PROGRAM 1

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

```python
import gensim.downloader as api
from gensim.models import KeyedVectors

# Step 1: Load a pre-trained Word2Vec model (using Gensim API
for easy access)
# You can also load a local model by providing the path to the
.bin file
# For example, Word2Vec Google News model:
# model =
KeyedVectors.load_word2vec_format('path_to_your_model.bin',
binary=True)

# Load a smaller pre-trained model via Gensim API
model = api.load("word2vec-google-news-300")  # Loading the
Google News model

# Step 2: Perform vector arithmetic (example: "King" - "Man" +
"Woman" = "Queen")
king_vector = model['king']
man_vector = model['man']
woman_vector = model['queen']

# Perform the vector operation: King - Man + Woman
result_vector = king_vector - man_vector + woman_vector

# Step 3: Find the word closest to the resulting vector (e.g.,
Queen)
result_word = model.most_similar([result_vector], topn=1)
print("Resulting word: ", result_word)
```

Output:

PROGRAM 2

Write a program to generate 5 semantically similar words for a given input. 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.

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

# Step 1: Load Pre-trained Word2Vec Model (Google News)
model = api.load("word2vec-google-news-300")  # Loading Google
News model (300-dimensional vectors)

# Step 2: Select 10 words from a specific domain (e.g., sports)
sports_words = ['football', 'basketball', 'soccer', 'tennis',
'baseball',
                'hockey', 'athlete', 'coach', 'stadium',
'referee']

# Step 3: Extract embeddings for the selected words
word_vectors = np.array([model[word] for word in sports_words])
```

```
# Step 4: Dimensionality Reduction (Using PCA or t-SNE)

# Option 1: PCA (For linear dimensionality reduction)
pca = PCA(n_components=2)
pca_result = pca.fit_transform(word_vectors)

# Option 2: t-SNE (For nonlinear dimensionality reduction)
# Uncomment the following if you prefer t-SNE
# tsne = TSNE(n_components=2, random_state=42)
# tsne_result = tsne.fit_transform(word_vectors)

# Step 5: Visualize the 2D Embeddings
plt.figure(figsize=(8, 6))

# Scatter plot for PCA
for i, word in enumerate(sports_words):
    plt.scatter(pca_result[i, 0], pca_result[i, 1])  # Plot PCA
result

    # Annotate the plot with the word
    plt.text(pca_result[i, 0] + 0.1, pca_result[i, 1] + 0.1,
word, fontsize=12)

# Optionally, if using t-SNE:
# for i, word in enumerate(sports_words):
#     plt.scatter(tsne_result[i, 0], tsne_result[i, 1])  # Plot
t-SNE result
#     plt.text(tsne_result[i, 0] + 0.1, tsne_result[i, 1] + 0.1,
word, fontsize=12)

plt.title('Word Embeddings for Sports Domain (PCA)')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.grid(True)
plt.show()
# Step 6: Generate 5 semantically similar words for an input
word
input_word = 'soccer'  # You can change this to any word in the
model

# Get the most similar words (top 5)
similar_words = model.most_similar(input_word, topn=5)

# Display the results
print(f"Top 5 words similar to '{input_word}':")
for word, similarity in similar_words:
    print(f"{word}: Similarity = {similarity:.4f}")
```

```
PROGRAM 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.

```
# Install required libraries
!pip install gensim matplotlib
# Import libraries
from gensim.models import Word2Vec
from gensim.models.word2vec import LineSentence
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE

import numpy as np
# Sample domain-specific corpus (medical domain)
medical_corpus = [
"The patient was diagnosed with diabetes and hypertension.",
"MRI scans reveal abnormalities in the brain tissue.",
"The treatment involves antibiotics and regular monitoring.",
"Symptoms include fever, fatigue, and muscle pain.",
"The vaccine is effective against several viral infections.",
"Doctors recommend physical therapy for recovery.",
"The clinical trial results were published in the journal.",
"The surgeon performed a minimally invasive procedure.",
"The prescription includes pain relievers and anti-inflammatory
drugs.",
"The diagnosis confirmed a rare genetic disorder."
]
# Preprocess corpus (tokenize sentences)
processed_corpus = [sentence.lower().split() for sentence in
medical_corpus]
# Train a Word2Vec model
print("Training Word2Vec model...")
```

```
model = Word2Vec(sentences=processed_corpus, vector_size=100,
window=5, min_count=1,
workers=4, epochs=50)
print("Model training complete!")
# Extract embeddings for visualization
words = list(model.wv.index_to_key)
embeddings = np.array([model.wv[word] for word in words])
# Dimensionality reduction using t-SNE
tsne = TSNE(n_components=2, random_state=42, perplexity=5,
n_iter=300)
tsne_result = tsne.fit_transform(embeddings)
# Visualization of word embeddings
plt.figure(figsize=(10, 8))
plt.scatter(tsne_result[:, 0], tsne_result[:, 1], color="blue")
for i, word in enumerate(words):
    plt.text(tsne_result[i, 0] + 0.02, tsne_result[i, 1] + 0.02,
word, fontsize=12)
plt.title("Word Embeddings Visualization (Medical Domain)")
plt.xlabel("Dimension 1")
plt.ylabel("Dimension 2")
plt.grid(True)
plt.show()

# Analyze domain-specific semantics
def find_similar_words(input_word, top_n=5):
try:
similar_words = model.wv.most_similar(input_word, topn=top_n)
print(f"Words similar to '{input_word}':")
for word, similarity in similar_words:
print(f" {word} ({similarity:.2f})")
except KeyError:
print(f"'{input_word}' not found in vocabulary.")
# Example: Generate semantically similar words
find_similar_words("treatment")
find_similar_words("vaccine")
```

PROGRAM 4

Use word embeddings to improve prompts for Generative AI model. Retrieve similar words using word embeddings. Use 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.

```
from openai import OpenAI
client =
OpenAI(api_key="sk-proj-GCDWhFzy7wST_dTlfCo2lmUTllFbNeIWMeWuD1vUtzNM1KhmFb
GHTfRksh_2WYBND99AQvL7-jT3BlbkFJXf8ys74orBRM_0wmfO6dNwAo3BM3_VmAlEMqouzlUm
qX1FZxXagyapgS3COp5t6FVgfnZVmasA")
completion = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[
    {"role": "user", "content": "honesty is the best"}
  ], n=3,temperature=1
)
#print(completion.to_dict())
for i in range(len(completion.choices)):
    print(completion.choices[i].message.content)
#print(completion.choices[0].message);

# The API key should be on a single line and properly terminated
completion = client.chat.completions.create(
  model="gpt-4o-mini",
  store=True,
  messages=[
    {"role": "user", "content": "honesty is the best"}
  ], n=3,temperature=1
)
#print(completion.to_dict())
for i in range (len(completion.choices)):
    print(completion.choices[i].message.content)
#print(completion.choices[0].message);
```

PROGRAM 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. Construct a short paragraph using these words.

```
!pip install gensim nltk
import gensim.downloader as api
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import random
import string
from nltk.tokenize import word_tokenize
import random
import string
import nltk
nltk.download('punkt')
nltk.download('stopwords')
# Load the Word2Vec model
word_vectors = api.load("word2vec-google-news-300")
def get_similar_words(word, top_n=5):
    try:
        similar_words = word_vectors.most_similar(word, topn=top_n)
        return [w[0] for w in similar_words]
    except KeyError:
        return []

def generate_paragraph(seed_word, num_sentences=5):
    similar_words = get_similar_words(seed_word)
    if not similar_words:
        return "No similar words found."

    # Remove stopwords and punctuation from similar words
    stop_words = set(stopwords.words('english'))
    filtered_words = [word for word in similar_words if word.lower() not
in stop_words and word not in string.punctuation]

    if not filtered_words:
        return "No suitable similar words found after filtering."
    sentences = []
    for _ in range(num_sentences):
        random.shuffle(filtered_words)
        sentence_words = filtered_words[:random.randint(3,
len(filtered_words))]
        sentence = f"Exploring the concept of '{seed_word}', we delve into
its various facets, including {', '.join(sentence_words)}."
        sentences.append(sentence)

    return " ".join(sentences)
if __name__ == "__main__":
    seed_word = input("Enter a seed word: ")
    paragraph = generate_paragraph(seed_word)
    print("\nGenerated Paragraph:")
    print(paragraph)
```

PROGRAM 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.

Pipeline: This is a high-level API that takes care of both the model and tokenizer setup internally.

"sentiment-analysis": This specifies the type of task you're interested in. In this case, it's sentiment analysis. This means the pipeline will classify input text as either "positive" or "negative" (though some models might have more granular labels like "neutral" as well).

```
!pip install transformers //transformers library provides a pipeline API
that abstracts away the complexity of using these models.
from transformers import pipeline
sentiment_analyzer = pipeline("sentiment-analysis")
sentences = [
    "I love using Hugging Face models for natural language processing!",
    "The weather today is quite gloomy and depressing.",
    "I'm excited about the upcoming holidays."
]

results = sentiment_analyzer(sentences)

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

```
OUTPUT:

Device set to use cpu
Sentence: I love using Hugging Face models for natural language processing!
Sentiment: POSITIVE, Confidence: 0.9992

Sentence: The weather today is quite gloomy and depressing.
Sentiment: NEGATIVE, Confidence: 0.9997
```

```
Sentence: I'm excited about the upcoming holidays.
Sentiment: POSITIVE, Confidence: 0.9998
```

PROGRAM 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

```python
# Install the necessary libraries first
# !pip install transformers

from transformers import pipeline

# Load the summarization pipeline from Hugging Face
summarizer = pipeline("summarization")

# Sample long passage to summarize
long_text = """
In recent years, advancements in artificial intelligence (AI) have
revolutionized the way industries operate, bringing about significant
transformations in fields such as healthcare, finance, and` manufacturing.
AI technologies, including machine learning, natural language processing,
and computer vision, have been integrated into various business processes,
leading to improved efficiency, enhanced decision-making, and better
customer experiences. However, the rapid growth of AI also raises concerns
related to ethics, privacy, and the potential for job displacement. As AI
continues to evolve, it is crucial for society to find a balance between
embracing technological innovations and addressing the challenges they
bring. Policymakers, industry leaders, and researchers must collaborate to
establish guidelines that ensure the responsible development and
```

```
deployment of AI systems, safeguarding human rights and promoting social
well-being.
"""

# Obtain the summary using the summarization pipeline
summary = summarizer(long_text, max_length=150, min_length=50,
do_sample=False) //Ensures that the model generates deterministic (not
random) summaries. If set to True, the summary can vary each time you run
the code.

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

No model was supplied, defaulted to sshleifer/distilbart-cnn-12-6 and revision
a4f8f3e (https://huggingface.co/sshleifer/distilbart-cnn-12-6).
Using a pipeline without specifying a model name and revision in production is
not recommended.
Device set to use cpu
Summary:
 AI technologies have been integrated into various business processes, leading
to improved efficiency, enhanced decision-making, and better customer
experiences . The rapid growth of AI also raises concerns related to ethics,
privacy, and the potential for job displacement . Policymakers, industry
leaders, and researchers must collaborate to establish guidelines that ensure
the responsible development and deployment of AI systems .
```

PROGRAM 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.

```
# prompt: 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.

!pip install langchain cohere langchain-community

import os
from langchain.document_loaders import GoogleDriveLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import CohereEmbeddings
from langchain.vectorstores import FAISS
from langchain.chains import RetrievalQA
from langchain.llms import Cohere
from langchain.prompts import PromptTemplate

# Replace with your actual API key
cohere_api_key = "qOJQH9S7Z7ssOz760ClbXZ6Xd3nJ0S4vX5Ue0NKO"  # Obtain this
from your Cohere account

os.environ["qOJQH9S7Z7ssOz760ClbXZ6Xd3nJ0S4vX5Ue0NKO"] = cohere_api_key

# Load document from Google Drive
loader = GoogleDriveLoader(

document_ids=["https://drive.google.com/file/d/16R4-jMSLX6_P-u4IjIDr-l/vie
w?pli=1"],
    # Optional: Specify the file type
    file_types=["txt"] # Change this to your file type (e.g. "pdf",
"docx")
)
```

```
documents = loader.load()

# Split the document into chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)


# Create embeddings and vector store
embeddings = CohereEmbeddings()
db = FAISS.from_documents(texts, embeddings)


# Create a retrieval QA chain
qa = RetrievalQA.from_chain_type(
    llm=Cohere(cohere_api_key=cohere_api_key),
    chain_type="stuff",
    retriever=db.as_retriever(),
    return_source_documents=True
)

# Example prompt template
prompt_template = """
Use the following pieces of context to answer the question at the end. If
you don't know the answer, just say that you don't know, don't try to make
up an answer.

{context}

Question: {question}
Answer:
"""
PROMPT = PromptTemplate(
    template=prompt_template, input_variables=["context", "question"]
)

# Example usage
query = "What is the main topic of the document?"
result = qa({"query": query, "prompt": PROMPT})
```

```python
# Display the answer and source documents in a specific format
print("Question:", query)
print("\nAnswer:")
print(result["result"])
print("\nSource Documents:")
for document in result["source_documents"]:
    print(f"- {document.page_content[:100]}...")  # Print the first 100 
chars of each document
```

PROGRAM 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.

```
!pip install wikipedia
import wikipedia
from pydantic import BaseModel, ValidationError
from typing import Optional
import re

# Define the Pydantic schema for the institution details
class InstitutionInfo(BaseModel):
    name: str
    founder: Optional[str]
    founded: Optional[str]
    branches: Optional[str]
    employees: Optional[str]
    summary: Optional[str]

def extract_institution_details(institution_name: str) -> InstitutionInfo:
    try:
        # Fetch the Wikipedia page content
        page = wikipedia.page(institution_name)
        content = page.content

        # Initialize fields
        founder = None
        founded = None
        branches = None
        employees = None

        # Extract founder
        founder_match = re.search(r'(?i)founder[s]?:\s*(.*)', content)
        if founder_match:
            founder = founder_match.group(1).split('\n')[0].strip()

        # Extract founded date
        founded_match = re.search(r'(?i)founde[dn]:\s*(.*)', content)
        if founded_match:
            founded = founded_match.group(1).split('\n')[0].strip()

        # Extract branches
        branches_match = re.search(r'(?i)branches:\s*(.*)', content)
        if branches_match:
            branches = branches_match.group(1).split('\n')[0].strip()

        # Extract number of employees
        employees_match = re.search(r'(?i)employee[s]?:\s*(.*)', content)
        if employees_match:
            employees = employees_match.group(1).split('\n')[0].strip()

        # Generate a brief 4-line summary
        summary_sentences = wikipedia.summary(institution_name,
sentences=4)

        # Create the InstitutionInfo object
        institution_info = InstitutionInfo(
            name=institution_name,
            founder=founder,
            founded=founded,
            branches=branches,
            employees=employees,
            summary=summary_sentences
        )

        return institution_info

    except wikipedia.exceptions.DisambiguationError as e:
        print(f"Disambiguation error: {e.options}")
    except wikipedia.exceptions.PageError:
        print("Page not found.")
    except ValidationError as ve:
        print(f"Validation error: {ve}")
```

```python
# Example usage
if __name__ == "__main__":
    institution_name = input("Enter the name of the institution: ")
    info = extract_institution_details(institution_name)
    if info:
        print("\nExtracted Institution Details:")
print(info.model_dump_json(indent=4)) # Use model_dump_json with indent
```

```
Enter the name of the institution: HKBK College of Engineering

Extracted Institution Details:
{
    "name": "HKBK College of Engineering",
    "founder": null,
    "founded": null,
    "branches": null,
    "employees": null,
    "summary": "HKBK College of Engineering was established in 1997 and is
affiliated to Visvesvaraya Technological University (VTU) and approved by
All India Council for Technical Education, New Delhi.\n\n\n== Campus
==\n\n\n=== Location ===\nThe campus is located in the city, opposite to
Manyata Tech park at Nagwara, on the north side of Bangalore,
Karnataka.\nHKBK College of Engineering is spread out on 16 acres of land,
opposite to Manyata Tech Park,  Nagavara, Bangalore.\n\n\n===
Infrastructure and facilities ===\nThe college has libraries, discussion
rooms, large playgrounds, Innovation center and state-of-the-art labs."
}
```

PROGRAM 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.

```
# prompt: 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.


import os

from langchain.document_loaders import TextLoader

from langchain.text_splitter import CharacterTextSplitter

from langchain.embeddings import CohereEmbeddings

from langchain.vectorstores import FAISS

from langchain.chains import RetrievalQA

from langchain.llms import Cohere

from langchain.prompts import PromptTemplate

from google.colab import drive


# Mount Google Drive

drive.mount('/content/drive')


# Replace with your actual Cohere API key

cohere_api_key = "qOJQH9S7Z7ssOz760ClbXZ6Xd3nJ0S4vX5Ue0NKO"  # Replace
with your actual API key

os.environ["COHERE_API_KEY"] = cohere_api_key
```

```
# Load the document

# Make sure the file path is correct

file_path = '/content/drive/MyDrive/indian_penal_code.txt' # Update with
the correct file path in your Google Drive

try:

    loader = TextLoader(file_path)

    documents = loader.load()

except FileNotFoundError:

    print(f"Error: File not found at {file_path}. Please check the file
path and ensure the file exists.")

    exit()




# Split the documents

text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)

texts = text_splitter.split_documents(documents)


# Create embeddings

embeddings = CohereEmbeddings()


# Create vector store

docsearch = FAISS.from_documents(texts, embeddings)


# Create the LLM

llm = Cohere(cohere_api_key=cohere_api_key)
```

```python
# Create a prompt template

prompt_template = """Use the following pieces of context to answer the
question at the end. If you don't know the answer, just say that you
don't know, don't try to make up an answer.


{context}


Question: {question}

Answer:"""

PROMPT = PromptTemplate(

    template=prompt_template, input_variables=["context", "question"]

)


# Create the QA chain

qa = RetrievalQA.from_chain_type(

    llm=llm,

    chain_type="stuff",

    retriever=docsearch.as_retriever(),

    return_source_documents=True,

    chain_type_kwargs={"prompt": PROMPT}

)


# Chatbot loop

while True:

    query = input("Ask a question about the Indian Penal Code (or type
'exit' to quit): ")

    if query.lower() == 'exit':
        break

    try:

        result = qa({"query": query})

        print(result["result"])

    except Exception as e:

        print(f"An error occurred: {e}")
```