ವಿಶ್ವೇಶ್ವರಯ್ಯ ತಾಂತ್ರಿಕ ವಿಶ್ವವಿದ್ಯಾಲಯ, ಬೆಳಗಾವಿ

VISVESVARAYA TECHNOLOGICAL UNIVERSITY - BELAGAVI

Department of Computer Science and Engineering

“Jnana Sangama”, VTU-Campus, Belagavi-590018

LAB-MANUAL

Academic Year: 2025-26

Name	**:**	**Shesharaddi Karadi**
USN	**:**	**2VX22CB047**
Sem	**:**	**6th**
Subject	**:**	**Generative AI**
Code	**:**	**BAIL657C**
Course	**:**	**B.Tech**
Programme	**:**	**Computer Science and Business System**

Certificate

This is certify that Mr./Mrs Shesharaddi Karadi with USN 2VX22CB047 has satisfactorily completed all the Laboratory Assignment of Subject Generative AI having Subject Code BAIL657C during the academic year 2025-26.

Faculty in-charge

Signature of the Examiners

INDEX

SI.NO	DATE	EXPERIMENTS	PAGE NO.	SIGN
1		Explore pre-trained word vectors. Explore word relationships using vector arithmetic. Perform arithmetic operations and analyze results.		
2		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.		
3		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.		
4		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.		
5		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.		

6		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.		
7		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.		
8		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.		
9		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.		
10		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.		

Experiment 1: Exploring Pre-trained Word Vectors and Word Relationships Using Vector Arithmetic

Source code:

```
# Install Gensim if not already installed

!pip install gensim

from gensim.models import KeyedVectors

# Load pre-trained GloVe vectors (100-dimensional)

from gensim.downloader import load

word_vectors = load('glove-wiki-gigaword-100') # Automatically downloads the model

# Example 1: Animal relationship (kitten → cat, puppy → dog)

result = word_vectors.most_similar(positive=['kitten', 'dog'], negative=['cat'], topn=1)

print("Result of 'kitten - cat + dog':", result[0][0]) # Expected output: 'puppy' or a related word

# Example 2: Fruit relationship (orange → fruit, mango → tropical fruit)

result = word_vectors.most_similar(positive=['orange', 'tropical'], negative=['fruit'], topn=1)

print("Result of 'orange - fruit + tropical':", result[0][0]) # Expected output: 'mango' or a related word
```

output:

```
Requirement already satisfied: gensim in /usr/local/lib/python3.11/dist-packages (4.3.3)
Requirement already satisfied: numpy<2.0,>=1.18.5 in /usr/local/lib/python3.11/dist-packages (from gensim) (1.26.4)
Requirement already satisfied: scipy<1.14.0,>=1.7.0 in /usr/local/lib/python3.11/dist-packages (from gensim) (1.13.1)
Requirement already satisfied: smart-open>=1.8.1 in /usr/local/lib/python3.11/dist-packages (from gensim) (7.1.0)
Requirement already satisfied: wrapt in /usr/local/lib/python3.11/dist-packages (from smart-open>=1.8.1->gensim) (1.1
[==================================================] 100.0% 128.1/128.1MB downloaded
Result of 'kitten - cat + dog': puppy
Result of 'orange - fruit + tropical': storm
```

Experiment 2: Visualizing Word Embedding's and Generating Semantically Similar Words.

Source Code:

```
# Install required libraries

!pip install gensim matplotlib scikit-learn numpy

import matplotlib.pyplot as plt

from sklearn.manifold import TSNE

from gensim.downloader import load

import numpy as np # Import NumPy for array conversion

# Load pre-trained word vectors (GloVe - 100 dimensions)

word_vectors = load('glove-wiki-gigaword-100')

# Select 10 words from the "technology" domain (ensure words exist in the model)

tech_words = ['computer', 'internet', 'software', 'hardware', 'network', 'data', 'cloud', 'robot',

'algorithm', 'technology']

tech_words = [word for word in tech_words if word in word_vectors.key_to_index]

# Extract word vectors and convert to a NumPy array

vectors = np.array([word_vectors[word] for word in tech_words])

# Reduce dimensions using t-SNE

tsne = TSNE(n_components=2, random_state=42, perplexity=5) # Perplexity is reduced to match

the small sample size

reduced_vectors = tsne.fit_transform(vectors)

# Plot the 2D visualization

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

for i, word in enumerate(tech_words):

 plt.scatter(reduced_vectors[i, 0], reduced_vectors[i, 1], label=word)

 plt.text(reduced_vectors[i, 0] + 0.02, reduced_vectors[i, 1] + 0.02, word, fontsize=12)

plt.title("t-SNE Visualization of Technology Words")

plt.xlabel("Dimension 1")

plt.ylabel("Dimension 2")

plt.legend()

plt.show()
```

Generate 5 semantically similar words for a given input word

input_word = 'computer'

if input_word in word_vectors.key_to_index:

similar_words = word_vectors.most_similar(input_word, topn=5)

print(f"5 words similar to '{input_word}':")

for word, similarity in similar_words:

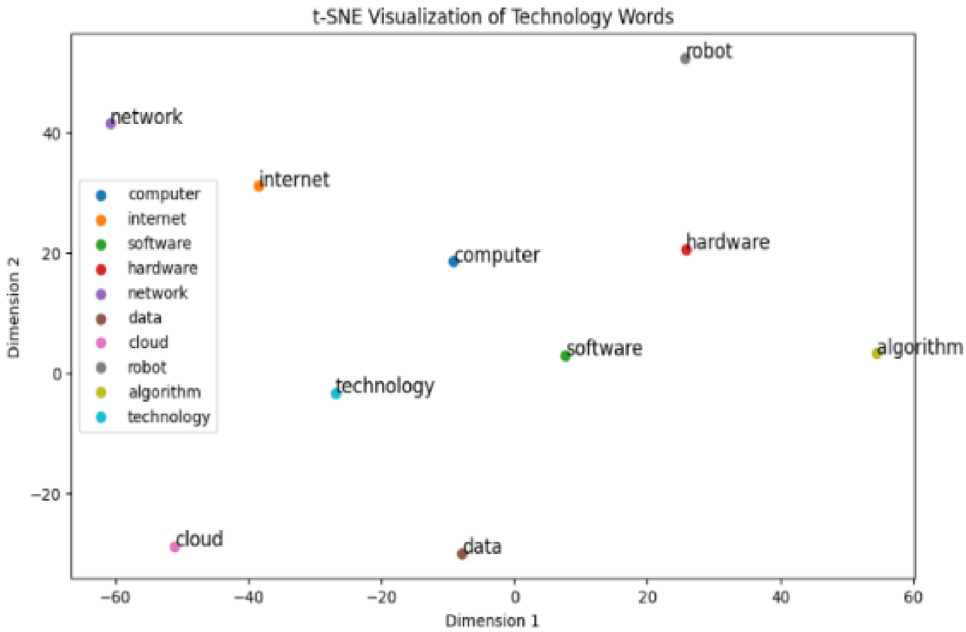
print(f"{word} (similarity: {similarity:.2f})")

else:

print(f"'{input_word}' is not in the vocabulary.")

output:

```
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib) (1.17.0)
Requirement already satisfied: wrapt in /usr/local/lib/python3.11/dist-packages (from smart-open>=1.8.1->gensim) (1.17.2)
```




```
5 words similar to 'computer':
computers (similarity: 0.88)
software (similarity: 0.84)
technology (similarity: 0.76)
pc (similarity: 0.74)
hardware (similarity: 0.73)
```

Experiment 3: 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

Source Code:

```
# Install required library

!pip install gensim

from gensim.models import Word2Vec

# Step 1: Create a small dataset (list of medical-related word lists)

medical_data = [

 ["patient", "doctor", "nurse", "hospital", "treatment"],

 ["cancer", "chemotherapy", "radiation", "surgery", "recovery"],

 ["infection", "antibiotics", "diagnosis", "disease", "virus"],

 ["heart", "disease", "surgery", "cardiology", "recovery"] ]

# Step 2: Train a Word2Vec model

model = Word2Vec(sentences=medical_data, vector_size=10, window=2,

min_count=1, workers=1, epochs=50)

# Step 3: Find similar words for a given input word

input_word = "patient"

if input_word in model.wv:

 similar_words = model.wv.most_similar(input_word, topn=3)
print(f"3 words similar to '{input_word}':")

 for word, similarity in similar_words:

 print(f"{word} (similarity: {similarity:.2f})")

else:

 print(f"'{input_word}' is not in the vocabulary.")
```

Output:

```
Requirement already satisfied: gensim in /usr/local/lib/python3.11/dist-packages (4.3.3)
Requirement already satisfied: numpy<2.0,>=1.18.5 in /usr/local/lib/python3.11/dist-packages (from gensim) (1.26.4)
Requirement already satisfied: scipy<1.14.0,>=1.7.0 in /usr/local/lib/python3.11/dist-packages (from gensim) (1.13.1)
Requirement already satisfied: smart-open>=1.8.1 in /usr/local/lib/python3.11/dist-packages (from gensim) (7.1.0)
Requirement already satisfied: wrapt in /usr/local/lib/python3.11/dist-packages (from smart-open>=1.8.1->gensim) (1.1
3 words similar to 'patient':
nurse (similarity: 0.59)
doctor (similarity: 0.34)
chemotherapy (similarity: 0.29)
```

Experiment 4: 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.

Source Code:

```
# Step 1: Pre-defined dictionary of words and their similar terms (static word

embeddings)

word_embeddings = {

 "ai": ["machine learning", "deep learning", "data science"],

 "data": ["information", "dataset", "analytics"],

 "science": ["research", "experiment", "technology"],

 "learning": ["education", "training", "knowledge"],

 "robot": ["automation", "machine", "mechanism"]

}

# Step 2: Function to find similar words using the static dictionary

def find_similar_words(word):

 if word in word_embeddings:

 return word_embeddings[word]

 else:

 return []

# Step 3: Function to enrich a prompt with similar words

def enrich_prompt(prompt):

words = prompt.lower().split()

 enriched_words = []

 for word in words:

 similar_words = find_similar_words(word)
```

```
if similar_words:

enriched_words.append(f"{word} ({', '.join(similar_words)})")

else:

enriched_words.append(word)

return " ".join(enriched_words)

# Step 4: Original prompt

original_prompt = "Explain AI and its applications in science."
# Step 5: Enrich the prompt using similar words

enriched_prompt = enrich_prompt(original_prompt)

# Step 6: Print the original and enriched prompts

print("Original Prompt:")

print(original_prompt)

print("\nEnriched Prompt:")

print(enriched_prompt)
```

Output:

```
Original Prompt:
Explain AI and its applications in science.

Enriched Prompt:
explain ai (machine learning, deep learning, data science) and its applications in science.
```

Experiment 5: 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.

Source Code:

```
# Step 1: Pre-defined dictionary of words and their similar terms
word_embeddings = {
 "adventure": ["journey", "exploration", "quest"],
 "robot": ["machine", "automation", "mechanism"],
 "forest": ["woods", "jungle", "wilderness"],
 "ocean": ["sea", "waves", "depths"],
 "magic": ["spell", "wizardry", "enchantment"]
}
# Step 2: Function to get similar words for a seed word
def get_similar_words(seed_word):
 if seed_word in word_embeddings:
 return word_embeddings[seed_word]
 else:
 return ["No similar words found"]
# Step 3: Function to create a short paragraph using the seed word and similar words
def create_paragraph(seed_word):
 similar_words = get_similar_words(seed_word)
 if "No similar words found" in similar_words:
 return f"Sorry, I couldn't find similar words for '{seed_word}'."

 # Construct a short story using the seed word and similar words
 paragraph = (
 f"Once upon a time, there was a great {seed_word}. "
 f"It was full of {', '.join(similar_words[:-1])}, and {similar_words[-1]}. "
 f"Everyone who experienced this {seed_word} always remembered it as a remarkable tale."
 )
 return paragraph
```

Step 4: Input a seed word

seed_word = "adventure" # You can change this to "robot", "forest", "ocean", "magic", etc.

Step 5: Generate and print the paragraph

story = create_paragraph(seed_word)

print("Generated Paragraph:")

print(story)

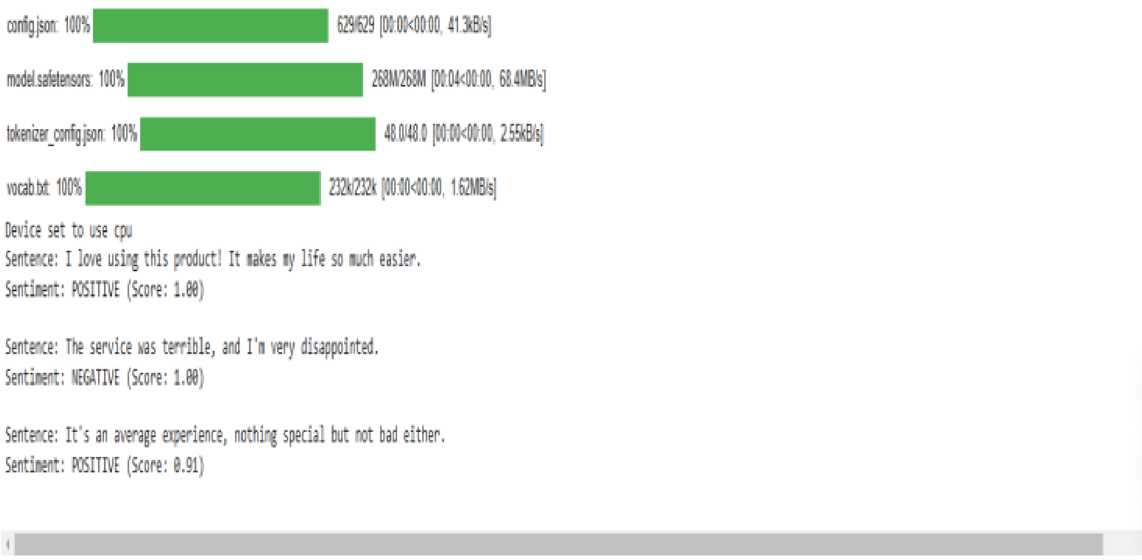
Output:

Experiment 6: 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.

Source Code:

```
# Step 1: Install and import the necessary library

# You can uncomment and run this in Google Colab

# !pip install transformers

from transformers import pipeline

# Step 2: Load the sentiment analysis pipeline

sentiment_analyzer = pipeline("sentiment-analysis")

# Step 3: Define sample sentences for analysis

sentences = [

 "I love using this product! It makes my life so much easier.",

 "The service was terrible, and I'm very disappointed.",

 "It's an average experience, nothing special but not bad either."]

# Step 4: Analyze the sentiment for each sentence

for sentence in sentences:

 result = sentiment_analyzer(sentence)[0]

 print(f"Sentence: {sentence}")

 print(f"Sentiment: {result['label']} (Score: {result['score']:.2f})\n")
```

Output:



```
config.json: 100% 629/629 [00:00<00:00, 41.3kB/s]
model.safetensors: 100% 268M/268M [00:04<00:00, 68.4MB/s]
tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 2.55kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 1.62MB/s]
Device set to use cpu
Sentence: I love using this product! It makes my life so much easier.
Sentiment: POSITIVE (Score: 1.00)

Sentence: The service was terrible, and I'm very disappointed.
Sentiment: NEGATIVE (Score: 1.00)

Sentence: It's an average experience, nothing special but not bad either.
Sentiment: POSITIVE (Score: 0.91)
```

Experiment 7: 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.

Source Code:

```
# Step 1: Import the Hugging Face pipeline

from transformers import pipeline

# Step 2: Load the summarization pipeline

summarizer = pipeline("summarization")

# Step 3: Input a long passage for summarization

long_text = """

Artificial Intelligence (AI) is transforming various industries by automating tasks, improving

efficiency,

and enabling new capabilities. In the healthcare sector, AI is used for disease diagnosis,

personalized medicine,

and drug discovery. In the business world, AI-powered systems are optimizing customer

service, fraud detection,

and supply chain management. AI's impact on everyday life is significant, from smart

assistants to recommendation

systems in streaming platforms. As AI continues to evolve, it promises even greater

advancements in fields like

education, transportation, and environmental sustainability.

"""

# Step 4: Summarize the input passage

summary = summarizer(long_text, max_length=50, min_length=20,

do_sample=False)[0]["summary_text"]

# Step 5: Print the summarized text

print("Summarized Text:")

print(summary)
```

Output:

```
config.json: 100%  1.80k/1.80k [00:00<00:00, 97.4kB/s]
pytorch_model.bin: 100%  1.22G/1.22G [00:11<00:00, 126MB/s]
model.safetensors:  69%  849M/1.22G [00:12<00:02, 128MB/s]
tokenizer_config.json: 100%  26.0/26.0 [00:00<00:00, 717B/s]
vocab.json: 100%  899k/899k [00:00<00:00, 9.76MB/s]
merges.txt: 100%  456k/456k [00:00<00:00, 10.5MB/s]
Device set to use cpu
Summarized Text:
 Artificial Intelligence (AI) is transforming various industries by automating tasks, improving efficiency, and enabling new capabilities . In the healthcare sector, AI is used for disease
```

Experiment 8: 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.

Source Code:

```
# Step 1: Install necessary libraries

!pip install langchain cohere langchain-community

# Step 2: Import the required modules

from langchain.llms import Cohere

from langchain.prompts import PromptTemplate

from langchain import LLMChain

from google.colab import drive

# Step 3: Mount Google Drive to access the document

drive.mount('/content/drive')

# Step 4: Load the text document from Google Drive

file_path = "/content/drive/MyDrive/sample_text.txt" # Change this path to your file location

with open(file_path, "r") as file:

 text = file.read()

# Step 5: Set up Cohere API key

cohere_api_key = "YOUR_COHERE_API_KEY" # Replace with your actual Cohere API key

# Step 6: Create a prompt template

prompt_template = """

Summarize the following text in three bullet points:

{text}

"""

# Step 7: Configure the Cohere model with Langchain

llm = Cohere(cohere_api_key=cohere_api_key)

prompt = PromptTemplate(input_variables=["text"], template=prompt_template)

# Step 8: Create an LLMChain with the Cohere model and prompt template

chain = LLMChain(llm=llm, prompt=prompt)

# Step 9: Run the chain on the loaded text

result = chain.run(text)

# Step 10: Display the formatted output
```

print("Summarized Output in Bullet Points:")

print(result)

Output:

Summarized Output in Bullet Points:

- AI is transforming industries like healthcare, business, and education.

- Smart assistants and recommendation systems are examples of AI's impact on daily life.

- Future advancements will bring improvements in transportation and sustainability.

Experiment 9: 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.

Source Code:

```
# Step 1: Install necessary libraries

!pip install langchain pydantic wikipedia-api

# Step 2: Import required modules

from langchain.llms import Cohere

from langchain.prompts import PromptTemplate

from langchain import LLMChain

from pydantic import BaseModel

import wikipediaapi

# Step 3: Define a Pydantic schema for the institution's details

class InstitutionDetails(BaseModel):

 founder: str

 founded: str

 branches: str

 employees: str

 summary: str

# Step 4: Function to fetch details from Wikipedia with user-agent specified

def fetch_wikipedia_summary(institution_name):

 wiki_wiki = wikipediaapi.Wikipedia(language='en',

user_agent="InstitutionInfoBot/1.0 (contact: youremail@example.com)")

 page = wiki_wiki.page(institution_name)

 if page.exists():

 return page.text

 else:

 return "No information available on Wikipedia for this institution."

# Step 5: Prompt template for extracting relevant details

prompt_template = """
```

```
Extract the following information from the given text:
- Founder
- Founded (year)
- Current branches
- Number of employees
- 4-line brief summary
Text: {text}
Provide the information in the following format:
Founder: <founder>
Founded: <founded>
Branches: <branches>
Employees: <employees>
Summary: <summary>
# Step 6: Take institution name as input
institution_name = input("Enter the name of the institution: ")
# Step 7: Fetch Wikipedia data for the institution
wiki_text = fetch_wikipedia_summary(institution_name)
# Step 8: Set up Cohere (Replace YOUR_COHERE_API_KEY with your actual key)
cohere_api_key = "YOUR_COHERE_API_KEY"
llm = Cohere(cohere_api_key=cohere_api_key)
# Step 9: Create the Langchain prompt and chain
prompt = PromptTemplate(input_variables=["text"], template=prompt_template)
chain = LLMChain(llm=llm, prompt=prompt)
# Step 10: Run the chain and parse the output
response = chain.run(wiki_text)
# Step 11: Parse the response using Pydantic
try:
 details = InstitutionDetails.parse_raw(response)
 print("Institution Details:")
 print(f"Founder: {details.founder}")
print(f"Founded: {details.founded}")
```

```
    print(f"Branches: {details.branches}")
    print(f"Employees: {details.employees}")
    print(f"Summary: {details.summary}")
except Exception as e:
    print("Error parsing the response:", e)
```

Output:

Enter the name of the institution: Google

Institution Details:

Founder: Larry Page, Sergey Brin

Founded: 1998

Branches: Global offices in more than 50 countries

Employees: Over 100,000

Summary: Google is a multinational technology company specializing in internetrelated services and products. It is known for its search engine, online advertising,

cloud computing, and software. Google is one of the Big Five tech companies. It was

founded by Larry Page and Sergey Brin in 1998.

Experiment 10: 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

Source Code:

```
# Step 1: Install necessary packages

!pip install langchain pydantic wikipedia-api openai

# Step 2: Import required modules

from langchain.chains import load_qa_chain

from langchain.docstore.document import Document

from langchain.llms import OpenAI

# Step 3: Load the Indian Penal Code text from a file

ipc_file_path = "path_to_your_ipc_file.txt" # Replace with the actual path to your IPC text file

# Read the IPC document

with open(ipc_file_path, "r", encoding="utf-8") as file:

 ipc_text = file.read()

# Step 4: Create a Langchain Document object

ipc_document = Document(page_content=ipc_text)

# Step 5: Set up OpenAI (or any other LLM of your choice)

llm = OpenAI(openai_api_key="YOUR_OPENAI_API_KEY", temperature=0.3) # Use

temperature=0.3 for more factual responses

# Step 6: Create a simple question-answering chain

qa_chain = load_qa_chain(llm, chain_type="stuff")

# Step 7: Chat with the chatbot

print("Chatbot for the Indian Penal Code (IPC)")

print("Ask a question about the Indian Penal Code (type 'exit' to stop):")

while True:

 user_question = input("\nYour question: ")

 if user_question.lower() == "exit":

 print("Goodbye!")

 break

 # Use the QA chain to answer the question
```

```
response = qa_chain.run(input_documents=[ipc_document], question=user_question)
print(f"Answer: {response}
```

Output:

Chatbot for the Indian Penal Code (IPC)

Ask a question about the Indian Penal Code (type 'exit' to stop):

Your question: What is Section 302 of the IPC?

Answer: Section 302 of the Indian Penal Code refers to punishment for murder,

which is punishable with death or life imprisonment and a fine.

Your question: exit

Goodbye!