Generative AI

**Course outcomes (Course Skill Set):**

At the end of the course the student will be able to:

* Develop the ability to explore and analyze word embeddings, perform vector arithmetic to investigate word relationships, visualize embeddings using dimensionality reduction techniques
* Apply prompt engineering skills to real-world scenarios, such as information retrieval, text generation.
* Utilize pre-trained Hugging Face models for real-world applications, including sentiment analysis and text summarization.
* Apply different architectures used in large language models, such as transformers, and understand their advantages and limitations.

**Books:**

1. Modern Generative AI with ChatGPT and OpenAI Models: Leverage the Capabilities of OpenAI's LLM for Productivity and Innovation with GPT3 and GPT4, by Valentina Alto, Packt Publishing Ltd, 2023.
2. Generative AI for Cloud Solutions: Architect modern AI LLMs in secure, scalable, and ethical cloud environments, by Paul Singh, Anurag Karuparti ,Packt Publishing Ltd, 2024.
3. **Explore pre-trained word vectors. Explore word relationships using vector arithmetic. Perform arithmetic operations and analyze results.**

**Code:**

from gensim.downloader import load

# Load the pre-trained GloVe model (50 dimensions) print("Loading pre-trained GloVe model (50 dimensions)...") model = load("glove-wiki-gigaword-50")

# Function to perform vector arithmetic and analyze relationships def ewr():

result = model.most\_similar(positive=['king', 'woman'], negative=['man'], topn=1)

print("\nking - man + woman = ?", result[0][0])

print("similarity:",result[0][1])

result = model.most\_similar(positive=['paris', 'italy'], negative=['france'], topn=1)

print("\nparis - france + italy = ?", result[0][0]) print("similarity:",result[0][1])

# Example 4: Find analogies for programming

result = model.most\_similar(positive=['programming'], topn=5) print("\nTop 5 words similar to 'programming':")

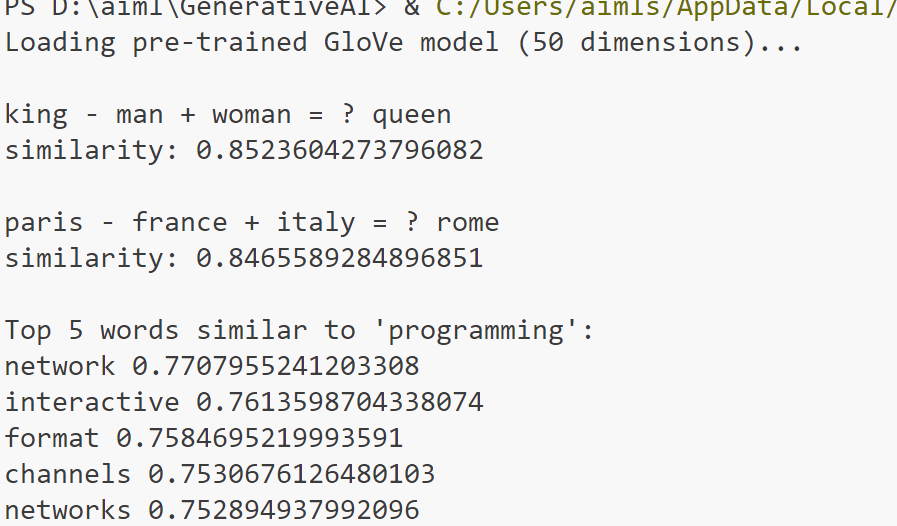
for word, similarity in result: print(word, similarity)

ewr()

**Explanation:**

Gensim is used to perform vector arithmetic and analyze word relationships in a 50-dimensional semantic space. First, the GloVe model (glove-wiki-gigaword-50) is loaded, containing word vectors trained on a large corpus of text. The ewr() function showcases examples of vector arithmetic. It finds the most semantically similar word to a computed vector using the most\_similar method. For instance, king - man + woman predicts a word that reflects the concept of "queen," while paris - france + italy predicts a word like "rome." The function also retrieves the top 5 words most similar to "programming." This illustrates how word embeddings capture meaningful relationships and analogies between words in the vector space.

**Output:**

****

1. **Use dimensionality reduction (e.g., PCA or t-SNE) to visualize word embeddings for Q 1. Select 10 words from a specific domain (e.g., sports, technology) and visualize their embeddings. Analyze clusters and relationships. Generate contextually rich outputs using embeddings. Write a program to generate 5 semantically similar words for a given input.**

**Code:**

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA from gensim.downloader import load

# Dimensionality reduction using PCA def rd(ems):

pca = PCA(n\_components=2) r = pca.fit\_transform(ems) return r

# Visualize word embeddings def visualize(words, ems):

plt.figure(figsize=(10, 6))

for i, word in enumerate(words): x, y = ems[i]

plt.scatter(x, y, marker='o', color='blue') plt.text(x + 0.02, y + 0.02, word, fontsize=12)

plt.show()

# Generate semantically similar words def gsm(word):

sw=model.most\_similar(word, topn=5) for word,s in sw:

print(word,s)

# Load pre-trained GloVe model from Gensim API print("Loading pre-trained GloVe model (50 dimensions)...") model = load("glove-wiki-gigaword-50")

words = ['football', 'basketball', 'soccer', 'tennis', 'cricket'] ems = [model[word] for word in words]

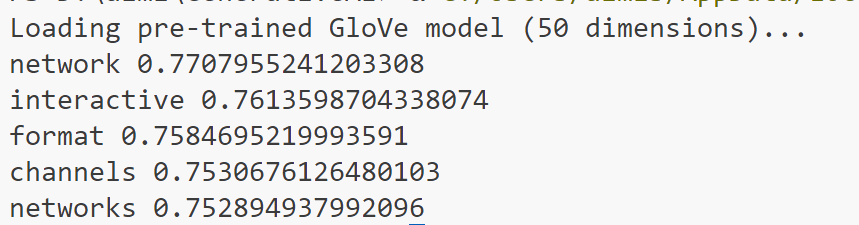
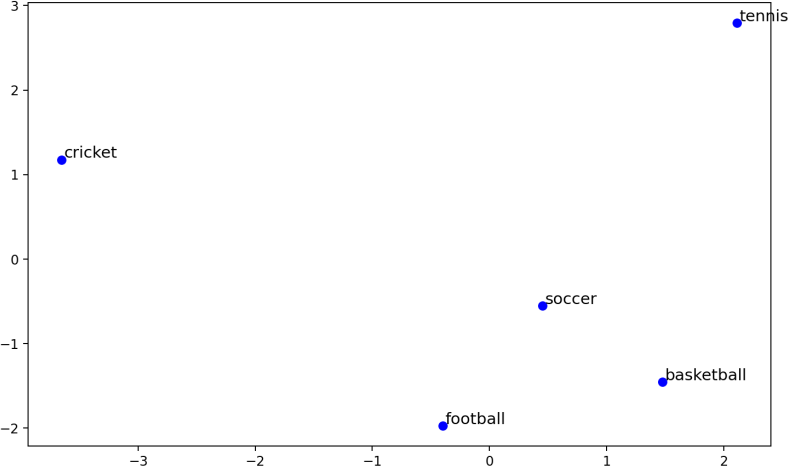
e=rd(ems) visualize(words,e) gsm("programming")

**Explanation:**

Word embeddings are input to PCA (Principal Component Analysis) for dimensionality reduction. Semantic relationships among words are analyzed using a pre-trained GloVe model. The GloVe embeddings (glove- wiki-gigaword-50) are loaded via Gensim, providing 50-dimensional

vector representations of words. The rd function reduces these embeddings to 2 dimensions using PCA, making them suitable for visualization. The visualize function plots the 2D representations of selected words (e.g., "football", "basketball", etc.), displaying their relative semantic positions in the vector space. The gsm function retrieves and prints the top 5 semantically similar words for a given word (e.g., "programming") using the GloVe model's most\_similar method. This process helps illustrate both the semantic relationships between words and their visual clustering in a lower- dimensional space.

**Output:**

****

1. **Train a custom Word2Vec model on a small dataset. Train embeddings on a domain-specific corpus (e.g., legal, medical) and analyze how embeddings capture domain-specific semantics.**

**Code:**

from gensim.models import Word2Vec

#Custom Word2Vec model def cw(corpus):

model = Word2Vec( sentences=corpus,

vector\_size=50, # Dimensionality of word vectors window=5, # Context window size

min\_count=1, # Minimum frequency for a word to be considered workers=4, # Number of worker threads

epochs=10, # Number of training epochs

)

return model

# Analyze trained embeddings def anal(model, word):

sw = model.wv.most\_similar(word, topn=5) for w, s in sw:

print(w,s)

# Example domain-specific dataset (medical/legal/etc.) corpus = [

"The patient was prescribed antibiotics to treat the infection.".split(), "The court ruled in favor of the defendant after reviewing the

evidence.".split(),

"Diagnosis of diabetes mellitus requires specific blood tests.".split(), "The legal contract must be signed in the presence of a witness.".split(), "Symptoms of the disease include fever, cough, and fatigue.".split(),

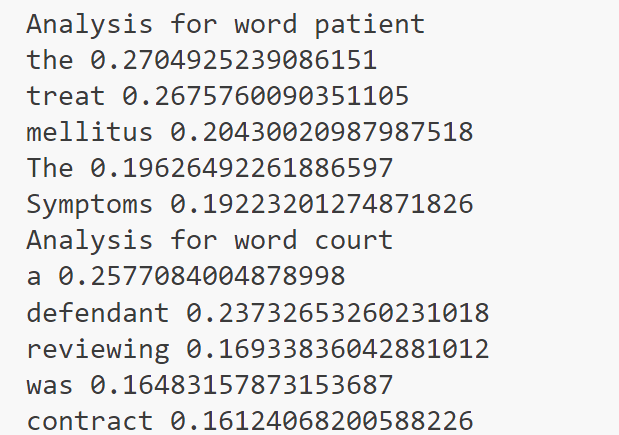
]

model = cw(corpus) print("Analysis for word patient") anal(model, "patient") print("Analysis for word court") anal(model, "court")

**Explanation:**

The cw function trains the model on a small, tokenized dataset of sentences using the Gensim Word2Vec implementation. Key parameters include vector\_size (50-dimensional embeddings), window (context window size), min\_count (minimum word frequency), and epochs (number of training passes). Once trained, the anal function retrieves the top 5 semantically similar words for a given input word using the most\_similar method, revealing the relationships learned from the corpus. In the example, the model is trained on a dataset with sentences about medical and legal topics. Queries like "patient" and "court" analyze the embeddings, returning words from the corpus that are contextually related, showcasing how Word2Vec captures semantic relationships in custom text corpora.

**Output:**

****

1. **Use word embeddings to improve prompts for Generative AI model. Retrieve similar words using word embeddings. Use the similar words to enrich a GenAI prompt. Use the AI model to generate responses for the original and enriched prompts. Compare the outputs in terms of detail and relevance**

**Code:**

from gensim.downloader import load import torch

from transformers import pipeline

# Load pre-trained word embeddings (GloVe)

model = load("glove-wiki-gigaword-50") # GloVe model with 50 dimensions torch.manual\_seed(42)

# Define contextually relevant word enrichment def enrich(prompt):

ep = "" # Start with the original prompt

words = prompt.split() # Split the prompt into words for word in words:

sw = model.most\_similar(word, topn=3) enw=[]

for s,w in sw: enw.append(s)

ep+=" " + " ".join(enw) return ep

# Example prompt to be enriched op = "lung cancer"

ep = enrich(op)

# Display the results print("Original Prompt:", op) print("Enriched Prompt:", ep)

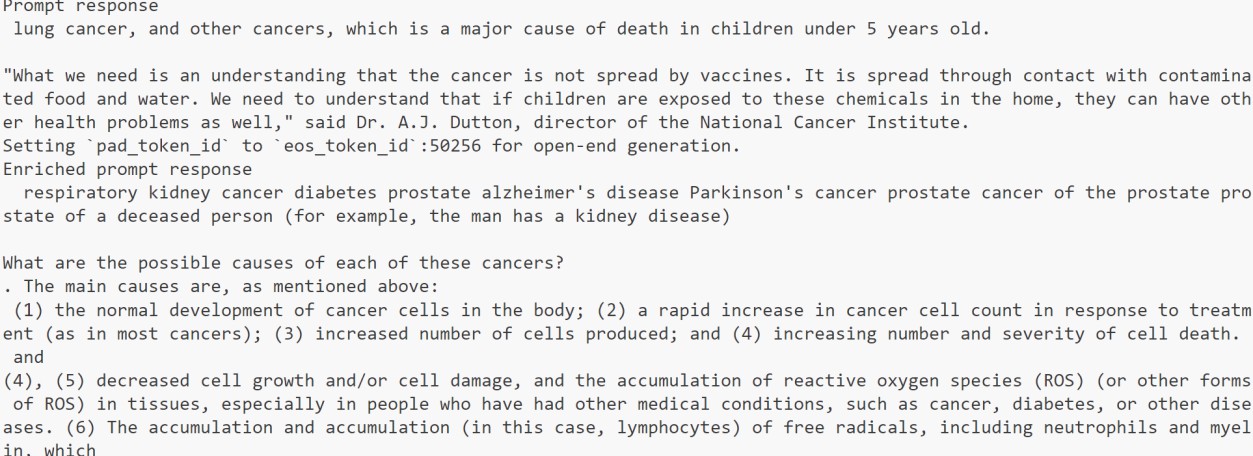
generator = pipeline("text-generation", model="gpt2", tokenizer="gpt2") response = generator(op, max\_length=200, num\_return\_sequences=1, no\_repeat\_ngram\_size=2, top\_p=0.95, temperature=0.7)

print("Prompt response\n",response[0]["generated\_text"]) response = generator(ep, max\_length=200, num\_return\_sequences=1, no\_repeat\_ngram\_size=2, top\_p=0.95, temperature=0.7) print("Enriched prompt response\n",response[0]["generated\_text"])

**Explanation**

Given prompt is enriched using pre-trained **GloVe word embeddings** and then generate responses using a **GPT-2 model**. The enrich() function adds contextually similar words to the original prompt by finding the most similar words for each term in the prompt using GloVe. After enriching the prompt, both the original and enriched prompts are fed into the GPT-2 model to generate responses. The model's output is controlled with parameters like max\_length, top\_p, and temperature to ensure diversity and coherence in the generated text. This approach allows for comparing how enriching a prompt can affect the generated response.

**Output:**

****

1. **Use word embeddings to create meaningful sentences for creative tasks. Retrieve similar words for a seed word.Create a sentence or story using these words as a starting point. Write a program that: Takes a seed word. Generates similar words. Constructs a short paragraph using these words.**

**Code:**

from gensim.downloader import load import random

# Load the pre-trained GloVe model

print("Loading pre-trained GloVe model (50 dimensions)...") model = load("glove-wiki-gigaword-50")

print("Model loaded successfully!")

# Function to construct a meaningful paragraph def create\_paragraph(iw, sws):

paragraph = f"The topic of {iw} is fascinating, often linked to terms like

"

random.shuffle(sws) # Shuffle to add variety for word in sws:

paragraph += str(word) + ", "

paragraph = paragraph.rstrip(", ") + "." return paragraph

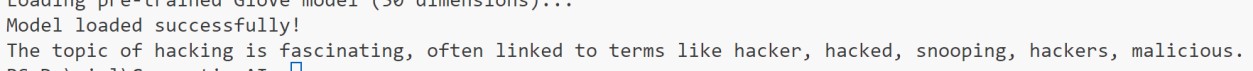
iw = "hacking"

sws = model.most\_similar(iw, topn=5) words = [word for word, s in sws] paragraph = create\_paragraph(iw, words) print(paragraph)

**Explanation:**

The code starts by loading the GloVe model to find the top 5 most similar words to the input word (iw, in this case, "hacking"). These similar words are obtained using the most\_similar method, which identifies words with close semantic meaning. The create\_paragraph function constructs a paragraph by concatenating the input word with its similar terms in a sentence structure. The similar words are shuffled to add variation, and the paragraph is formatted to end cleanly. This creates a contextualized, short descriptive paragraph, showcasing relationships between the input word and related terms.

**Output:**

****

1. **Use a pre-trained Hugging Face model to analyze sentiment in text. Assume a real-world application, Load the sentiment analysis pipeline. Analyze the sentiment by giving sentences to input.**

**Code:**

from transformers import pipeline

# Load the pre-trained sentiment analysis pipeline sentiment\_analyzer = pipeline("sentiment-analysis")

customer\_feedback = [

"The product is amazing! I love it!", "Terrible service, I am very disappointed.",

"This is a great experience, I will buy again.",

"Worst purchase I’ve ever made. Completely dissatisfied.", "I'm happy with the quality, but the delivery was delayed."

]

for feedback in customer\_feedback:

sentiment\_result = sentiment\_analyzer(feedback) sentiment\_label = sentiment\_result[0]['label'] sentiment\_score = sentiment\_result[0]['score']

# Display sentiment results print(f"Feedback: {feedback}")

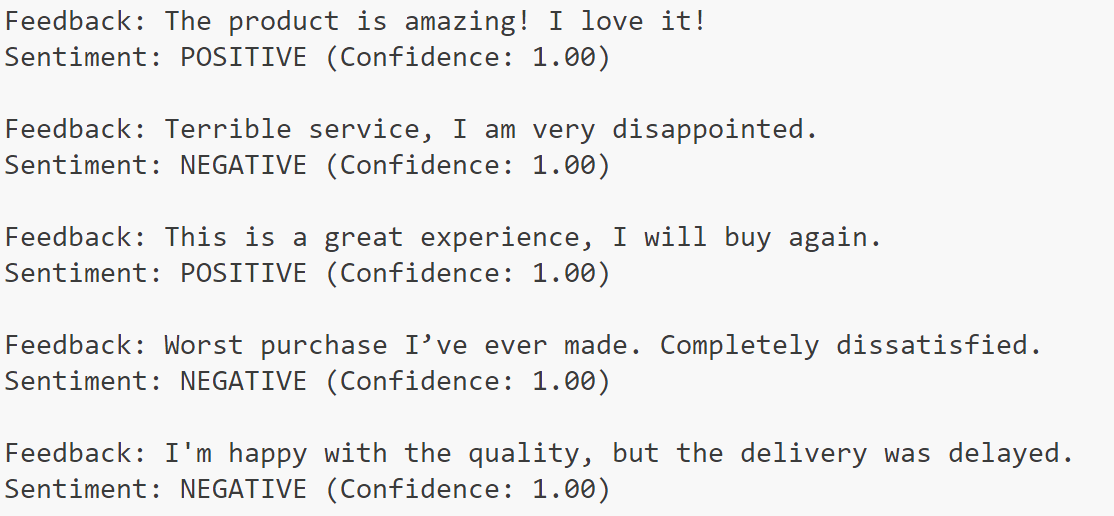
print(f"Sentiment: {sentiment\_label} (Confidence:

{sentiment\_score:.2f})\n")

**Explanation:**

This Python code analyzes the sentiment of customer feedback using a pre-trained sentiment analysis model from the **Hugging Face Transformers library**. The pipeline("sentiment-analysis") function initializes a sentiment analysis model that categorizes text as either "POSITIVE" or "NEGATIVE" with an associated confidence score. A list of feedback comments is iterated through, and for each comment, the model predicts the sentiment label and its confidence. The results, including the original feedback, sentiment, and confidence score, are printed. This approach simplifies sentiment analysis tasks, making it easy to process and interpret text data.

**Output:**

****

1. **Summarize long texts using a pre-trained summarization model using Hugging face model. Load the summarization pipeline. Take a passage as input and obtain the summarized text.**

**Code:**

from transformers import pipeline

# Load the pre-trained summarization pipeline summarizer = pipeline("summarization")

# Function to summarize a given passage def summarize\_text(text):

# Summarizing the text using the pipeline

summary = summarizer(text, max\_length=150, min\_length=50, do\_sample=False) return summary[0]['summary\_text']

text = """

Natural language processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and humans through natural language.

The ultimate goal of NLP is to enable computers to understand, interpret, and generate human language in a way that is valuable.

NLP techniques are used in many applications, such as speech recognition, sentiment analysis, machine translation, and chatbot functionality.

Machine learning algorithms play a significant role in NLP, as they help computers to learn from vast amounts of language data and improve their ability to process and generate text.

However, NLP still faces many challenges, such as handling ambiguity, understanding context, and processing complex linguistic structures. Advances in NLP have been driven by deep learning models, such as transformers, which have significantly improved the performance of many NLP tasks.

"""

# Get the summarized text summarized\_text = summarize\_text(text)

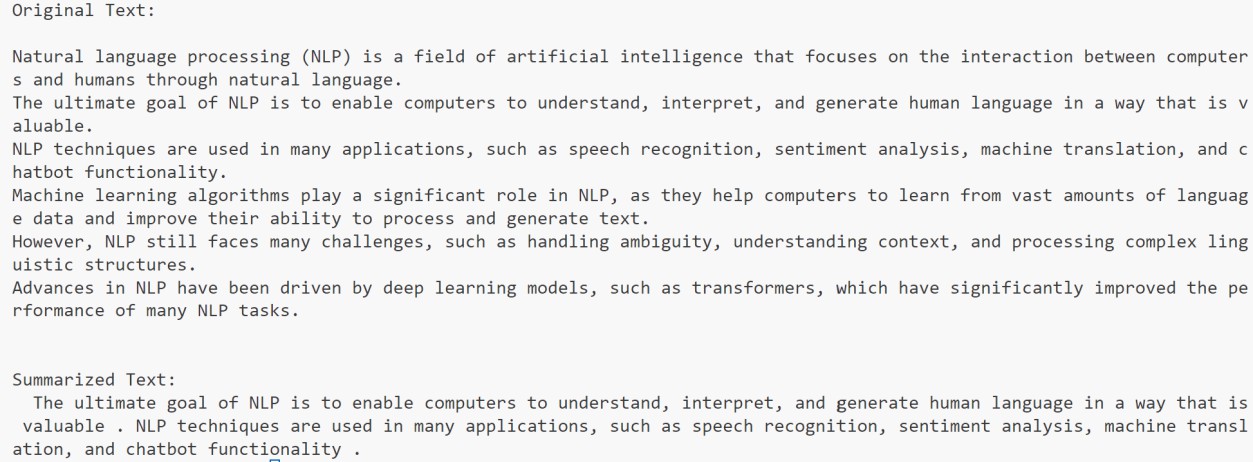
# Display the summarized text print("Original Text:\n", text)

print("\nSummarized Text:\n", summarized\_text)

**Explanation:**

This Python code utilizes the **Hugging Face Transformers library** to summarize a passage using a pre-trained text summarization model. The pipeline("summarization") initializes the summarizer, and the function summarize\_text processes the input text, generating a concise summary while preserving key ideas. Parameters like max\_length and min\_length ensure the summary stays within a specified length range. When provided with a detailed passage on natural language processing, the summarizer extracts its essence, highlighting the main points in fewer words. The original and summarized texts are then displayed for comparison.

**Output:**

****

1. **Install langchain, cohere (for key), langchain-community. Get the api key( By logging into Cohere and obtaining the cohere key). Load a text document from your google drive . Create a prompt template to display the output in a particular manner.**

**Code:**

from google.oauth2.service\_account import Credentials from googleapiclient.discovery import build

from langchain.prompts import PromptTemplate from langchain.llms import Cohere

from langchain.chains import LLMChain

# Set your Cohere API Key import cohere

# Function to load a text document from Google Drive def ld(ds, fid):

request = ds.files().get\_media(fileId=fid) file\_content = request.execute()

return file\_content.decode('utf-8')

# Google Drive File ID and Credentials fid = '1do91VkEgCECFFwnb0Eamo6cOyA6DyUWE'

creds = Credentials.from\_service\_account\_file( 'credentials.json', scopes=["https://[www.googleapis.com/auth/drive.readonly](http://www.googleapis.com/auth/drive.readonly)"]

)

ds = build('drive', 'v3', credentials=creds) document\_text = ld(ds, fid)

# Create prompt template prompt\_template = """{document\_text}"""

# Format the prompt with the loaded document content

formatted\_prompt = prompt\_template.format(document\_text=document\_text)

# Load Cohere for language model generation with low temperature for deterministic output

cohere\_model = Cohere(cohere\_api\_key="88t76YKbB5CMN6b0WBUgq1NUaN7KxqJu7Ci3K0W4", temperature=0.0)

# Set up LangChain LLM with Cohere llm\_chain = LLMChain(llm=cohere\_model,

prompt=PromptTemplate(template=formatted\_prompt))

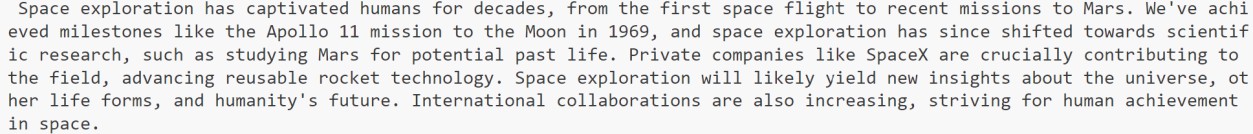
# Generate output using the LLMChain

output = llm\_chain.run(input\_document=document\_text) print(output)

**Explanation:**

The provided Python code loads a text document from Google Drive using the Google Drive API and processes it with a Cohere language model via LangChain for generating structured text output. It begins by authenticating with Google Drive using a service account and fetching the document's content using its file ID. The content is then formatted into a prompt template, which is designed to guide the Cohere language model in generating a response. The Cohere model is configured with a low temperature (temperature=0.0) to ensure deterministic and consistent output. The LangChain library connects the prompt and the Cohere model to create a language model chain (LLMChain), which processes the input document and generates a structured summary or response. Finally, the output is printed, providing a predictable and context-aware text generation based on the input document.

**Output:**

****

1. **Take the Institution name as input. Use Pydantic to define the schema for the desired output and create a custom output parser. Invoke the Chain and Fetch Results. Extract the below Institution related details from Wikipedia: The founder of the Institution. When it was founded. The current branches in the institution . How many employees are working in it. A brief 4-line summary of the institution.**

**Code:**

from pydantic import BaseModel import wikipediaapi

# Define the Pydantic Schema

class InstitutionDetails(BaseModel): name: str

founder: str founded: str branches: str employees: str summary: str

# Function to Fetch and Extract Details from Wikipedia def fetch(institution\_name):

user\_agent = "InstitutionInfoFetcher/1.0 (https://example.com; [contact@example.com)"](mailto:contact@example.com)

wiki = wikipediaapi.Wikipedia('en', headers={"User-Agent": user\_agent}) page = wiki.page(institution\_name)

content = page.text

# Basic parsing for demonstration purposes

founder = next((line for line in content.split('\n') if "founder" in line.lower()), "Not available")

founded = next((line for line in content.split('\n') if "founded" in line.lower() or "established" in line.lower()), "Not available")

branches = next((line for line in content.split('\n') if "branch" in line.lower()), "Not available")

employees = next((line for line in content.split('\n') if "employee" in line.lower()), "Not available")

summary = "\n".join(content.split('\n')[:4])

return InstitutionDetails( name=institution\_name, founder=founder, founded=founded, branches=branches, employees=employees, summary=summary

)

details = fetch("JNNCE") print("\nExtracted Institution Details:")

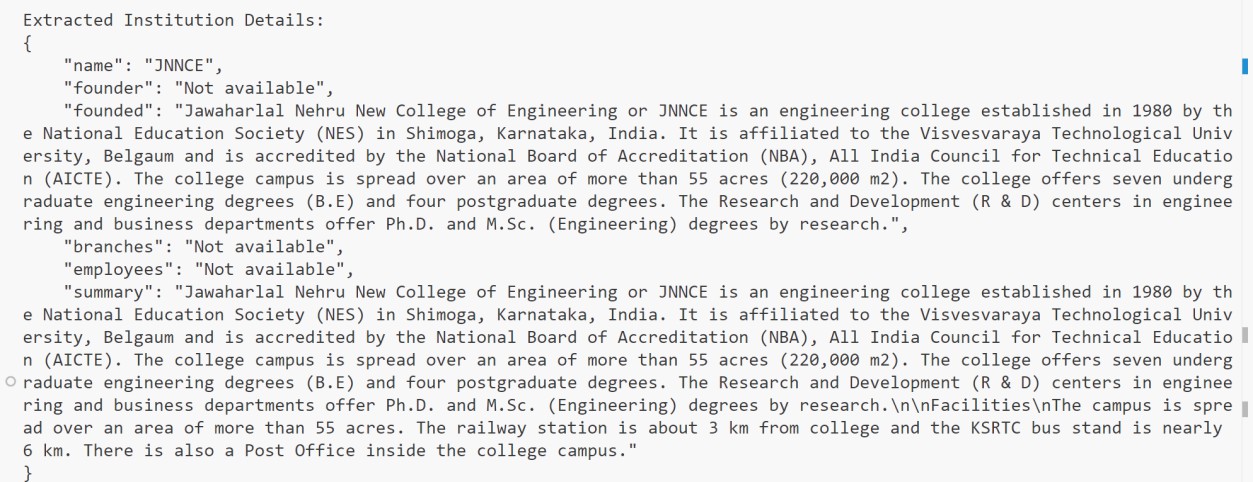
print(details.model\_dump\_json(indent=4)) # Use model\_dump\_json instead of

.json()

**Explanation:**

The code uses the wikipediaapi library to fetch information about an institution from Wikipedia and the pydantic library to structure the data. It defines a Pydantic model, InstitutionDetails, with fields such as name, founder, founded date, branches, employees, and a summary. The fetch function retrieves the Wikipedia page for a given institution name, parses its content, and extracts key details using simple string matching for terms like "founder," "founded," and "employee." A summary is generated from the first few lines of the content. Finally, the extracted data is returned as an InstitutionDetails instance and displayed in JSON format using the model\_dump\_json method. This approach ensures a structured representation of institution information.

**Output:**

****

1. **Build a chatbot for the Indian Penal Code. We'll start by downloading the official Indian Penal Code document, and then we'll create a chatbot that can interact with it. Users will be able to ask questions about the Indian Penal Code and have a conversation with it.**

**Code:**

import pdfplumber

# Step 1: Extract Text from IPC PDF def extract(file):

with pdfplumber.open(file) as pdf: text = ""

for page in pdf.pages:

text += page.extract\_text() return text

# Step 2: Search for Relevant Sections in IPC def search(query, ipc):

query = query.lower() lines = ipc.split("\n")

results = [line for line in lines if query in line.lower()] return results if results else ["No relevant section found."]

# Step 3: Main Chatbot Function def chatbot():

print("Loading IPC document...") ipc = extract("IPC.pdf")

while True:

query = input("Ask a question about the IPC: ") if query.lower() == "exit":

print("Goodbye!") break

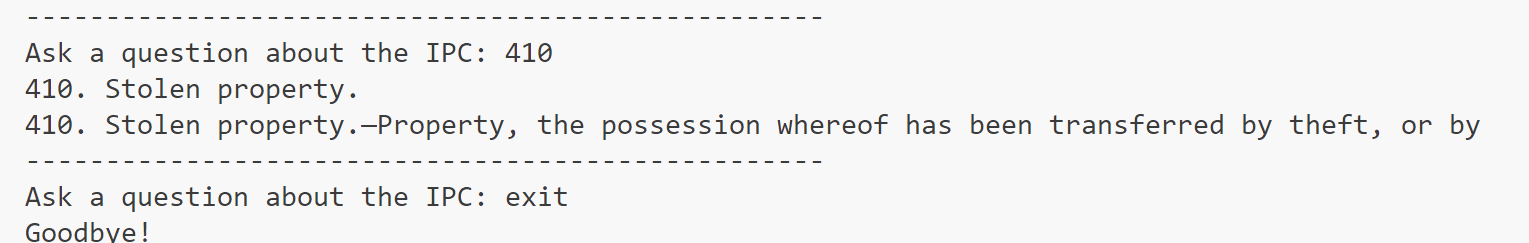
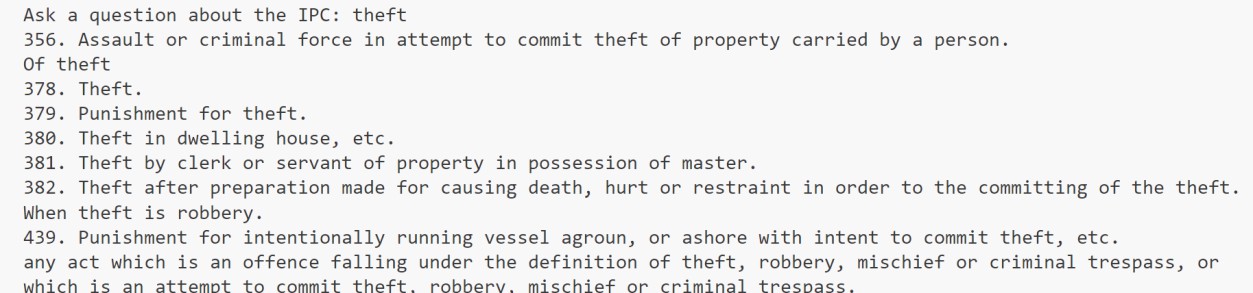
results = search(query, ipc) print("\n".join(results)) print("-" \* 50)

chatbot()

**Explanation:**

The code extracts text from an IPC (Indian Penal Code) PDF document using the pdfplumber library and allows users to query specific sections of the IPC. The extract function reads the text content from each page of the PDF, and the search function looks for matching lines in the document based on the user's query. It performs a case- insensitive search for relevant sections. The chatbot function provides an interactive interface where users can ask questions about the IPC, and it responds with the matching text from the document. The program continues until the user enters "exit", at which point it terminates.

**Output:**

****