```
1)  import gensim.downloader as api
from scipy.spatial.distance import cosine

print("Loading Word2Vec model...")
model = api.load("word2vec-google-news-300")
print("Model loaded successfully.\n")

vector = model['king']
print("First 10 dimensions of 'king' vector:")
print(vector[:10], "\n")

print("Top 10 words most similar to 'king':")
for word, similarity in model.most_similar('king'):
    print(f"{word}: {similarity:.4f}")
print()

result = model.most_similar(positive=['king', 'woman'], negative=['man'], topn=1)
print("Analogy - 'king' - 'man' + 'woman' ≈ ?")
print(f"Result: {result[0][0]} (Similarity: {result[0][1]:.4f})\n")

print("Analogy - 'paris' + 'italy' - 'france' ≈ ?")
for word, similarity in model.most_similar(positive=['paris', 'italy'], negative=['france']):
    print(f"{word}: {similarity:.4f}")
print()

print("Analogy - 'walking' + 'swimming' - 'walk' ≈ ?")
for word, similarity in model.most_similar(positive=['walking', 'swimming'], negative=['walk']):
    print(f"{word}: {similarity:.4f}")
print()

similarity = 1 - cosine(model['king'], model['queen'])
print(f"Cosine similarity between 'king' and 'queen': {similarity:.4f}")
```

```
2)!pip install gensim matplotlib scikit-learn --quiet

import gensim.downloader as api
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from gensim.models import Word2Vec

print("Loading pre-trained Word2Vec model...")
word_vectors = api.load("word2vec-google-news-300")
print("Model loaded.")

words = ["computer", "laptop", "AI", "machine", "robot", "software", "hardware", "algorithm",
"network"]
vectors = np.array([word_vectors[word] for word in words])
```

```
def plot_embeddings(vectors, words, method="PCA"):
    if method == "PCA":
        reduced = PCA(n_components=2).fit_transform(vectors)
    else:
        reduced = TSNE(n_components=2, perplexity=5, random_state=42).fit_transform(vectors)

    plt.figure(figsize=(10, 7))
    plt.scatter(reduced[:, 0], reduced[:, 1], color="skyblue", s=100)
    for i, word in enumerate(words):
        plt.annotate(word, (reduced[i, 0] + 0.02, reduced[i, 1] + 0.02), fontsize=12)
    plt.title(f"Pretrained Word Embedding Visualization using {method}")
    plt.grid(True)
    plt.show()

plot_embeddings(vectors, words, method="PCA")
plot_embeddings(vectors, words, method="t-SNE")
```

3)

```
import gensim
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from gensim.models import Word2Vec

medical_sentences = [
    ['patient', 'diagnosed', 'cancer', 'treatment', 'chemotherapy'],
    ['doctor', 'prescribes', 'medication', 'therapy', 'recovery'],
    ['hospital', 'surgery', 'nurse', 'care', 'treatment'],
    ['virus', 'infection', 'vaccine', 'immune', 'system'],
    ['diabetes', 'insulin', 'blood', 'sugar', 'health'],
    ['heart', 'disease', 'cardiac', 'attack', 'stroke'],
    ['brain', 'neuroscience', 'mental', 'health', 'psychology'],
    ['radiology', 'MRI', 'X-ray', 'diagnosis', 'scan'],
    ['nutrition', 'diet', 'exercise', 'wellness', 'fitness'],
    ['epidemic', 'pandemic', 'COVID', 'quarantine', 'vaccine']
]

model = Word2Vec(sentences=medical_sentences, vector_size=100, window=3, min_count=1,
workers=4)

similar_words = model.wv.most_similar('treatment', topn=5)

print("\nTop 5 words similar to 'treatment':")
for word, score in similar_words:
    print(f"{word}: {score:.4f}")

words = list(model.wv.index_to_key)
word_vectors = np.array([model.wv[word] for word in words])

tsne = TSNE(n_components=2, random_state=0, perplexity=3)
word_vectors_2d = tsne.fit_transform(word_vectors)
```

```
plt.figure(figsize=(10, 8))
for i, word in enumerate(words):
    plt.scatter(word_vectors_2d[i, 0], word_vectors_2d[i, 1])
    plt.text(word_vectors_2d[i, 0] + 0.05, word_vectors_2d[i, 1] + 0.05, word, fontsize=12)

plt.title("t-SNE Visualization of Custom Medical Word Embeddings")
plt.xlabel("Component 1")
plt.ylabel("Component 2")
plt.grid(True)
plt.show()
```

```
4) !pip install sentence-transformers
from sentence_transformers import SentenceTransformer, util
import torch

model = SentenceTransformer('all-MiniLM-L6-v2')

def get_similar_words(word, top_k=5):
 embeddings = model.encode([word], convert_to_tensor=True)
 cosine_scores = util.pytorch_cos_sim(embeddings, model.encode(['dog', 'cat', 'animal', 'pet', 'mammal',
'food'], convert_to_tensor=True))
 top_results = torch.topk(cosine_scores[0], k=top_k)
 similar_words = []
 for score, idx in zip(top_results[0], top_results[1]):
  similar_words.append(['dog', 'cat', 'animal', 'pet', 'mammal', 'food'][idx.item()])
 return similar_words

def enrich_prompt(prompt):
 words = prompt.split()
 enriched_prompt = ""
 for word in words:
  similar_words = get_similar_words(word)
  enriched_prompt += word + " (" + ", ".join(similar_words) + ") "
 return enriched_prompt

original_prompt = "Describe the characteristics of a dog."
enriched_prompt = enrich_prompt(original_prompt)

def generate_response(prompt):
 response = f"Response for prompt: {prompt}"
 return response

original_response = generate_response(original_prompt)
enriched_response = generate_response(enriched_prompt)

print(f"Original Prompt: {original_prompt}")
print(f"Original Response: {original_response}")
print(f"Enriched Prompt: {enriched_prompt}")
print(f"Enriched Response: {enriched_response}")
```

```
5)  from sentence_transformers import SentenceTransformer, util
import torch
model = SentenceTransformer('all-MiniLM-L6-v2')
def get_similar_words(word, top_k=5):
    """
    Finds similar words using word embeddings.

    Args:
        word: The word to find similar words for.
        top_k: The number of similar words to return.

    Returns:
        A list of similar words.
    """
    embeddings = model.encode([word], convert_to_tensor=True)

    vocabulary = ['dog', 'cat', 'animal', 'pet', 'mammal', 'food', 'happy', 'sad', 'excited', 'angry']
    cosine_scores = util.pytorch_cos_sim(
        embeddings, model.encode(vocabulary, convert_to_tensor=True)
    )

    top_results = torch.topk(cosine_scores[0], k=top_k)
    similar_words = []
    for score, idx in zip(top_results[0], top_results[1]):
        similar_words.append(vocabulary[idx.item()])

    return similar_words

def create_sentence(seed_word):
    """
    Creates a short paragraph using similar words.

    Args:
        seed_word: The seed word to start with.

    Returns:
        A short paragraph.
    """
    similar_words = get_similar_words(seed_word)
    sentence = (
        f"The {seed_word} was {similar_words[0]}, and it made me feel {similar_words[1]}. "
        f"I wondered if it was like a {similar_words[2]}, or maybe more like a {similar_words[3]}."
    )
    return sentence

seed_word = "sunrise"
paragraph = create_sentence(seed_word)
print(paragraph)
```

```
6)  from transformers import pipeline
sentiment_pipeline = pipeline("sentiment-analysis")
def analyze_sentiment(text):

  result = sentiment_pipeline(text)[0]
  label = result["label"]
  confidence = result["score"]
  return f"Sentiment: {label} (Confidence: {confidence:.2f})"

texts = [
"I love this product! It's amazing.",
"This is the worst experience I've ever had.", "The movie was okay, but nothing special.", "I'm extremely
happy with my new laptop!",
"This service is so frustrating and disappointing."
]

for text in texts: print(f"Text: {text}")
print(analyze_sentiment(text))
print("-" * 50)
```

```
7)from transformers import pipeline

summarizer = pipeline("summarization")

def summarize_text(text, max_length=130, min_length=30):
 summary = summarizer(text, max_length=max_length, min_length=min_length,
do_sample=False)[0]['summary_text']
 return summary
passage = """
The Gemini API gives you access to Gemini models created by Google
DeepMind. Gemini
models are built from the ground up to be multimodal, so you can reason
seamlessly across
text, images, code, and audio.
"""

summary = summarize_text(passage)
summary
```

```
9)
!pip install wikipedia
!pip install pydantic

import wikipedia
from pydantic import BaseModel, Field
from typing import List, Optional

class InstitutionDetails(BaseModel):
    """
    Pydantic schema for institution details.
    """
    founder: Optional[str] = Field(None, description="Founder of the institution")
    founded: Optional[int] = Field(None, description="Year of founding")
    branches: Optional[List[str]] = Field(None, description="Current branches of the institution")
    num_employees: Optional[int] = Field(None, description="Number of employees")
    summary: Optional[str] = Field(None, description="A brief summary of the institution")

def parse_wikipedia_page(page_title: str) -> InstitutionDetails:
    """
    Parses the Wikipedia page content to extract the relevant details.

    Args:
        page_title (str): The title of the Wikipedia page.

    Returns:
        InstitutionDetails: Parsed institution details.
    """
    details = InstitutionDetails()
    try:

        details.summary = wikipedia.summary(page_title, sentences=4)


    except Exception as e:
        print(f"Error parsing Wikipedia page: {e}")

    return details

if __name__ == "__main__":
    institution_name = input("Enter the institution name: ")

    try:
        page = wikipedia.page(institution_name)
        details = parse_wikipedia_page(institution_name)
        print(details.model_dump_json(indent=2))

    except wikipedia.exceptions.PageError:
        print(f"Wikipedia page not found for '{institution_name}'")
    except wikipedia.exceptions.DisambiguationError as e:
        print(f"Disambiguation error: Options include {e.options}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

10)

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

class IPCSection(BaseModel):
 section_number: str = Field(..., description="Section number of the IPC")
 description: Optional[str] = Field(None, description="Description of the
section")
 punishment: Optional[str] = Field(None, description="Punishment
prescribed for the offence")

def parse_ipc_section(section_text: str) -> IPCSection:
 section = IPCSection(section_number=section_text.split(". ")[0])
 description_match = re.search(r"(?<=Whoever).*(?=\s*Shall be punished)",
section_text, re.DOTALL)
 punishment_match = re.search(r"(?<=Shall be punished).*(?=\.)",
section_text, re.DOTALL)
 section.description = description_match.group(0).strip() if
description_match else "Description not found"
 section.punishment = punishment_match.group(0).strip() if
punishment_match else "Punishment not found"
 return section

def search_ipc(query: str) -> List[IPCSection]:
 try:
  page = wikipedia.page("Indian Penal Code")
  content = page.content
  sections = []
  matches = re.findall(rf"{query}.*?(?=\n\d+\.)", content, re.DOTALL)
  for match in matches:
   sections.append(parse_ipc_section(match.strip()))
  return sections
 except wikipedia.exceptions.PageError:
  print(f"Wikipedia page not found for 'Indian Penal Code'")
  return []
 except Exception as e:
  print(f"An unexpected error occurred: {e}")
  return []

if __name__ == "__main__":
 while True:
  user_query = input("Ask a question about the Indian Penal Code (or type
'exit'): ")
  if user_query.lower() == 'exit':
   break
  results = search_ipc(user_query)
  if results:
   for section in results:
    print(section.model_dump_json(indent=2))
  else:
   print("No matching sections found.")
```