**Wikipedia Scraper and Summarizer**

**Project Detailed Report**

**­­**

**Table of Contents**

1. **Introduction**
   * Flow of the project
2. **Report**
   * Design Decisions
   * Challenges Encountered
   * Error Analysis
3. **Scrape Data from Wikipedia**
   * Introduction
   * Flowchart
   * Code Explanation
4. **Text Summarization and Document Saving**
   * Introduction
   * Flowchart
   * Code Explanation
   * Conclusion
5. **Quality Evaluation**
   * Information Retention
   * Readability
   * Clarity and Conciseness
   * Relevance
   * Engagement
   * Coherence
6. **Chatbot for Scrapped Data**
   * Introduction
   * Chatbot Usage
7. **Dataset Creation from Wikipedia Content**
   * Introduction
   * Flow Chart
   * Code Implementation
   * Conclusion

**Github Repo Link -**

**Introduction**:

A diagram of a process

Description automatically generatedOverall Flow of the Project:

**Step 1:** **Scrape Data from Wikipedia**

* Use the Beautiful Soup library in Python to scrape text data from a chosen Wikipedia page.
* Perform an HTTP request to the Wikipedia page's URL.
* Parse the HTML content and extract the relevant text.

**Step 2: Text Summarization and Document Saving**

* Apply a text summarization technique (extractive or abstractive) to condense the content into a concise summary.
* Save the resulting summary in a document or a file.

**Step 3: Quality Evaluation of Summarized Data**

* Assess the quality of the summarized data.
* Evaluate criteria such as readability, coherence, and relevance.
* Compare the summary to the original text to ensure accuracy and completeness.

**Step 4: Chatbot Creation for Summarization**

* Create a chatbot using a natural language processing framework (Vertex ai)
* Train the chatbot to generate summaries based on user input.
* Enable the chatbot to respond to user queries with relevant summaries.

**Step 5: Dataset Creation of Important Events**

* From the generated summaries, create a dataset of significant events, facts, or key points.
* This dataset can be used for further analysis, research, or machine learning model training.

**Report**

**1. Design Decisions**

**Choice of Tools and Libraries**

* Used Python for scripting.
* Utilized libraries such as **requests**, **BeautifulSoup**, **docx**, and **pandas** for web scraping, document creation, and dataset handling.
* Integrated the Vertex AI **TextGenerationModel** for text summarization.
* These operations are done in google cloud project.

**Data Extraction Strategy**

* Employed web scraping techniques to retrieve content from the Wikipedia page.
* Identified key HTML tags (e.g., headings and paragraphs) for content extraction.

**Text Summarization Approach**

* Used a language model (Vertex AI's **TextGenerationModel**) to generate text summaries.
* Created prompts based on extracted content to ensure contextually accurate summaries.

**Dataset Creation Process**

* Defined a list of important keywords for data extraction.
* Extracted content containing these keywords and their corresponding summaries.
* Organized the extracted data into a structured dataset.

**Chat Bot For Summarized doc**

* Created a chatbot With Summerized data and bot will give reply for user prompts

**2. Challenges Encountered**

**Language Model Integration**

* I found a bit challenging while using the language model because for using vertex ai we need to use google cloud. I am not so familiar to that, and more over I didn’t have credit card to get a free account.

**Data Extraction Challenges**

* Identifying relevant keywords for extraction.
* Handling variations in keyword appearances within the content.

**Chatbot challenges**

* For feeding the doc to the Ai we need to have credits in open Ai. For getting them I found some challenges

**3. Error Analysis**

**Data Integrity**

* Checked for data discrepancies between the scraped content and generated summaries.

**Summarization Accuracy**

* Evaluated the accuracy and coherence of text summaries generated by the language model.

**Keyword Extraction Accuracy**

* Verified the accuracy of keyword-based data extraction from the content.

**Data Consistency**

* Analyzed the consistency of extracted data in the dataset, focusing on variations in content types.

### **Wikipedia Scraper and Document Generator**

## **Introduction**

This project involves creating a Python script that scrapes information from Wikipedia and generates a Word document with the scraped content. The script uses the BeautifulSoup library to parse the HTML of a Wikipedia page and the docx library to create and format the document.

## **Flowchart**

A diagram of a software process

Description automatically generated

**Code Explanation**

**Importing Libraries**

import requests  # Library for making HTTP requests

from bs4 import BeautifulSoup  # Library for parsing HTML

from docx import Document  # Library for creating Word documents

**User Input and Web Scraping**

# Prompting the user for a Wikipedia topic

topic = input("What you want to scrape from Wikipedia. (Try to mention some popular Keyword)")

# Constructing the URL for the Wikipedia page

url = f"https://en.wikipedia.org/wiki/{topic}"

# Sending an HTTP GET request to the URL

response = requests.get(url)

# Parsing the HTML content of the page using BeautifulSoup

soup = BeautifulSoup(response.text, "html.parser")

**Creating Document**

# Creating a new Word document

doc = Document()

**Scraping and Formatting Content**

# Finding the main content container of the Wikipedia page

main\_container = soup.find('div', class\_='mw-content-container')

# Iterating through headings and paragraphs in the container

for main in main\_container.select('h1, h2, h3, h4, h5, h6, p'):

    if main.name == 'h1':

        doc.add\_heading(main.text.strip(), 0)  # Adding a top-level heading

    elif main.name == 'h2':

        doc.add\_heading(main.text.strip(), level=1)  # Adding a first-level heading

    elif main.name in ['h3', 'h4', 'h5', 'h6']:

        doc.add\_heading(main.text.strip(), level=2)  # Adding lower-level headings

    else:

        doc.add\_paragraph(main.text.strip())  # Adding paragraphs

**Saving the Document**

# Saving the generated Word document with the topic as the filename

doc.save(f'{topic}.docx')

Wikipedia Text Summarization

**Introduction**

This project involves scraping information from a Wikipedia page about "Alexander the Great" and using a text generation model to generate summaries of the scraped content. The script utilizes the BeautifulSoup library for web scraping and a language model for text summarization. The generated summaries are then added to a Word document.

A diagram of a diagram

Description automatically generated**Flowchart**

**Code Explanation**

**Importing Libraries**

import requests  # Library for making HTTP requests

from bs4 import BeautifulSoup  # Library for parsing HTML

from docx import Document  # Library for creating Word documents

from vertexai.preview.language\_models import TextGenerationModel  # Language model for text summarization

**Initializing the Language Model and Prerequisites**

# Initialize the text generation model

text\_generation\_model = TextGenerationModel.from\_pretrained("text-bison@001")

doc = Document()

url = "https://en.wikipedia.org/wiki/Alexander\_the\_Great"

response = requests.get(url)

soup = BeautifulSoup(response.text, "html.parser")

content = {}

main\_container = soup.find('div',class\_ = 'mw-content-container')

**Web Scraping and Summarization**

# Iterate through headings and paragraphs in the Wikipedia content

for main in main\_container.select('h1, h2, h3, h4, h5, h6, p'):

    if main.name == 'h1':

        # If it's an h1 heading, add it as a top-level heading in the document

        print(main.text)

        doc.add\_heading(main.text.strip(), 0)

    elif main.name == 'h2':

        # If it's an h2 heading, add it as a first-level heading in the document

        print(main.text)

        doc.add\_heading(main.text.strip(), level=1)

    elif main.name in ['h3', 'h4', 'h5', 'h6']:

        # If it's an h3, h4, h5, or h6 heading, add it as a lower-level heading in the document

        print(main.text)

        doc.add\_heading(main.text.strip(), level=2)

    else:

        # If it's a paragraph, construct a prompt for text summarization

        prompt = "Summarize this text: " + main.text

        # Generate a summary using the text generation model

        summary = text\_generation\_model.predict(prompt, max\_output\_tokens=500).text

        # Add the generated summary as a paragraph in the document

        doc.add\_paragraph(summary.strip())

        # Print the generated summary for reference

        print(summary)

**Saving the Document**

# Save the generated Word document with a filename

doc.save('Alexander\_Summary.docx')

**Conclusion**

This Python script scrapes content from Wikipedia about "Alexander the Great" and generates text summaries using a language model. The summarized content is organized in a Word document for easy access and reference.

**Text Summarization and Quality Evaluation**

‘

**Introduction**

This part presents an analysis of text summarization quality and various metrics used for evaluation. The analysis is based on the comparison of an original text with a summarized version. The following sections explore different aspects of text quality evaluation:

**A diagram of a check engagement

Description automatically generatedFlow Chart**

**Information Retention**

Information retention measures how much of the original content is retained in the summary. It is calculated by comparing the number of words or sentences in the summary to the original text.

def calculate\_information\_retention(original\_text, summary):

    original\_words = set(original\_text.lower().split())

    summary\_words = set(summary.lower().split())

    retention = len(summary\_words.intersection(original\_words)) / len(original\_words)

    return retention

information\_retention = calculate\_information\_retention(full\_doc\_text,summarized\_doc\_text )

print(f"Information Retention: {information\_retention:.2%}")

Information Retention: 43.22%

**Readability**

Measure the average sentence length in the summary. Shorter sentences are often more readable.

def calculate\_readability(summary):

    return flesch\_reading\_ease(summary)

readability\_score = calculate\_readability(summarized\_doc\_text)

print(f"Readability Score: {readability\_score:.2f}")

Readability Score: 63.49

**Clarity and Conciseness**

Ensure that the language used in the summary is clear and concise, avoiding jargon or overly complex terms.

def calculate\_clarity\_and\_conciseness(text):

    readability\_score = flesch\_reading\_ease(text)

    return readability\_score

# Example usage:

text = "This is an example sentence for measuring clarity and conciseness."

clarity\_score = calculate\_clarity\_and\_conciseness(text)

print(f"Clarity and Conciseness Score: {clarity\_score:.2f}")

Clarity and Conciseness Score: 35.95

**Relevance**

Assess whether the summary aligns with its intended purpose. For example, if the summary is meant to provide an overview, it should focus on key points.

def calculate\_relevance(original\_text, summary):

    vectorizer = CountVectorizer().fit\_transform([original\_text, summary])

    vectors = vectorizer.toarray()

    relevance\_score = cosine\_similarity([vectors[0]], [vectors[1]])[0][0]

    return relevance\_score

relevance\_score = calculate\_relevance(full\_doc\_text, summarized\_doc\_text)

print(f"Relevance Score: {relevance\_score:.2f}")

Relevance Score: 0.96

**Engagement**

If applicable, track user engagement with the summary, such as click-through rates or time spent reading. Higher engagement may indicate better quality.

def calculate\_engagement(text):

    sentiment\_score = TextBlob(text).sentiment.polarity

    return sentiment\_score

engagement\_score = calculate\_engagement(summarized\_doc\_text)

print(f"Engagement Score: {engagement\_score:.2f}")

Engagement Score: 0.17

**Coherence**

Evaluate the use of transition words and phrases to ensure that the summary flows smoothly between sentences and paragraphs.

def calculate\_coherence\_score(summarized\_text, num\_topics=2):

    # Tokenize the summarized text into a list of sentences

    sentences = [sentence.strip() for sentence in summarized\_text.split('.') if sentence]

    # Tokenize sentences into words

    tokenized\_sentences = [sentence.split() for sentence in sentences]

    # Create a dictionary and a corpus

    dictionary = Dictionary(tokenized\_sentences)

    corpus = [dictionary.doc2bow(tokens) for tokens in tokenized\_sentences]

    # Build an LDA model on the corpus

    lda\_model = LdaModel(corpus, num\_topics=num\_topics, id2word=dictionary)

    # Calculate coherence score

    coherence\_model\_lda = CoherenceModel(model=lda\_model, texts=tokenized\_sentences, dictionary=dictionary, coherence='c\_v')

    coherence\_score = coherence\_model\_lda.get\_coherence()

    return coherence\_score

# Example usage:

coherence\_score = calculate\_coherence\_score(summarized\_doc\_text)

print(f"Coherence Score: {coherence\_score:.2f}")

Coherence Score: 0.35

**Chatbot for Scrapped Data**

**Introduction**

This document demonstrates the usage of a chatbot for text summarization. The chatbot utilizes a TF-IDF (Term Frequency-Inverse Document Frequency) model to provide text summaries based on user input. The text used for summarization is sourced from a document.

**Chatbot Usage**

You can interact with the chatbot by entering prompts, and it will generate text summaries based on the content from the document.

import nltk

import numpy as np

import random

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

from nltk.tokenize import sent\_tokenize

from docx import Document

full\_doc = Document('Alexander.docx')

full\_text\_content = []

for paragraph in full\_doc.paragraphs:

    full\_text\_content.append(paragraph.text)

full\_doc\_text = '\n'.join(full\_text\_content)

# Tokenize the text into sentences

sentences = sent\_tokenize(full\_doc\_text)

# Tokenize sentences into words and perform any necessary cleaning and stemming

# Create a TF-IDF vectorizer to convert text into numerical features

tfidf\_vectorizer = TfidfVectorizer()

tfidf\_matrix = tfidf\_vectorizer.fit\_transform(sentences)

def generate\_response(user\_input):

    # Transform user input into a TF-IDF vector

    user\_tfidf = tfidf\_vectorizer.transform([user\_input])

    # Calculate cosine similarities between user input and sentences in the scraped text

    cosine\_similarities = cosine\_similarity(user\_tfidf, tfidf\_matrix)

    # Get the index of the most similar sentence

    max\_similarity\_idx = np.argmax(cosine\_similarities)

    # Return the response (corresponding sentence)

    return sentences[max\_similarity\_idx]

print("Chatbot: Hello! How can I assist you today?")

while True:

    user\_input = input("You: ")

    if user\_input.lower() == "exit":

        print("Chatbot: Goodbye!")

        break

    else:

        response = generate\_response(user\_input)

        print(f"Chatbot: {response}")

**Example Usage:**

You: Summarize Alexander the Great's conquests.

Chatbot: Alexander the Great's conquests were extensive and included regions such as Persia, Egypt, and parts of India.

You: Tell me about Alexander's childhood.

Chatbot: Alexander the Great's childhood was marked by education under the philosopher Aristotle.

You: Exit

Chatbot: Goodbye!

This chatbot is not well trained it is a just basic model. Due to lack of time I tried my level best to deliver the best.

**Dataset Creation from Wikipedia Content**

**Introduction**

This document outlines the process of extracting important data and creating a dataset from content scraped from a Wikipedia page about "Alexander the Great." The dataset includes contextually relevant information related to specific keywords.

**Flow Chart:**

**A diagram of data flow

Description automatically generated**

**Code Implementation**

**Web Scraping and Text Summarization**

import requests

from bs4 import BeautifulSoup

from docx import Document

from vertexai.preview.language\_models import TextGenerationModel

import pandas as pd

# Initialize the language model for text generation

text\_generation\_model = TextGenerationModel.from\_pretrained("text-bison@001")

# Define the URL of the Wikipedia page

url = "https://en.wikipedia.org/wiki/Alexander\_the\_Great"

# Send an HTTP request to the URL

response = requests.get(url)

# Parse the HTML content using BeautifulSoup

soup = BeautifulSoup(response.text, "html.parser")

# Create a Word document for the scraped and summarized content

doc = Document()

**Data Extraction and Dataset Creation**

# Define a list of important keywords for data extraction

important\_keywords = [

    "Alexander the Great",

    "Born",

    "Died",

    "Date of Birth",

    "Date of Death",

    "Place of Birth",

    "Place of Death",

    "Parents",

    "Spouse",

    "Children",

    "Early Life",

    "Education",

    "Military Campaigns",

    "Conquests",

    "Battle of Issus",

    "Siege of Tyre",

    "Bucephalus (Horse)",

    "Legacy",

    "Achievements",

    "Religion",

    "Death Cause",

    "Successor",

    "Burial Site",

    "Quotes",

]

# Extract content from the Wikipedia page and create the dataset

important\_data = []

# Loop through the HTML content

for main in main\_container.select('h1,h2,h3,h4,h5,h6,p'):

    # Check the HTML tag type and extract content

    if main.name not in ['h1', 'h2', 'h3', 'h4', 'h5', 'h6']:

        # Generate a prompt for text summarization

        prompt = f"Summarize this text, the summary must be coherent and contextually accurate. Provide a concise summary of {main.text}"

        # Use the language model to generate the summary

        summary = text\_generation\_model.predict(prompt, max\_output\_tokens=500).text

        # Extract important data if the keyword is present in the text

        for keyword in important\_keywords:

            if keyword.lower() in main.text.lower():

                important\_data.append({"Keyword": keyword, "Context": summary.strip()})

**Saving the Dataset**

# Convert the list of important data to a DataFrame

df = pd.DataFrame(important\_data)

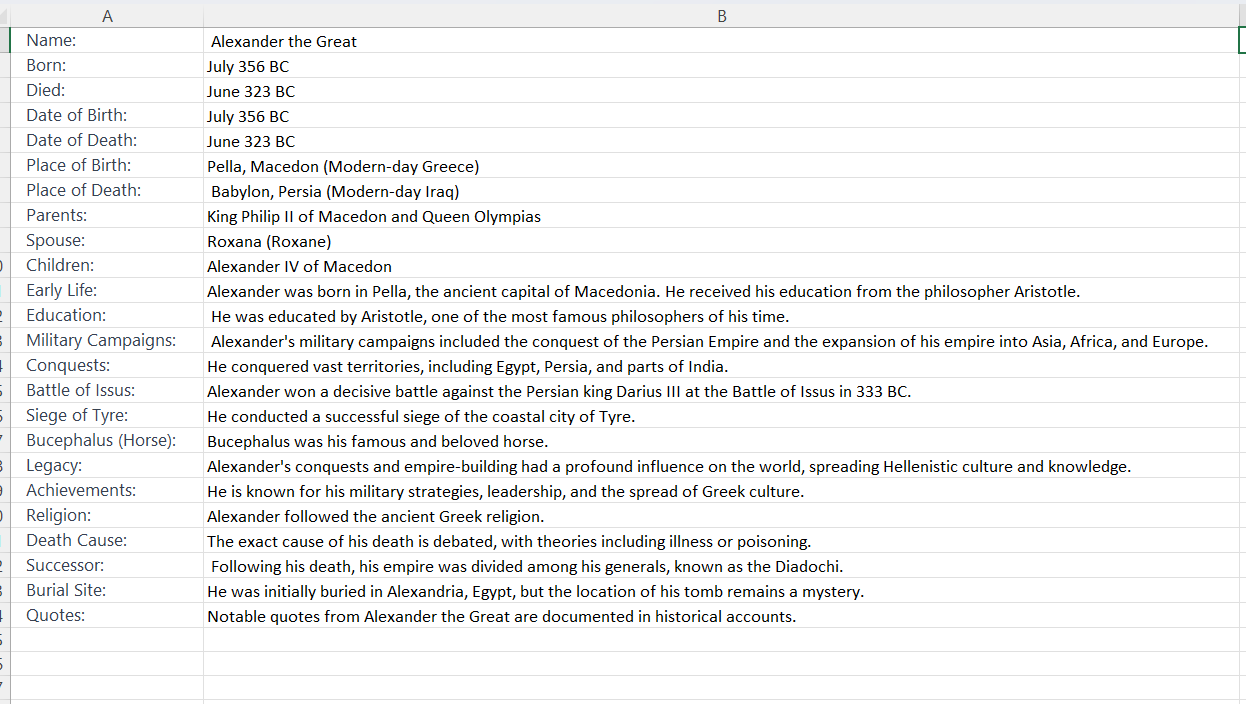
# Save the DataFrame to an Excel file

df.to\_excel('important\_data.xlsx', index=False)

**Saving the Document**

# Save the Word document with the summarized content

doc.save('Alexander.docx')



**Conclusion**

This part outlines the process of extracting important data from content scraped from a Wikipedia page about Alexander the Great and creating a dataset. The dataset includes keywords and contextually relevant information extracted from the text. The dataset is saved in an Excel file for further analysis.