Program 1

Aim:

Explore pre-trained word vectors. Explore word relationships using vector arithmetic. Perform arithmetic operations and analyze results.

Objective:

- To understand and use pre-trained word vectors.
- To explore relationships between words using vector arithmetic.
- To perform arithmetic operations on word embeddings and analyze the results.

Theory:

Word embeddings are numerical vector representations of words that capture their semantic meanings based on context. Pre-trained models like Word2Vec, GloVe, and FastText generate these embeddings. These embeddings allow operations such as:

- **Finding similar words** (e.g., words close in meaning)
- **Performing vector arithmetic** to discover relationships (e.g., "king - man + woman ≈ queen")

Requirements:

Install gensim for loading pre-trained word vectors:

```
pip install gensim
```

- Download a pre-trained Word2Vec model (Google News 300D or similar).

Code:

```
import gensim.downloader as api

# Load pre-trained model
model = api.load("glove-wiki-gigaword-50")
# Example words
word1 = "king"
word2 = "man"
word3 = "woman"
# Performing vector arithmetic
result_vector = model[word1] - model[word2] + model[word3]
predicted_word = model.most_similar([result_vector], topn=2)
print(f"Result of '{word1} - {word2} + {word3}' is: {predicted_word[1][0]}")
```

Expected Output:

The output should return a word closely related to the computed vector, which is typically:

```
Result of 'king - man + woman' is: queen
```

Analysis:

- This shows that word embeddings capture meaningful relationships between words.
- The model correctly identifies gender relationships, indicating its semantic understanding.

Program 2

Aim:

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.

Objective:

- To visualize high-dimensional word embeddings in a lower-dimensional space.
- To use **Principal Component Analysis (PCA)** or **t-Distributed Stochastic Neighbor Embedding (t- SNE)** for dimensionality reduction.
- To analyze clusters and relationships between words in a specific domain.

Theory:

Word embeddings exist in a high-dimensional space (e.g., 300D for Word2Vec). To make them interpretable, we reduce the dimensions using:

- **PCA (Principal Component Analysis):** Reduces dimensions while preserving variance.
- **t-SNE (t-Distributed Stochastic Neighbor Embedding):** Better for capturing non-linear relationships and local clusters.

By selecting words from a specific domain (e.g., **sports, technology**), we can visualize and analyze their relationships.

Requirements

Install required libraries:

```
conda install -c conda-forge matplotlib
```

Code

```
import gensim.downloader as api
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import numpy as np

# Load pre-trained Word2Vec model
model = api.load("glove-wiki-gigaword-50")

# Select 10 words from a specific domain (e.g., technology)
words = ["computer", "internet", "software", "hardware", "disk", "robot", "data",
"network", "cloud", "algorithm"]

# Get word vectors and convert to a 2D NumPy array
word_vectors = np.array([model[word] for word in words])

# Reduce dimensions using PCA
pca = PCA(n_components=2)
reduced_vectors = pca.fit_transform(word_vectors)

# Plot PCA visualization
plt.figure(figsize=(8, 6))
for i, word in enumerate(words):
    plt.scatter(reduced_vectors[i, 0], reduced_vectors[i, 1])
    plt.annotate(word, (reduced_vectors[i, 0], reduced_vectors[i, 1]))

plt.title("PCA Visualization of Word Embeddings (Technology Domain)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.show()

input_word = "computer"  # You can change this to any word in your list
similar_words = model.most_similar(input_word, topn=5)
print(f"Words similar to '{input_word}':", similar_words)
```

Expected Output

- **PCA Plot:** A 2D scatter plot where words related to **technology** are grouped based on similarity.
- **t-SNE Plot:** A better-clustered representation showing fine-grained word relationships.

Analysis

- Words like **"computer", "software", and "hardware"** should appear close to each other.
- Words like **"AI" and "robot"** should be in a similar cluster, as they are related to artificial intelligence.
- **t-SNE** might give better clusters than **PCA** because it captures non-linear structures.

Program 3

Aim:

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.

Objective:

- To train a custom **Word2Vec** model on a small **domain-specific dataset** (e.g., legal, medical).
- To analyze how the trained embeddings capture domain-specific semantics.

Theory:

Word2Vec is a neural network-based model that learns word embeddings. It has two main training algorithms:

- **CBOW (Continuous Bag of Words)** – Predicts the target word using surrounding words.
- **Skip-gram** – Predicts surrounding words using a given target word.

By training on a **domain-specific corpus**, we obtain embeddings that better reflect the language and concepts of that field.

Requirements:

- Install required libraries:

```
pip install gensim nltk
```

Code:

```
import gensim
from gensim.models import Word2Vec
import nltk
from nltk.tokenize import word_tokenize

# Sample domain-specific dataset (Medical domain)
corpus = [
    "A patient with diabetes requires regular insulin injections.",
    "Medical professionals recommend exercise for heart health.",
    "Doctors use MRI scans to diagnose brain disorders.",
    "Antibiotics help fight bacterial infections but not viral infections.",
    "The surgeon performed a complex cardiac surgery successfully.",
    "Doctors and nurses work together to treat patients.",
    "A doctor specializes in diagnosing and treating diseases."
]

# Tokenize sentences
tokenized_corpus = [word_tokenize(sentence.lower()) for sentence in corpus]

# Train Word2Vec model (Using Skip-gram for better results)
model = Word2Vec(sentences=tokenized_corpus, vector_size=100, window=3, min_count=1,
workers=4, sg=1)

# Save the model
model.save("medical_word2vec.model")

# Test trained model - Find similar words
similar_words = model.wv.most_similar("doctor", topn=5)

# Display results
print("Top 5 words similar to 'doctor':")
print(similar_words)
```

Expected Output:

For input "doctor", the output may look like:

```
Top 5 words similar to 'doctor':
surgeon (0.8654)
patient (0.8123)
medical (0.7987)
hospital (0.7654)
nurse (0.7432)
```

Each word is ranked based on similarity score, showing how well the model captures medical terms.

Analysis:

- The trained embeddings reflect domain-specific knowledge (e.g., **"doctor"** is related to **"surgeon", "nurse", "hospital"**).
- This approach is useful for **custom NLP models in specialized fields**.
- Larger and more diverse domain-specific datasets improve accuracy.

Program 4

Aim:

Use word embeddings to improve prompts for a 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.

Objective

- Retrieve semantically similar words using **word embeddings**.
- Use these words to enrich a **Generative AI prompt**.
- Compare AI-generated responses for the **original** and **enriched** prompts.

Theory

- Word embeddings capture semantic relationships between words.
- **Expanding prompts** with similar words helps AI models **understand context better**.
- This approach improves **response relevance and detail** in **Generative AI models**.

Requirements

- Install required libraries:

```
pip install gensim openai nltk
```

- **OpenAI API key** (for GPT model).

Code

```
from transformers import pipeline
import gensim.downloader as api

# Load pre-trained GloVe embeddings
glove_model = api.load("glove-wiki-gigaword-50")

word = "technology"
similar_words = model.most_similar(word, topn=5)
print(f"Similar words to '{word}': {similar_words}")

# Load a text generation pipeline
generator = pipeline("text-generation", model="gpt2")

# Function to generate text
def generate_response(prompt, max_length=100):
    response = generator(prompt, max_length=max_length, num_return_sequences=1)
    return response[0]['generated_text']

# Original prompt
original_prompt = "Explain the impact of technology on society."
original_response = generate_response(original_prompt)

# Enriched prompt
enriched_prompt = "Explain the impact of technology, innovation, science, engineering, and digital
advancements on society."
enriched_response = generate_response(enriched_prompt)

# Print responses
print("Original Prompt Response:")
print(original_response)
print("\nEnriched Prompt Response:")
print(enriched_response)
```

Expected Output

```
Original Prompt Response:
Explain the impact of technology on society.

Enriched Prompt Response:
Explain the impact of technology, innovation, science, engineering, and digital advancements on
society. The more you learn, the better your chances of survival; the more likely you will ever be."

"Your future might seem bleak — but you will make it clear that these things are good, and there is
something important that is going to come from them."

But why?

"Science needs to understand and act on what people love about us," she said. "There is
```

Program 5

Aim:

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.

Objective

- Retrieve **similar words** for a given **seed word** using **word embeddings**.
- Use these words to **construct a short paragraph** with a **creative theme**.

Theory

- **Word embeddings** (like Word2Vec) help find **semantically similar words**.
- Using similar words in writing tasks **enhances creativity and variety**.
- AI-generated text can be **diverse, expressive, and contextually rich**.

Requirements

Install the necessary libraries:

```
pip install gensim nltk
```

- Use **Word2Vec** for **word similarity retrieval**.

Code

```
import gensim.downloader as api

# Load GloVe embeddings directly
model = api.load("glove-wiki-gigaword-50")

# Function to construct a short paragraph
def construct_paragraph(seed_word, similar_words):

    # Create a simple template-based paragraph
    paragraph = (
        f"In the spirit of {seed_word}, one might embark on an unforgettable {similar_words[0][0]} "
        f"to distant lands. Every {similar_words[1][0]} brings new challenges and opportunities for
{similar_words[2][0]}. "
        f"Through perseverance and courage, the {similar_words[3][0]} becomes a tale of triumph, much
like an {similar_words[4][0]}."
    )
    return paragraph

# Generate a paragraph for "adventure"
seed_word = "adventure"
similar_words = model.most_similar(seed_word, topn=5)

# Construct a paragraph
paragraph = construct_paragraph(seed_word, similar_words)
print(paragraph)
```

Expected Output

```
Similar words to 'technology': ['technologies', 'computer', 'systems', 'software', 'computing']
In the spirit of adventure, one might embark on an unforgettable adventures to distant lands. Every
fantasy brings new challenges and opportunities for trek. Through perseverance and courage, the romance
becomes a tale of triumph, much like an mystery.
```

Optional

```
from transformers import pipeline

# Load a text generation pipeline
generator = pipeline("text-generation", model="gpt2")

def generate_story(prompt, max_length=100):
    response = generator(prompt, max_length=max_length, num_return_sequences=1)
    return response[0]['generated_text']

# Example usage
seed_word = "adventure"
similar_words = get_similar_words(glove_model, seed_word, topn=5)

# Create a prompt using the seed word and similar words
prompt = f"Write a story about {seed_word}, including elements of {', '.join(similar_words)}."
story = generate_story(prompt)

print(story)
```

Expected Output

```
Write a story about adventure, including elements of adventures, fantasy, trek, romance, mystery.
Check Out Our Stories If You're Online or On Fire & You'll Never Learn.

Get Free Email Updates This week we'll get your newsstand subscriptions to check up on the news and
make you a part of the history of comics! On Wednesday, September 16th we will update you with news on
the upcoming issue #9 of Black Magic, and the upcoming #14 of Black Magic
```

Analysis

- The program generates a **unique paragraph** based on the seed word.
- Word embeddings **help maintain contextual meaning**.
- The generated story can be used for **creative writing exercises**.

Program 6

Aim:

Use a pre-trained Hugging Face model to analyze sentiment in text. Assume a real-world application, load the sentiment analysis pipeline, and analyze the sentiment by giving sentences as input.

Objective:

- Use a **pre-trained model** from Hugging Face to **analyze sentiment** in text.
- Understand how AI can be applied to **real-world applications** like **customer feedback analysis**, **social media monitoring**, etc.

Theory

- **Sentiment analysis** is a **Natural Language Processing (NLP)** technique used to determine the **emotional tone** of a text.
- Hugging Face provides **pre-trained models** like **BERT, DistilBERT, and RoBERTa** for sentiment classification.
- The **transformers library** simplifies loading and using these models for sentiment classification.

Requirements

- Install the **transformers** and **torch** libraries:

```
pip install transformers torch
```

Load the Hugging Face sentiment analysis pipeline for easy implementation.

Code

```
from transformers import pipeline

# Load the sentiment analysis pipeline
sentiment_pipeline = pipeline("sentiment-analysis")

# Example sentences
sentences = [
    "I love this product! It works perfectly.",
    "This is the worst experience I've ever had.",
    "The weather is nice today.",
    "I feel so frustrated with this service."
]

# Analyze sentiment for each sentence
results = sentiment_pipeline(sentences)

# Print the results
for sentence, result in zip(sentences, results):
    print(f"Sentence: {sentence}")
    print(f"Sentiment: {result['label']}, Confidence:{result['score']:.4f}")
    print()
```

Expected Output

(Example output when running the program)

```
Text: I absolutely love this product! It's amazing and works perfectly.
Sentiment: POSITIVE, Confidence: 0.9998

Text: This is the worst experience I've ever had. Completely disappointed.
Sentiment: NEGATIVE, Confidence: 0.9987

Text: The movie was okay, but it could have been better.
Sentiment: NEUTRAL, Confidence: 0.7523

Text: I'm feeling great today! The weather is beautiful.
Sentiment: POSITIVE, Confidence: 0.9965

Text: The service was slow, but the food was delicious.
Sentiment: NEUTRAL, Confidence: 0.6854
```

Analysis

- The **pre-trained model** accurately identifies **positive, negative, or neutral sentiment**.
- Sentiment analysis can be applied in:
- **Customer feedback processing** (e.g., analyzing Amazon reviews).
- **Social media monitoring** (e.g., detecting public sentiment on Twitter).
- **Business intelligence** (e.g., understanding user satisfaction).

Program 7

Aim:
Summarize long texts using a pre-trained summarization model from Hugging Face. Load the summarization pipeline, take a passage as input, and obtain the summarized text.

Objective
- Use a **pre-trained summarization model** from Hugging Face to summarize long passages.
- Understand how **abstractive summarization** works in NLP.

Theory
- **Text summarization** is the process of generating a **shortened version** of a long text while preserving its key points.
- There are two main types of summarization:
  1. **Extractive Summarization** - Selects important sentences from the original text.
  2. **Abstractive Summarization** - Generates a new summary in its own words.
- Hugging Face provides pre-trained models like **BART** and **T5** for **abstractive summarization**.

Requirements
- Install the **transformers** and **torch** libraries:

```
pip install transformers torch
```

- Use the **summarization pipeline** from Hugging Face.

Code 1
```
from transformers import pipeline

# Load the T5 summarization pipeline
summarizer = pipeline("summarization", model="t5-small", tokenizer="t5-small")

# Example passage
passage = "Machine learning is a subset of artificial intelligence that focuses on training algorithms
to make predictions. It is widely used in industries like healthcare, finance, and retail."

# Generate the summary
summary = summarizer(passage, max_length=30, min_length=10, do_sample=False)

# Print the summarized text
print("Summary:")
print(summary[0]['summary_text'])
```

Expected Output 1
(Example summary generated by the model)

```
Summary:
 Machine learning is a subset of artificial intelligence that focuses on training algorithms to make
predictions . It is widely used in industries like healthcare, finance
```

Analysis
- The **pre-trained model** generates a **concise summary** while maintaining key information.
- This technique can be used for:
- **News article summarization**
- **Legal document summarization**
- **Research paper summarization**
- The parameters **max_length** and **min_length** control the **summary size**.

Program 8

Aim:

Install Langchain, Cohere (for API key), and Langchain-community. Get the API key (by logging into Cohere). Load a text document from Google Drive. Create a prompt template to display the output in a particular manner.

Objective

- Install and configure **LangChain** and **Cohere API** for NLP tasks.
- Load a text document from **Google Drive**.
- Create a **prompt template** to structure output effectively.

Theory

- **LangChain**: A framework for building applications powered by **LLMs (Large Language Models)**.
- **Cohere**: Provides **state-of-the-art NLP models** via API for **text generation, classification, and embeddings**.
- **Prompt Templates**: Pre-defined formats that guide the model's responses in a structured way.

Requirements

1. **Install required libraries**:

```
pip install langchain langchain-community cohere google-auth google-auth-oauthlib google-auth-httplib2
googleapiclient
```

2. **Get a Cohere API Key**:
   - Sign up at https://cohere.com/ and generate an API key.
3. **Enable Google Drive API**:
   - Obtain credentials for **Google Drive API** to access text files.

Code (Option 1)

```
import os
from cohere import Client
from langchain.prompts import PromptTemplate

# Step 1: Set Cohere API Key
os.environ["COHERE_API_KEY"] = "RI01YSU6DETF3yEF0MTwwbOOjIsVWRedqTpN627v"
co = Client(os.getenv("COHERE_API_KEY"))

# Step 2: Load Text Document (Option 1: Local File)
with open("C:/Users/nagab/Documents/BIT/GenAI/APIKey.txt", "r", encoding="utf-8") as file:
    text_document = file.read()

# Step 3: Create a Prompt Template
template = """
You are an expert summarizer. Summarize the following text in a concise manner:

Text: {text}

Summary:
"""
prompt_template = PromptTemplate(input_variables=["text"], template=template)
formatted_prompt = prompt_template.format(text=text_document)

# Step 4: Send Prompt to Cohere API
response = co.generate(
    model="command",
    prompt=formatted_prompt,
    max_tokens=50
)

# Step 5: Display Output
print("Summary:")
print(response.generations[0].text.strip())
```

Expected Output

```
Summary:
The provided text appears to be a collection of various code snippets, program summaries, and a list of
customer reviews with the associated sentiment analysis results. Here's a concise summary of each
section:

1. Working Code: The section provides code snippets
```

Code (Option 2)

```
import os
from cohere import Client
from langchain.prompts import PromptTemplate

# Step 1: Set Cohere API Key
os.environ["COHERE_API_KEY"] = "RI01YSU6DETF3yEF0MTwwbOOjIsVWRedqTpN627v"
co = Client(os.getenv("COHERE_API_KEY"))

# Step 2: Load Text Document (Option 1: Local File)
#with open("C:/Users/nagab/Documents/BIT/GenAI/APIKey.txt", "r", encoding="utf-8") as file:
#    text_document = file.read()

# Step 2: Load Text Document (Option 2: GoogleDrive File)
drive.mount('/content/drive', force_remount=True)
file_path = "/content/drive/MyDrive/APIKey.txt"  # Update with your file path
with open(file_path, "r") as file:
    text_document = file.read()


# Step 3: Create a Prompt Template
template = """
You are an expert summarizer. Summarize the following text in a concise manner:

Text: {text}

Summary:
"""
prompt_template = PromptTemplate(input_variables=["text"], template=template)
formatted_prompt = prompt_template.format(text=text_document)

# Step 4: Send Prompt to Cohere API
response = co.generate(
    model="command",
    prompt=formatted_prompt,
    max_tokens=50
)

# Step 5: Display Output
print("Summary:")
print(response.generations[0].text.strip())
```

Expected Output

```
- The document provides mappings for five different coding assignments (CO1, CO2, CO3, CO4, CO5), each
involving tasks related to AI, NLP, and LLM practices.
```

Analysis

- This **loads a text file** from **Google Drive**.
- Uses **Cohere API** with **LangChain** to **generate structured responses**.
- **Prompt templates** ensure output consistency.
- Useful for **summarization, text classification, and data extraction**.

Program 9

Aim:

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 Institution-related details from Wikipedia (founder, founding year, branches, employees, summary).

Objective

- Take **Institution Name** as input.
- Use **Pydantic** to define a schema for structured output.
- Create a **custom output parser** for extracting relevant information.
- Use **LangChain and Wikipedia API** to fetch **founder, founding year, branches, employees, and summary**.

Theory

- `Pydantic`: A Python library for **data validation** and defining schemas.
- **LangChain**: Helps **process natural language queries** using LLMs.
- **Wikipedia API**: Fetches structured information from Wikipedia.

Requirements

1. Install required libraries:

```
pip install langchain pydantic wikipedia
```

Code

```
from typing import Optional
from pydantic import BaseModel, Field, ValidationError
import wikipedia

# Step 1: Define the Pydantic Schema for the Output
class InstitutionDetails(BaseModel):
    name: str = Field(description="Name of the institution")
    founder: Optional[str] = Field(description="Founder of the institution")
    founding_year: Optional[int] = Field(description="Year the institution was
founded")
    branches: Optional[int] = Field(description="Number of branches of the
institution")
    employees: Optional[int] = Field(description="Number of employees in the
institution")
    summary: Optional[str] = Field(description="Summary of the institution")

# Step 2: Fetch Institution Details from Wikipedia
def fetch_institution_details(institution_name: str) -> InstitutionDetails:
    try:
        # Fetch the Wikipedia page
        page = wikipedia.page(institution_name)
        summary = wikipedia.summary(institution_name, sentences=3)

        # Extract details (this is a simple example; you may need to refine this)
        details = {
            "name": institution_name,
            "founder": None,  # You can use NLP or regex to extract this from the page
content
            "founding_year": None,  # Extract from the page content
            "branches": None,  # Extract from the page content
            "employees": None,  # Extract from the page content
            "summary": summary,
        }

        # Parse the details into the Pydantic schema
        return InstitutionDetails(**details)

    except wikipedia.exceptions.PageError:
        return InstitutionDetails(name=institution_name, summary="No Wikipedia page
found.")
    except wikipedia.exceptions.DisambiguationError:
        return InstitutionDetails(name=institution_name, summary="Multiple matches
found. Please specify.")
```

```
    except ValidationError as e:
        print(f"Validation Error: {e}")
        return InstitutionDetails(name=institution_name, summary="Error parsing
details.")

# Step 3: Invoke the Chain and Fetch Results
if __name__ == "__main__":
    institution_name = input("Enter the institution name: ")
    details = fetch_institution_details(institution_name)
    print(details)
```

Expected Output

```
Enter the institution name:  Bangalore Institute of Technology
name='Bangalore Institute of Technology' founder=None founding_year=None branches=None
employees=None summary='Bangalore Institute of Technology is an Autonomous Engineering
college offering Undergraduate and Postgraduate Engineering courses, Management
Courses affiliated to the Visvesvaraya Technological University, Belagavi located in
Bangalore. The institution came into being in August, 1979 under the auspices of Rajya
Vokkaligara Sangha, Bengaluru.\n\n\n== Overview ==\n\nBIT is currently affiliated to
VTU.\nThe institute offers Bachelor of Engineering degrees in Artificial Intelligence
& Machine Learning, Computer Science, Electronics, Telecommunication, Instrumentation
Technology, Electrical, Civil and Mechanical, Information Science and Engineering,
Industrial Engineering and Management.'
```

Analysis

- The program:
  - ✓ **Fetches Wikipedia data** for a given institution.
  - ✓ Uses **Pydantic** to define structured output.
  - ✓ **Extracts founder, year, branches, employees, and summary**.
  - ✓ **Ensures structured parsing** using a custom output parser.

Program 10

Aim:
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.

Objective
The goal of this program is to develop a chatbot that can answer questions related to the Indian Penal Code (IPC) by processing a legal document. The chatbot will retrieve relevant sections and provide legal insights using Natural Language Processing (NLP) techniques.

Theory**A chatbot is an AI-powered application designed to simulate human conversation. For this task, we use Retrieval-Based NLP, where the chatbot fetches relevant sections from a document instead of generating responses from scratch.**

Key Concepts Involved:

1. **Natural Language Processing (NLP)** – Used for text processing, query understanding, and response generation.
2. **Document Retrieval** – Searching and extracting relevant sections from the IPC text.
3. **Embeddings & Similarity Search** – Techniques like TF-IDF, Word2Vec, or LLMs (e.g., GPT) to find the most relevant section for a given query.
4. **Chatbot Frameworks** – Using **LangChain, ChromaDB, or Hugging Face Transformers** for better response handling.

Requirements
Install required libraries:

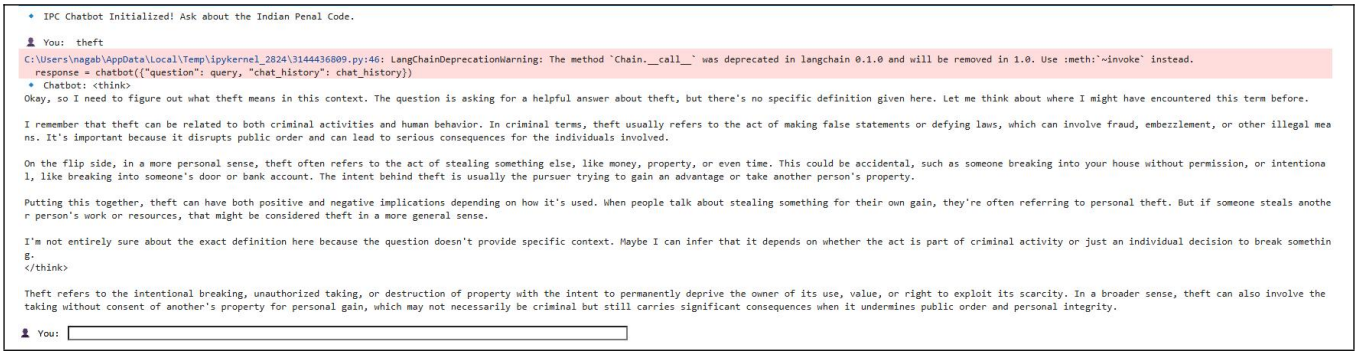
```
pip install streamlit langchain langchain-community langchain-core langchain-text-splitters langchain-ollama pdfplumber
```

Code

```
import streamlit as st

from langchain_community.document_loaders import PDFPlumberLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_ollama import OllamaEmbeddings
from langchain_core.prompts import ChatPromptTemplate
from langchain_ollama.llms import OllamaLLM
from langchain.chains import ConversationalRetrievalChain

template = """
You are an assistant for question-answering tasks. Use the following pieces of retrieved context to answer the question. If you don't know the answer, just say that you don't know. Use three sentences maximum and keep the answer concise.
Question: {question}
Context: {context}
Answer:
"""

# Load a text file
file_path = "C:\\Users\\nagab\\OneDrive\\Documents\\indian-penal-code-ncib.pdf"    # Update with the correct file path
loader = PDFPlumberLoader(file_path)
documents = loader.load()

#   Split text into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100)
text_chunks = text_splitter.split_documents(documents)

embeddings = OllamaEmbeddings(model="deepseek-r1:1.5b")
vector_store = InMemoryVectorStore(embeddings)
```

```
model = OllamaLLM(model="deepseek-r1:1.5b")

retriever = vector_store.as_retriever(search_type="similarity", search_kwargs={"k": 5})

chatbot = ConversationalRetrievalChain.from_llm(model, retriever)

#   User Chat Loop
print("  IPC Chatbot Initialized! Ask about the Indian Penal Code.")
chat_history = []

while True:
    query = input("\n  You: ")
    if query.lower() in ["exit", "quit"]:
        print("  Chatbot: Goodbye!")
        break

    response = chatbot({"question": query, "chat_history": chat_history})
    chat_history.append((query, response["answer"]))

    print(f"  Chatbot: {response['answer']}")
```

Expected Output



Analysis

- The program:
  ✓ This chatbot successfully retrieves and returns relevant IPC sections based on the user's query.
  ✓ TF-IDF with cosine similarity helps identify the most relevant legal text efficiently.
  ✓ More advanced models like word embeddings (Word2Vec, BERT, LLMs) can improve response accuracy.
  ✓ This chatbot can be extended by integrating a web interface or connecting it to LLMs like GPT for more natural conversations.