a set of documents and the terms they contain.
- I created two types of summaries: brief (3 sentences) and detailed (7 sentences).

c) File Handling:

- My application can accept both direct text input and file uploads (PDF and TXT).
- For PDFs, I used PyPDF2 to extract text from the uploaded file.

d) Routes:

- Main route ('/'):
    - Handles both GET and POST requests.
    - Processes user input (text or file), generates summaries, and calculates word counts.
- Export route ('/export-pdf'):
    - Creates a PDF of the original text and summary using reportlab.

3. Frontend (index.html):

This is the user interface of my web application.

a) Design:

- I created a responsive design using HTML and CSS.
- The interface includes a text area for input, file upload option, and radio buttons to choose summary type.

b) Features:

- Users can input text directly or upload PDF/TXT files.
- They can choose between brief and detailed summaries.
- The page displays the original text, summary, and word counts for both.
- Users can export the results as a PDF.

c) Interactivity:

- I used some JavaScript to enhance the user experience, like showing the name of the uploaded file.

How It All Works Together:

1. When a user accesses my web app, they see the interface created by index.html.
2. The user can enter text or upload a file and choose a summary type.
3. When they submit, the Flask backend (app.py) processes this input:
    - If it's a file, it extracts the text.
    - It then uses the LSA summarizer to create a summary.
    - It calculates word counts for the original and summarized text.
4. The results are sent back to the frontend and displayed to the user.
5. The user can then choose to export these results as a PDF.