Article Summarizer AI

November 6, 2024

1 Research Paper Summarizing Tool

1.0.1 Web Scraping Research Papers with BeatifulSoup

Internet Archive Scholar is a free search tool for scholarly documents and research articles that claims to have access to over 35 million scholarly documents. To train our AI tool, we must provide data to train our model, which we will scrape from Internet Archive Scholar.

Abstract: The goal of this project is to develop a web scraper using the BeautifulSoup library to fetch and parse research papers from Internet Archive Scholar. The scraper will navigate through multiple pages of the search results and gather links to scholarly documents.

1.0.2 Overview

The Internet Archive Scholar does not provide a simple way to directly access all articles on its platform via static URLs. However, by using a wildcard search (*), we can retrieve every document available on the site, as it returns a broad match for all content. The site uses pagination, with each page displaying 15 documents at a time, and the offset parameter increments by 15 for each page turn.

1.0.3 Approach

- URL Structure:

The base URL starts at offset=0 for the first page, and with each subsequent page, the offset increases by 15 (e.g., offset=15 for page 2, offset=30 for page 3, etc.). We use this structure to loop through the pages and collect article links.

- Data Collection:

On each page, the program looks for tags that contain the article links. These tags are identified by specific title values, such as 'read fulltext microfilm' or 'fulltext access'. Each article URL is extracted from the href attribute of these tags and added to a Python set to ensure uniqueness.

- Handling Duplicates:

A set is used to store the article URLs, ensuring that only unique links are captured, even if duplicates appear across multiple pages or within the same page.

- Testing and Timeouts:

To prevent the script from timing out or overwhelming the server during testing, we limit the number of URLs fetched per run. For example, we can limit the scraper to fetching only the first page of results during early stages of testing.

1.0.4 Error Handling and Edge Cases

- The scraper uses a try-except block to handle network-related issues, such as timeouts. If the server does not respond in time or other network errors occur, the scraper will catch the exception and print a message indicating the timeout.
- If a page returns fewer than 15 articles, the script assumes that it has reached the last page and stops fetching further pages. This ensures efficiency and avoids unnecessary requests.

[1]:
```
from bs4 import BeautifulSoup
import requests

def fetch_articles(max_pages):
    base_url = "https://scholar.archive.org/search?q=*&offset={}"
    offset = 0
    articles_per_page = 15
    articles_found = set()

    try:
        while offset < max_pages * articles_per_page:
            internet_archive_scholar = requests.get(base_url.format(offset))
            soup = BeautifulSoup(internet_archive_scholar.text, "html.parser")

            current_page_articles = 0

            for a in soup.find_all("a"):
                if 'title' in a.attrs and (a['title'] == 'read fulltext␣
 ↪microfilm' or a['title'] == 'fulltext access') and 'href' in a.attrs:
                    url = a['href']
                    if url not in articles_found:
                        articles_found.add(url)
                        current_page_articles += 1

            if current_page_articles < articles_per_page:
                print("Less than 15 articles found, likely last page.")
                break

            offset += articles_per_page
        return articles_found

    except requests.exceptions.Timeout:
        print("The request timed out.")

article_urls = fetch_articles(1)
```

```
print(article_urls)
```

```
{'https://archive.org/details/sim_american-journal-of-occupational-
therapy_november-december-1949_3_6/page/288',
'https://archive.org/details/sim_london-quarterly-and-holborn-
review_1868-07_30_60_0/page/366', 'https://archive.org/details/sim_eclectic-
magazine-of-foreign-literature_1867-09_6_3/page/316',
'https://archive.org/details/sim_frasers-magazine_1847-12_36_216/page/704',
'https://archive.org/details/sim_automobile-magazine_1899-11_1_2/page/146',
'https://archive.org/details/sim_gentlemans-magazine_1830-04_100/page/311',
'https://archive.org/details/sim_journal-of-teacher-education_november-
december-1992_43_5/page/333', 'https://archive.org/details/sim_british-critic-
and-quarterly-theological-review_1802-06_19/page/689',
'https://archive.org/details/sim_forum-and-century_1920-07_64_1/page/98',
'https://archive.org/details/sim_growth-development-and-
aging_1937-09_1_3/page/191', 'https://archive.org/details/sim_american-journal-
of-physiology_1998-03_43_3/page/34',
'https://archive.org/details/sim_quiver_may-21-june-18-1864_6_32/page/201',
'https://archive.org/details/sim_dial_1921-01_70/page/21',
'https://archive.org/details/sim_united-states-national-museum-annual-
report_1921-06-30/page/85', 'https://archive.org/download/crossref-
pre-1923-scholarly-works/10.1007%252Fbf01427558.zip/10.1007%252Fbf01428189.pdf'}
```

1.0.5 Grabbing Full Text From Article

After obtaining the URLs of each article, the next step is to extract the full-text content from these articles. For this purpose, we need to navigate through each article page, locate the link that provides the full-text version of the article, if exists, and then extract it. Below is an explanation of the process followed by the code implementation.

1.0.6 Approach

1. Fetch Article HTML:

For each URL in the list of article URLs, we send a request to fetch the HTML content of that page. This is achieved using the requests.get() method, which retrieves the HTML response for the article.

2. Parse the HTML:

Once the HTML content is retrieved, it is passed to BeautifulSoup to parse the HTML structure. This parsing allows us to easily navigate through the different HTML tags and extract the data we need.

3. Locate the Full-Text Link:

On each article page, the link to download the full text is usually embedded in an tag with some associated text, often labeled as “FULL-TEXT download” or similar.

We loop through all tags on the page using soup.find_all(“a”) and examine the text content of each anchor tag. We use the .get_text().strip() method to clean up any extra spaces around the

anchor text and compare it against the expected string “FULLTEXTdownload”, ignoring spaces.

4. Extract and Store the Full Text URL:

Once we identify the correct anchor tag that contains the full-text link, we extract the URL from the href attribute. Since the URLs on the site are often relative (not including the domain), we prepend the base URL (https://archive.org) to the relative link.

Since a FULL-TEXT link exists, the author and title are extracted from the page by using a try-and-catch on the “span” tags for the name and publisher. The title and author are then joined and used as a key in our hashmap and the full-text URLs are then stored as the value.

5. Return the Full-Text Links:

After processing all articles, the hashmap is returned, which can then be used for further processing, such as downloading and scraping the text content of each full-text article.

[11]:
```
def fetch_full_text(url_list):
    title_to_link = {}
    unique_counter = 1

    for url in url_list:
        article = requests.get(url)
        soup = BeautifulSoup(article.text, "html.parser")

        for a in soup.find_all('a'):
            anchor_text = a.get_text().strip()

            if anchor_text.replace(" ", "") == "FULLTEXTdownload":
                if 'href' in a.attrs:
                    url = "https://archive.org" + a['href']
                    try:
                        identifier = soup.find('span', itemprop="identifier").
 ↪get_text()
                        if not identifier:
                            identifier = f"unique-{unique_counter}"
                            unique_counter += 1
                    except AttributeError:
                        identifier = f"unique-{unique_counter}"
                        unique_counter += 1

                    title_to_link[identifier] = url

    return title_to_link

article_map = fetch_full_text(article_urls)
print(article_map)
```

```
{'sim_american-journal-of-occupational-therapy_november-december-1949_3_6':
'https://archive.org/stream/sim_american-journal-of-occupational-
```

therapy_november-december-1949_3_6/sim_american-journal-of-occupational-therapy_november-december-1949_3_6_djvu.txt', 'sim_london-quarterly-and-holborn-review_1868-07_30_60_0': 'https://archive.org/stream/sim_london-quarterly-and-holborn-review_1868-07_30_60_0/sim_london-quarterly-and-holborn-review_1868-07_30_60_0_djvu.txt', 'sim_eclectic-magazine-of-foreign-literature_1867-09_6_3': 'https://archive.org/stream/sim_eclectic-magazine-of-foreign-literature_1867-09_6_3/sim_eclectic-magazine-of-foreign-literature_1867-09_6_3_djvu.txt', 'sim_frasers-magazine_1847-12_36_216': 'https://archive.org/stream/sim_frasers-magazine_1847-12_36_216/sim_frasers-magazine_1847-12_36_216_djvu.txt', 'sim_automobile-magazine_1899-11_1_2': 'https://archive.org/stream/sim_automobile-magazine_1899-11_1_2/sim_automobile-magazine_1899-11_1_2_djvu.txt', 'sim_gentlemans-magazine_1830-04_100': 'https://archive.org/stream/sim_gentlemans-magazine_1830-04_100/sim_gentlemans-magazine_1830-04_100_djvu.txt', 'sim_british-critic-and-quarterly-theological-review_1802-06_19': 'https://archive.org/stream/sim_british-critic-and-quarterly-theological-review_1802-06_19/sim_british-critic-and-quarterly-theological-review_1802-06_19_djvu.txt', 'sim_forum-and-century_1920-07_64_1': 'https://archive.org/stream/sim_forum-and-century_1920-07_64_1/sim_forum-and-century_1920-07_64_1_djvu.txt', 'sim_growth-development-and-aging_1937-09_1_3': 'https://archive.org/stream/sim_growth-development-and-aging_1937-09_1_3/sim_growth-development-and-aging_1937-09_1_3_djvu.txt', 'sim_quiver_may-21-june-18-1864_6_32': 'https://archive.org/stream/sim_quiver_may-21-june-18-1864_6_32/sim_quiver_may-21-june-18-1864_6_32_djvu.txt', 'sim_dial_1921-01_70': 'https://archive.org/stream/sim_dial_1921-01_70/sim_dial_1921-01_70_djvu.txt'}

Scraping Text Content From Articles After grabbing the full-text links, we will now work on replacing the value (full-text URL link) with the text content from the article. Iterating over our newly populated hashmap, we search for valid "http" links extract the cleaned text content, and update our hashmap.

1.0.7 Approach

1. Fetch Full-Text URL

For each key in our hashmap, we store and identify the value, and if it contains a URL, we perform a request to fetch the HTML content of the FULL-TEXT URL.

2. Parse HTML Content and Locate Text-Content

When the HTML content is retrieved, it is passed through Beautiful Soup to parse the link. On each Full-Text page, the text content is stored inside of a 'pre' tag.

3. Extract and Store Text Content If the text_content exists, it is cleaned from excessive newlines and is used to update the hashmap value. Otherwise, the hashmap value would be updated to 'FULL TEXT NOT AVAILABLE'.

To test out our method, we update our entire hashmap and only print one hashmap value, which shows the title, author and text content. In this example, we are only showing 400 characters because the text files can be large.

[12]:
```
def update_value(article_map):
    for key in article_map:
        url = article_map[key]

        if not url.startswith("http"):
            article_map[key] = 'INVALID URL'
            continue

        try:
            full_text = requests.get(url)
            soup = BeautifulSoup(full_text.text, "html.parser")

            try:
                text_content = soup.find('pre')

                if text_content:
                    cleaned_text = ' '.join(text_content.get_text().split())
                    article_map[key] = cleaned_text
                else:
                    article_map[key] = 'FULL TEXT NOT AVAILABLE'

            except AttributeError:
                article_map[key] = 'FULL TEXT NOT AVAILABLE'

        except requests.exceptions.RequestException as e:
            article_map[key] = 'REQUEST FAILED'

update_value(article_map)

for key, value in article_map.items():
    print(f"Title: {key[0]}, Author: {key[1]}")
    print(f"Text: {value[:500]}...\n")
    break
```

```
Title: s, Author: i
Text: THE AMERICAN JOURNAL RAPY FFICIAL PUBLICATION OF THE AMERICAN OCCUPATIONAL
THERAPY ASSOCIATION Vol. Ill, No. 6 1949 November-December TABLE of CONTENTS
ARTICLES Physiological Aid to the Functional Training Donald Covalt, M.D.,
Leonard Yamshon, M.D. and Virginia Nowicki, O.T.R. Harvey August, M.D.
Measurable Factors in Psychiatric Occupational Therapy … 297 Ronald
Beals, O.T.R. O. R. Yoder, M.D. Problems Encountered in Dealing with Handicapped
and Emotionally Disturbed Children …
```

1.0.8 Full Code Implementation

To maximize efficiency, the code is designed to divide tasks and execute them asynchronously, ensuring a quick runtime. This Python script leverages asyncio and aiohttp to perform non-blocking

web scraping, efficiently gathering article links and their full-text content from the Internet Archive Scholar. The scraper handles rate limits, organizes article metadata, retrieves full-text content, and saves results into JSON files. This version processes a subset of 2 million articles, with plans to scale up to the full 85 million articles by partitioning JSON files, each containing data for 2 million articles.

1.0.9 Code Breakdown

1. Rate-Limited Page Fetching (fetch_page function):

- This function fetches the HTML content of a page from a given URL, handling HTTP 429 (rate-limit) errors with exponential backoff.
- It uses a semaphore to limit concurrent requests, preventing overwhelming the server.
- It retries up to three times if a rate-limit error is encountered, doubling the delay between attempts to respect the server's limits.

2. Collecting Article URLs (fetch_articles function):

- This function gathers URLs for articles by generating search result pages with different offsets.
- For each page, it extracts article links for "microtext" and "PDF" types by looking for specific titles in anchor tags.
- If a page has no new articles, the function stops fetching further pages to avoid redundant requests.
- It returns a dictionary with two sets of URLs: one for "microtext" links and one for "PDF" links.

3. Fetching Individual Article Links (fetch_article function):

- For each URL in the collected list, this function fetches an article's page and identifies links for full-text downloads by looking for a "FULLTEXT download" anchor.
- If found, it returns a unique identifier for the article along with the full-text URL. It uses a counter to ensure a unique identifier if the article lacks one.

4. Generating Full-Text URL Mapping (fetch_full_text function):

- Given a list of URLs, this function generates a dictionary mapping unique article identifiers to full-text URLs.
- It uses a semaphore to manage concurrency, ensuring that requests don't overwhelm the server.
- The function returns a dictionary where keys are article identifiers and values are their respective full-text URLs.

5. Extracting and Cleaning Full Text Content (fetch_full_text_content function):

- For each full-text URL, this function fetches the HTML content, then parses and extracts the main text within a
  tag.
- It cleans the extracted text by removing excessive whitespace and newlines. If the text is not available, it returns a placeholder message, "FULL TEXT NOT AVAILABLE."
- This function helps ensure the hashmap contains clean, usable text for each article.

6. Updating the Article Map (update_value function):

- This function updates the article map by replacing each URL with its cleaned text content.
- It iterates over the article_map dictionary, fetching and cleaning the text content for each entry.

7. **Main Execution Function (main function):**

- This orchestrator function first calls fetch_articles to gather URLs.
- It then calls fetch_full_text to obtain the full-text URLs for microtext articles and update_value to replace the URLs with the actual text content.
- Finally, it saves the microtext content and PDF links into JSON files: "microtext_data.json" (contains article identifiers and full text) and "pdf_links.json" (contains PDF URLs).

[ ]:
```
import asyncio
import aiohttp
from bs4 import BeautifulSoup
import json

async def fetch_page(session, url, semaphore):
    # Use semaphore to control the number of concurrent requests to avoid␣
 ↪server overload
    async with semaphore:
        retries = 3  # Set the maximum number of retries
        delay = 2  # Initial delay in seconds for backoff strategy if rate␣
 ↪limit error occurs

        # Attempt to fetch the page content, retrying if rate limits are␣
 ↪encountered
        for attempt in range(retries):
            try:
                # Make a GET request to the specified URL with a timeout of 10␣
 ↪seconds
                async with session.get(url, timeout=10) as response:
                    if response.status == 429:
                        print(f"Rate limit hit for {url}. Retrying in {delay}␣
 ↪seconds...")
                        await asyncio.sleep(delay)
                        delay *= 2  # Increase delay exponentially for each␣
 ↪retry
                        continue

                    response.raise_for_status()
                    # Return the HTML content of the page if successfully␣
 ↪fetched
                    return await response.text()

            except aiohttp.ClientResponseError as e:
                print(f"Error fetching {url}: {e}")
```

```
                # Retry if rate limit error occurs and retries are still␣
↪available
                if response.status == 429 and attempt < retries - 1:
                    await asyncio.sleep(delay)
                    delay *= 2
                else:
                    return None

            # Handle unexpected errors that may occur (e.g., network issues)
            except Exception as e:
                print(f"Unexpected error fetching {url}: {e}")
                return None  # Return None if an unexpected error occurs


async def fetch_articles():
    base_url = "https://scholar.archive.org/search?q=*&offset={}"
    articles_per_page = 15  # Number of articles per page, used to calculate␣
↪offsets
    max_offset = 2000000  # Fetching 2 million articles
    offsets = list(range(0, max_offset, articles_per_page))  # [0, 15, 30, 45,␣
↪60...]
    articles_found = {"microtext": set(), "pdf": set()}  # Store full text urls␣
↪and pdf urls
    semaphore = asyncio.Semaphore(5)

    # Create a session for making requests
    async with aiohttp.ClientSession() as session:
        tasks = []  # List to hold fetch tasks
        # Loop through each offset to create tasks for each page
        for offset in offsets:
            url = base_url.format(offset)
            tasks.append(fetch_page(session, url, semaphore))  # Add fetch task␣
↪to list
            await asyncio.sleep(0.1)

        # Process each completed task as they finish
        for future in asyncio.as_completed(tasks):
            text = await future  # Await the result of the fetch task
            if text:  # Check if text was successfully fetched
                soup = BeautifulSoup(text, "html.parser")
                new_articles = 0

                for a in soup.find_all("a"):
                    if 'title' in a.attrs and 'href' in a.attrs:
                        if a['title'] == 'read fulltext microfilm' or␣
↪a['title'] == 'fulltext access':
```

```
                            url = a['href']
                            if url not in articles_found["microtext"]:
                                articles_found["microtext"].add(url)
                                new_articles += 1
                        elif a['title'] == 'fulltext PDF download':
                            pdf_url = a['href']
                            if pdf_url not in articles_found["pdf"]:
                                articles_found["pdf"].add(pdf_url)
                                new_articles += 1

                # Stop fetching if no new articles were found on this page
                if new_articles == 0:
                    print("Reached the last page with no new articles.")
                    break

    # Convert sets to lists for JSON serialization
    articles_found["microtext"] = list(articles_found["microtext"])
    articles_found["pdf"] = list(articles_found["pdf"])
    return articles_found  # Return the dictionary of found articles

# Fetch individual article details, including full-text URL, if available
async def fetch_article(session, url, semaphore, unique_counter):
    # Use a semaphore to control concurrency and avoid rate limiting
    async with semaphore:
        try:
            # Send a GET request to fetch the article page
            async with session.get(url, timeout=10) as response:
                response.raise_for_status()  # Raise error for unsuccessful␣
 ↪requests
                text = await response.text()  # Retrieve HTML content as text
                soup = BeautifulSoup(text, "html.parser")  # Parse HTML content

                # Look for all anchor tags on the page
                for a in soup.find_all('a'):
                    anchor_text = a.get_text().strip()  # Get the text within␣
 ↪the anchor
                    # Check if the anchor text matches "FULLTEXT download"␣
 ↪(ignoring spaces)
                    if anchor_text.replace(" ", "") == "FULLTEXTdownload":
                        if 'href' in a.attrs:
                            # Form the full-text URL by appending the anchor's␣
 ↪href
                            fulltext_url = "https://archive.org" + a['href']
                            # Attempt to find a unique identifier within the␣
 ↪page
                            identifier = soup.find('span',␣
 ↪itemprop="identifier")
```

```
                            if identifier and identifier.get_text():
                                id_text = identifier.get_text()
                            else:
                                # If no identifier found, create one using the␣
 ↪unique counter
                                id_text = f"unique-{unique_counter}"
                            return id_text, fulltext_url  # Return the␣
 ↪identifier and URL
        except Exception as e:
            # Handle exceptions and log error message
            print(f"Failed to fetch article: {url}, error: {e}")
        return None  # Return None if an error occurred or no link was found

# Fetch full-text URLs for a list of article pages
async def fetch_full_text(url_list):
    if not url_list:
        print("No URLs found to fetch full text.")
        return {}

    article_map = {}  # Dictionary to store article identifiers and full-text␣
 ↪URLs
    semaphore = asyncio.Semaphore(5)
    unique_counter = 1  # Counter for generating unique identifiers if needed

    # Create an async session for making HTTP requests
    async with aiohttp.ClientSession() as session:
        tasks = []
        for url in url_list:
            tasks.append(fetch_article(session, url, semaphore, unique_counter))
            unique_counter += 1  # Increment counter for unique identifier␣
 ↪generation

        # Process each task as it completes
        for future in asyncio.as_completed(tasks):
            result = await future
            if result:
                identifier, fulltext_url = result
                article_map[identifier] = fulltext_url  # Add to the article map

    return article_map  # Return dictionary with article identifiers and␣
 ↪full-text URLs

# Fetch the content of each full-text URL and clean it
async def fetch_full_text_content(session, key_url, semaphore):
    key, url = key_url  # Extract key and URL
    if not url.startswith("http"):
        return key, 'INVALID URL'
```

```
    async with semaphore:
        try:
            async with session.get(url, timeout=10) as response:
                response.raise_for_status()
                text = await response.text()
                soup = BeautifulSoup(text, "html.parser")
                text_content = soup.find('pre')

                if text_content:
                    # Clean the text content by removing excess whitespace
                    cleaned_text = ' '.join(text_content.get_text().split())
                    return key, cleaned_text
                else:
                    # Return message if full text is not found
                    return key, 'FULL TEXT NOT AVAILABLE'
        except Exception as e:
            # Return error message if request fails
            return key, f'REQUEST FAILED: {e}'

# Update article map by replacing URLs with cleaned full-text content
async def update_value(article_map):
    semaphore = asyncio.Semaphore(5)
    async with aiohttp.ClientSession() as session:
        # Create fetch tasks for each URL in the article map
        tasks = [fetch_full_text_content(session, item, semaphore) for item in␣
 ↪article_map.items()]

        for future in asyncio.as_completed(tasks):
            key, value = await future
            article_map[key] = value  # Update the article map with fetched␣
 ↪content

async def main():
    article_urls = await fetch_articles()  # Fetch all article URLs

    if article_urls:
        # Log the number of articles found
        print(f"Found {len(article_urls['microtext'])} microtext articles.")
        print(f"Found {len(article_urls['pdf'])} PDF articles.")

        # Fetch full-text URLs for microtext articles and update with actual␣
 ↪content
        article_map = await fetch_full_text(article_urls["microtext"])
        await update_value(article_map)  # Replace URLs with full-text content

        # Save microtext content to JSON file with identifiers and full text
```

```
        with open("microtext_data.json", "w") as f:
            json.dump(article_map, f, indent=2)

        # Save PDF links to separate JSON file
        with open("pdf_links.json", "w") as f:
            json.dump(article_urls["pdf"], f, indent=2)
    else:
        # Log if no articles were found
        print("No articles found.")

# Run the main function
if __name__ == "__main__":
    asyncio.run(main())  # Start the asynchronous main function
```

1.0.10 Choosing Our AI Model

After gathering our data, the next step is to choose an AI model that best meets our criteria for summarization and fine-tuning. The models we will examine include BART, Longformer Encoder-Decoder (LED), FLAN, PEGASUS, and REFORMER. My task was to evaluate BART, LED, and FLAN by testing their capabilities on text content from one of the scraped articles, though the output from the initial test was unintelligible.

[ ]:
```
from transformers import pipeline

bart_summarizer = pipeline("summarization", model="facebook/bart-large-cnn",␣
 ↪device=0)

# Since BERT does not have a summarization tool, using LED since its powered by␣
 ↪BERT
led_summarizer = pipeline("summarization", model="allenai/led-base-16384",␣
 ↪device=0)

flan_summarizer = pipeline("summarization", model="google/flan-t5-large",␣
 ↪device=0)
```

```
source_text = "Intelligence is a multifaceted and elusive concept that has long␣
 ↪challenged psychologists, philosophers, and computer scientists. An attempt␣
 ↪to capture its essence was made in 1994 by a group of 52 psychologists who␣
 ↪signed onto a broad definition published in an editorial about the science␣
 ↪of intelligence [Got97]. The consensus group defined intelligence as a very␣
 ↪general mental capability that, among other things, involves the ability to␣
 ↪reason, plan, solve problems, think abstractly, comprehend complex ideas,␣
 ↪learn quickly and learn from experience. This definition implies that␣
 ↪intelligence is not limited to a specific domain or task, but rather␣
 ↪encompasses a broad range of cognitive skills and abilities. Building an␣
 ↪artificial system that exhibits the kind of general intelligence captured by␣
 ↪the 1994 consensus definition is a long-standing and ambitious goal of AI␣
 ↪research. In early writings, the founders of the modern discipline of␣
 ↪artificial intelligence (AI) research called out sets of aspirational goals␣
 ↪for understanding intelligence [MMRS06]. Over decades, AI researchers have␣
 ↪pursued principles of intelligence, including generalizable mechanisms for␣
 ↪reasoning (e.g., [NSS59], [LBFL93]) and construction of knowledge bases␣
 ↪containing large corpora of commonsense knowledge [Len95]. However, many of␣
 ↪the more recent successes in AI research can be described as being narrowly␣
 ↪focused on well-defined tasks and challenges, such as playing chess or Go,␣
 ↪which were mastered by AI systems in 1996 and 2016, respectively. In the␣
 ↪late-1990s and into the 2000s, there were increasing calls for developing␣
 ↪more general AI systems (e.g., [SBD+96]) and scholarship in the field has␣
 ↪sought to identify principles that might underly more generally intelligent␣
 ↪systems (e.g., [Leg08, GHT15]). The phrase, "artificial general␣
 ↪intelligence" (AGI), was popularized in the early-2000s (see [Goe14]) to␣
 ↪emphasize the aspiration of moving from the "narrow AI", as demonstrated in␣
 ↪the focused, real-world applications being developed, to broader notions of␣
 ↪intelligence, harkening back to the long-term aspirations and dreams of␣
 ↪earlier AI research. We use AGI to refer to systems that demonstrate broad␣
 ↪capabilities of intelligence as captured in the 1994 definition above, with␣
 ↪the additional requirement, perhaps implicit in the work of the consensus␣
 ↪group, that these capabilities are at or above human-level. We note however␣
 ↪that there is no single definition of AGI that is broadly accepted, and we␣
 ↪discuss other definitions in the conclusion section."

print("BART Summary:", bart_summarizer(source_text, max_length=130,␣
 ↪min_length=30, do_sample=False))
print("LED Summary:", led_summarizer(source_text, max_length=130,␣
 ↪min_length=30, do_sample=False))
print("FLAN Summary:", flan_summarizer(source_text, max_length=130,␣
 ↪min_length=30, do_sample=False))
```

[ ]: