



Arka Educational & Cultural Trust (Regd.)

Jain Institute of Technology, Davangere

(A Unit of Jain Group of Institutions, Bangalore)

(Affiliated to VTU, Belagavi | Approved by AICTE, New Delhi | Recognized by Government of Karnataka)



Department of Computer Science and Engineering

(Accredited by NBA, New Delhi, validity up to 30.06.2026)

GENERATIVE Ai

Laboratory Manual

Course Code: BAIL657C

VI Sem

“The art of Creation ,amplified by AI”

Course Coordinator

Dr. Latharani T R

Manual Credits:

Savita P N

Mandara R



Vision & Mission of the Program

Vision of the Department:

To develop socially responsible computer engineers and entrepreneurs with strong academic excellence, technical backgrounds, research and innovation, intellectual skills and creativity to cater the needs of IT Industry and society by adopting professional ethics.

Mission of the Department:

| | |
|----|---|
| M1 | To impart center of excellence by offering technical education and imbibing experiential learning skills to achieve teaching learning process. |
| M2 | Providing a Platform to discover and engage research and innovation strengths, talents, passions through collaborations, government, private agencies and industries. |
| M3 | Creating an environment to inculcate moral principles, professionalism and responsibilities towards the society. |

Template for Practical Course and if AEC is a practical Course Annexure-V

| Generative AI | | Semester | 6 |
|---|--|------------|-----|
| Course Code | BAIL657C | CIE Marks | 50 |
| Teaching Hours/Week (L:T:P: S) | 0:0:1:0 | SEE Marks | 50 |
| Credits | 01 | Exam Hours | 100 |
| Examination type (SEE) | Practical | | |
| Course objectives: | | | |
| <ul style="list-style-type: none">• Understand the principles and concepts behind generative AI models• Explain the knowledge gained to implement generative models using Prompt design frameworks.• Apply various Generative AI applications for increasing productivity.• Develop Large Language Model-based Apps. | | | |
| Sl.NO | Experiments | | |
| 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. | 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. | | |
| 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. | | |

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.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation (CIE):

CIE marks for the practical course are **50 Marks**.

The split-up of CIE marks for record/ journal and test are in the ratio **60:40**.

- Each experiment is to be evaluated for conduction with an observation sheet and record write-up. Rubrics for the evaluation of the journal/write-up for hardware/software experiments are designed by the faculty who is handling the laboratory session and are made known to students at the beginning of the practical session.
- Record should contain all the specified experiments in the syllabus and each experiment write-up will be evaluated for 10 marks.
- Total marks scored by the students are scaled down to **30 marks** (60% of maximum marks).
- Weightage to be given for neatness and submission of record/write-up on time.
- Department shall conduct a test of 100 marks after the completion of all the experiments listed in the syllabus.
- In a test, test write-up, conduction of experiment, acceptable result, and procedural knowledge will carry a weightage of 60% and the rest 40% for viva-voce.
- The suitable rubrics can be designed to evaluate each student's performance and learning ability.
- The marks scored shall be scaled down to **20 marks** (40% of the maximum marks).

The Sum of scaled-down marks scored in the report write-up/journal and marks of a test is the total CIE marks scored by the student.

Semester End Evaluation (SEE):

- SEE marks for the practical course are 50 Marks.
- SEE shall be conducted jointly by the two examiners of the same institute, examiners are appointed by the Head of the Institute.

- The examination schedule and names of examiners are informed to the university before the conduction of the examination. These practical examinations are to be conducted between the schedule mentioned in the academic calendar of the University.
- All laboratory experiments are to be included for practical examination.
- (Rubrics) Breakup of marks and the instructions printed on the cover page of the answer script to be strictly adhered to by the examiners. **OR** based on the course requirement evaluation rubrics shall be decided jointly by examiners.
- Students can pick one question (experiment) from the questions lot prepared by the examiners jointly.
- Evaluation of test write-up/ conduction procedure and result/viva will be conducted jointly by examiners.

General rubrics suggested for SEE are mentioned here, writeup-20%, Conduction procedure and result in -60%, Viva-voce 20% of maximum marks. SEE for practical shall be evaluated for 100 marks and scored marks shall be scaled down to 50 marks (however, based on course type, rubrics shall be decided by the examiners)

Change of experiment is allowed only once and 15% of Marks allotted to the procedure part are to be made zero.

The minimum duration of SEE is 02 hours

Suggested Learning Resources:

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.

Web links and Video Lectures (e-Resources):

- https://www.w3schools.com/gen_ai/index.php
- <https://youtu.be/eTPiL3DF27U>
- <https://youtu.be/je6AlVeGOV0>
- <https://youtu.be/RLVqsA8ns6k>
- <https://youtu.be/0SAKM7wiC-A>
- https://youtu.be/28_9xMyrdjg
- <https://youtu.be/8iuijz-c-EBw>
- <https://youtu.be/7oQ8VtEKcgE>
- <https://youtu.be/seXp0VWWZV0>

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

```
1 #1st prgm
2 !pip install gensim
3 from gensim.downloader import load
4 print("Loading pre-trained GloVe model (50 dimensions)...")
5 model = load("glove-wiki-gigaword-50")
6 def ewr():
7     result = model.most_similar(positive=['king', 'woman'], negative=['man'], topn=1)
8     print("\nking - man + woman = ?", result[0][0])
9     print("similarity:", result[0][1])
10    result = model.most_similar(positive=['paris', 'italy'], negative=['france'], topn=1)
11    print("\nparis - france + italy = ?", result[0][0])
12    print("similarity:", result[0][1])
13    result = model.most_similar(positive=['programming'], topn=5)
14    print("\nTop 5 words similar to 'programming':")
15    for word, similarity in result:
16        print(word, similarity)
17 ewr()
```

Output :

```
Requirement already satisfied: gensim in c:\users\sinch\anaconda3\lib\site-packages (4.3.3)
Requirement already satisfied: numpy<2.0,>=1.18.5 in c:\users\sinch\anaconda3\lib\site-packages (from gensim) (1.26.4)
Requirement already satisfied: scipy<1.14.0,>=1.7.0 in c:\users\sinch\anaconda3\lib\site-packages (from gensim) (1.13.1)
Requirement already satisfied: smart-open>=1.8.1 in c:\users\sinch\anaconda3\lib\site-packages (from gensim) (5.2.1)
Loading pre-trained GloVe model (50 dimensions)...

king - man + woman = ? queen
similarity: 0.8523604273796082

paris - france + italy = ? rome
similarity: 0.8465589284896851

Top 5 words similar to 'programming':
network 0.7707955241203308
interactive 0.7613597512245178
format 0.7584694623947144
channels 0.753067672252655
networks 0.752894937992096
```

Explanation :

Step 1: Install & Import Gensim

`!pip install gensim`

`from gensim.downloader import load`

`!pip install gensim`: Installs the Gensim library — used for working with word vectors (word embeddings).

`from gensim.downloader import load`: Lets you download pre-trained word models, like GloVe, directly.

Step 2: Load the Pre-trained GloVe Model

```
print("Loading pre-trained GloVe model (50 dimensions)...")
```

```
model = load("glove-wiki-gigaword-50")
```

Loads the GloVe model with 50-dimensional vectors trained on Wikipedia + Gigaword corpus.

GloVe gives each word a numerical vector representing its meaning based on context.

Step 3: Define Function to Play with Word Vectors

```
def ewr():
```


A function ewr() is defined to perform word vector operations like similarity and analogy.

Step 4: Word Analogies & Similarities

```
result = model.most_similar(positive=['king', 'woman'], negative=['man'], topn=1)
```

This performs: king - man + woman

It finds the word closest in meaning to that result.

Expected output: queen 

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

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

Prints the predicted word and its cosine similarity score (how close the vectors are).

Step 5: Another Analogy

```
result = model.most_similar(positive=['paris', 'italy'], negative=['france'], topn=1)
```

This does: paris - france + italy

It tries to find a city in Italy like Paris is to France.

Example output: rome or milan

Step 6: Find Similar Words

```
result = model.most_similar(positive=['programming'], topn=5)
```

Finds the top 5 words most similar to "programming".

Example: software, coding, development, etc.

for word, similarity in result:

```
    print(word, similarity)
```

Prints each similar word with how close it is in meaning (similarity score).

Step 7: Run the Function

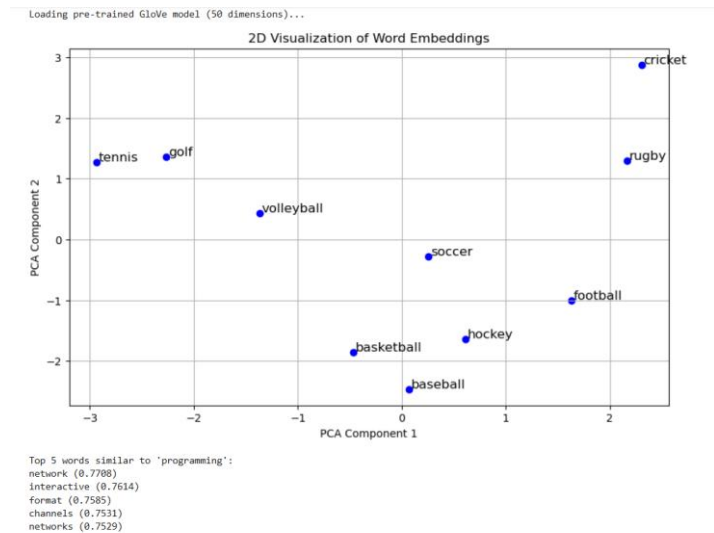
`ewr()`

Calls the function to run all the logic above.

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.

```
1 #2nd prgm
2 import matplotlib.pyplot as plt
3 from sklearn.decomposition import PCA
4 from gensim.downloader import load
5 def reduce_dimensions(embeddings):
6     pca = PCA(n_components=2)
7     reduced_embeddings = pca.fit_transform(embeddings)
8     return reduced_embeddings
9 def visualize_embeddings(words, reduced_embeddings):
10     plt.figure(figsize=(10, 6))
11     for i, word in enumerate(words):
12         x, y = reduced_embeddings[i]
13         plt.scatter(x, y, color='blue', marker='o')
14         plt.text(x + 0.02, y + 0.02, word, fontsize=12)
15     plt.title("2D Visualization of Word Embeddings")
16     plt.xlabel("PCA Component 1")
17     plt.ylabel("PCA Component 2")
18     plt.grid()
19     plt.show()
20 def get_similar_words(word, model):
21     print(f"Top 5 words similar to '{word}':")
22     similar_words = model.most_similar(word, topn=10)
23     for similar_word, similarity in similar_words:
24         print(f"{similar_word} ({similarity:.4f})")
25 print("Loading pre-trained GloVe model (50 dimensions)...")
26 model = load("glove-wiki-gigaword-50")
27 words = ['football', 'basketball', 'soccer', 'tennis', 'cricket',
28         'hockey', 'baseball', 'golf', 'volleyball', 'rugby']
29 embeddings = [model[word] for word in words]
30 reduced_embeddings = reduce_dimensions(embeddings)
31 visualize_embeddings(words, reduced_embeddings)
32 get_similar_words("programming", model)
```

Output :



Explanation :

Step 1: Import Required Libraries

```
import matplotlib.pyplot as plt  
  
from sklearn.decomposition import PCA  
  
from gensim.downloader import load  
  
matplotlib: For plotting.
```

PCA: Used to reduce high-dimensional word vectors to 2D.

gensim: For loading pre-trained GloVe embeddings.

Step 2: Function to Reduce Dimensions

```
def reduce_dimensions(embeddings):  
    pca = PCA(n_components=2)  
    reduced_embeddings = pca.fit_transform(embeddings)  
    return reduced_embeddings
```

Uses PCA to convert 50D word vectors into 2D, which can be plotted.

Step 3: Function to Visualize Words

```
def visualize_embeddings(words, reduced_embeddings):
```

Plots each word on a 2D graph using its reduced vector.

Each point is labeled with the word name.

Helps you "see" how closely words are grouped.

Step 4: Function to Find Similar Words

```
def get_similar_words(word, model):
```

Prints the top 5 words that are most similar to the given word (programming here).

Step 5: Load GloVe Model

```
model = load("glove-wiki-gigaword-50")
```

Loads 50-dimensional pre-trained word vectors.

Step 6: Prepare Data for Plotting

```
words = ['football', 'basketball', 'soccer', ...]
```

```
embeddings = [model[word] for word in words]
```

Picks 10 sports-related words.

Gets their vector representations.

Step 7: Reduce & Visualize

```
reduced_embeddings = reduce_dimensions(embeddings)
```

```
visualize_embeddings(words, reduced_embeddings)
```

Reduces dimensions using PCA.

Plots the 10 words in 2D space so you can see clusters and relationships.

Step 8: Find Similar Words to “Programming”

```
get_similar_words("programming", model)
```

Prints words like coding, developer, etc., showing semantic similarity.

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.

```
1 #3rd
2 from gensim.models import Word2Vec
3 from sklearn.decomposition import PCA
4 import matplotlib.pyplot as plt
5 corpus = [
6     "The patient was diagnosed with diabetes and hypertension.",
7     "RI scans reveal abnormalities in the brain tissue.",
8     "The treatment involves antibiotics and regular monitoring.",
9     "Symptoms include fever, fatigue, and muscle pain.",
10    "The vaccine is effective against several viral infections.",
11    "Doctors recommend physical therapy for recovery.",
12    "The clinical trial results were published in the journal.",
13    "The surgeon performed a minimally invasive procedure.",
14    "The prescription includes pain relievers and anti-inflammatory drugs.",
15    "The diagnosis confirmed a rare genetic disorder."
16 ]
17 tokenized_corpus = [sentence.lower().split() for sentence in corpus]
18 model = Word2Vec(sentences=tokenized_corpus, vector_size=5, window=2, min_count=1, epochs=5)
19 word = input("Enter a word: ").lower()
20 if word in model.wv:
21     similar = model.wv.most_similar(word, topn=5)
22     print(f"Words similar to '{word}':")
23     for i, (w, score) in enumerate(similar, 1):
24         print(f"{i}. {w} (Similarity: {score:.4f})")
25 else:
26     print("Word not found in vocabulary.")
27 words = list(model.wv.index_to_key)
28 word_vectors = model.wv[words]
29 pca = PCA(n_components=2)
30 result = pca.fit_transform(word_vectors)
31 plt.figure(figsize=(10, 8))
32 plt.scatter(result[:, 0], result[:, 1])
33 for i, word in enumerate(words):
34     plt.annotate(word, xy=(result[i, 0], result[i, 1]))
35 plt.title("Word Embeddings (PCA Projection)")
36 plt.xlabel("PCA 1")
37 plt.ylabel("PCA 2")
38 plt.grid(True)
39 plt.show()
```

```
Enter a word: patient
Words similar to 'patient':
1. for (Similarity: 0.8092)
2. viral (Similarity: 0.7880)
3. disorder. (Similarity: 0.7471)
4. include (Similarity: 0.6800)
5. clinical (Similarity: 0.6754)
```



```
corpus = [
    "The patient was diagnosed with diabetes and hypertension.",
    ...
]
```

These sentences are the training data for Word2Vec.

```
tokenized_corpus = [sentence.lower().split() for sentence in corpus]
```

This gives input format like: `[['the', 'patient', 'was', ...], [...]]`

```
model = Word2Vec(sentences=tokenized_corpus, vector_size=5, window=2,
min count=1, epochs=5)
```

vector size=5: Each word is represented as a 5-dimensional vector.

window=2: Looks at 2 words before and after the target word.

min_count=1: Keeps all words (no minimum frequency).

epochs=5: Number of times the model goes over the data.

Step 4: Find Similar Words

```
word = input("Enter a word: ").lower()
```

```
if word in model.wv:
```

```
    similar = model.wv.most_similar(word, topn=5)
```

Prompts the user to enter a word.

If the word exists in the vocabulary:

It finds the top 5 most similar words based on trained embeddings.

Else, shows an error.

Step 5: Visualize Word Embeddings

```
words = list(model.wv.index_to_key)
```

```
word_vectors = model.wv[words]
```

```
pca = PCA(n_components=2)
```

```
result = pca.fit_transform(word_vectors)
```

Gets the word list and their vector representations.

Uses PCA to reduce 5D vectors to 2D for visualization.

Step 6: Plot the Words in 2D

```
plt.scatter(result[:, 0], result[:, 1])
```

```
for i, word in enumerate(words):
```

```
    plt.annotate(word, xy=(result[i, 0], result[i, 1]))
```

Creates a scatter plot of words in 2D.

Labels each point with the corresponding word.

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.

```
1 #4th
2 !pip install cohere gensim
3 import cohere
4 import gensim.downloader as api
5 co = cohere.Client("iKCV07rBnBU5uYH40gPabT4cFY8DkEiZnZgxtIrr")
6 print("Loading word embeddings...")
7 model = api.load("glove-wiki-gigaword-100")
8 print("Model loaded successfully.")
9 prompt = "write an essay on natural disaster"
10 def get_first_enriched_prompt(prompt, topn=3):
11     for word in prompt.split():
12         try:
13             similar_words = model.most_similar(word.strip('.,!?').lower(), topn=topn)
14             for sim, _ in similar_words:
15                 enriched = prompt.replace(word, sim)
16                 return enriched
17         except:
18             continue
19     return None
20 def get_response(text):
21     try:
22         return co.chat(model="command-r", message=text).text.strip()
23     except Exception as e:
24         return f"Error: {e}"
25 print(f"\nOriginal Prompt:\n{prompt}\nResponse:\n{get_response(prompt)}")
26 enriched_prompt = get_first_enriched_prompt(prompt)
27 if enriched_prompt:
28     print(f"\nEnriched Prompt:\n{enriched_prompt}\nResponse:\n{get_response(enriched_prompt)}")
29 else:
30     print("\nNo enriched prompt could be generated.")
```

Output :

Original Prompt:

write an essay on natural disaster

Response:

Natural disasters are some of the most powerful and devastating events that occur on our planet. They are often described as acts of nature, beyond human control, which can wreak havoc on communities and ecosystems. From earthquakes and hurricanes to floods and wildfires, these events showcase the raw strength of nature and remind us of our vulnerability.

One of the most destructive natural disasters is the earthquake. Occurring when there is a sudden release of energy along a fault line in the Earth's crust, earthquakes can lead to massive damage to infrastructure and loss of life. The impacts can be felt for years, as affected communities struggle to rebuild amidst the ruins. The 2010 earthquake in Haiti, which killed over 300,000 people and displaced 1.5 million, is a tragic example of the devastation earthquakes can cause. Similarly, volcanic eruptions, while less frequent, can also wreak havoc, releasing ash and molten rock, and threatening nearby populations.

Another formidable force of nature is hurricanes and typhoons, characterized by their powerful winds and intense rainfall. These rotating storm systems can cause extensive damage to coastal areas and islands, leading to flooding, infrastructure collapse, and widespread destruction. The 2017 Hurricane Maria, which impacted Puerto Rico, caused a humanitarian crisis, with widespread flooding, landslides, and a death toll exceeding 3,000. The recovery process was lengthy and challenging, showcasing the enduring impact of these meteorological phenomena.

Floods are also frequent natural disasters, often caused by intense rainfall, melting snow, or the rupturing of dams. They can swiftly engulf communities, carrying devastating forces that damage homes, infrastructure, and the environment. Climate change has intensified rainfall patterns, leading to more frequent and severe flooding in many regions. The 2019 floods in India, for instance, displaced millions, underlining the ongoing threat these events pose to human settlements.

Enriched Prompt:

writing an essay on natural disaster

Response:

Natural disasters are phenomena that occur naturally and often have devastating effects on the environment and human society. They can be incredibly destructive, causing immense damage to infrastructure, property, and the natural landscape, while also disrupting the lives and livelihoods of those affected.

When writing an essay on natural disasters, you might consider the following points:

1. **Definition and Types:** Begin by explaining the definition of a natural disaster and listing the various types, including earthquakes, hurricanes, tornadoes, floods, tsunamis, wildfires, and other severe weather events. Discuss the distinct characteristics and impacts of each.
2. **Causes and Effects:** Explore the causes of natural disasters. This can involve geological factors, such as tectonic plate movements triggering earthquakes and volcanic eruptions, or meteorological phenomena leading to storms and floods. Explain the scientific processes behind these events and their subsequent effects on the environment and human settlements.

3. **Destruction and Impact:** Describe the immense destruction that natural disasters can wreak. Explain the direct impacts on physical infrastructure, homes, and public amenities, as well as the loss of life and livelihood. Discuss the long-term effects, such as displacement of populations, economic losses, and the strain on emergency response and relief efforts.

4. **Preparedness and Response:** Evaluate the measures that can be taken to mitigate the impacts of natural disasters. This may include early warning systems, emergency planning, and evacuation procedures. Examine the roles of government agencies, relief organizations, and community groups in responding to and managing these events. Highlight successful stories of resilience and the human spirit in the face of natural calamities.

Explanation:

Step 1: Load Tools & Models

```
!pip install cohere gensim
```

```
model = api.load("glove-wiki-gigaword-100")
```

```
co = cohere.Client("...")
```

Uses GloVe word embeddings to understand word meanings.

Connects to Cohere's powerful text generation model (like ChatGPT).

Step 2: Input Prompt

```
prompt = "write an essay on natural disaster"
```

Simple prompt asking the AI to generate an essay.

Step 3: Enrich the Prompt

```
get_first_enriched_prompt()
```

For each word in the prompt:

Finds similar words using GloVe.

Replaces one word in the prompt with its most similar alternative.

Returns a new "enriched" version of the prompt.

Example:

Original: write an essay on natural disaster

Enriched: compose an essay on natural disaster

Step 4: Generate Response Using Cohere AI

```
get_response(prompt)
```

```
get_response(enriched_prompt)
```

Sends both the original and enriched prompts to Cohere AI.

Prints the essay responses for both.

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.

```
1  #5th |
2  from gensim.downloader import load
3  import random
4  #Load the pre-trained Glove model
5  print("Loading pre-trained Glove model (50 dimensions)...")
6  model =load("glove-wiki-gigaword-50")
7  print("Model loaded successfully!")
8  #Function to construct a meaningful paragraph
9  def create_paragraph (iw, sws):
10     paragraph ="The topic of (iw) is fascinating, often linked to terms like"
11     random.shuffle (sws) # Shuffle to add variety
12     for word in sws:
13         paragraph += str(word) + ","
14     paragraph = paragraph.rstrip(", ") + "."
15     return paragraph
16 iw = "hacking"
17 sws =model.most_similar(iw, topn=5)
18 words=[word for word, s in sws]
19 paragraph =create_paragraph (iw, words)
20 print (paragraph)
```

Output :

```
Loading pre-trained Glove model (50 dimensions)...
Model loaded successfully!
The topic of (iw) is fascinating, often linked to terms likemalicious,hackers,hacker,hacked,
snooping.
```

Explanation :

Step 1: Load Pre-trained Word Vectors

```
model = load("glove-wiki-gigaword-50")
```

Loads GloVe word embeddings (50-dimensional vectors).

These vectors know how words relate in meaning (e.g., "hacking" is close to "phishing").

Step 2: Define the Main Word

```
iw = "hacking"
```

iw stands for input word or topic.

You can change it to anything like "science", "AI", etc.

Step 3: Find Related Words

```
sws = model.most_similar(iw, topn=5)
```

Finds the top 5 most similar words to "hacking", using the model's knowledge.

Step 4: Generate a Paragraph

```
def create_paragraph(iw, sws):
```

```
    paragraph = "The topic of hacking is fascinating, often linked to terms like..."
```

It builds a short paragraph using the input word and its similar words.

Uses random.shuffle to vary the order each time for uniqueness.

- 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.**

```

1 #6 |
2 !pip install transformers torch
3 from transformers import pipeline
4 sentiment_analyzer = pipeline("sentiment-analysis")
5 while True:
6     user_input = input("\nEnter a sentence (or type 'exit' to quit): ")
7     if user_input.lower() == "exit":
8         print("Goodbye!")
9         break
10    if not user_input.strip():
11        print("Please enter a non-empty sentence.")
12        continue
13    result = sentiment_analyzer(user_input)[0]
14    print(f"\nLabel: {result['label']}")
15    print(f"Confidence: {result['score']:.4f}")

```

Output :

Enter a sentence (or type 'exit' to quit): she is good girl

Label: POSITIVE

Confidence: 0.9999

Enter a sentence (or type 'exit' to quit): exit

Goodbye!

Explanation :

Step 1: Install Required Libraries

!pip install transformers torch

Downloads the Transformers library (for NLP models).

Installs PyTorch, used to run deep learning models.

Step 2: Load the Sentiment Model

from transformers import pipeline

sentiment_analyzer = pipeline("sentiment-analysis")

Loads a ready-made pipeline for sentiment analysis.

Uses pretrained BERT-like models under the hood.

Step 3: Interactive Input Loop

while True:

user_input = input("Enter a sentence:")

Asks the user to type a sentence.

Keeps running until the user types "exit".

Step 4: Analyze Sentiment

```
result = sentiment_analyzer(user_input)[0]
```

```
print(f'Label: {result['label']}')
```

```
print(f'Confidence: {result['score']:.4f}')
```

Tells whether the sentence is POSITIVE or NEGATIVE.

Shows a confidence score (between 0 and 1).

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.

```
1 #r7 |
2 from transformers import pipeline
3
4 summarizer = pipeline("summarization", model="facebook/bart-large-cnn")
5
6 text = input("Enter text to summarize:\n")
7
8 if len(text.split()) < 30:
9     print("Text is too short to summarize.")
10 else:
11     summary = summarizer(text, max_length=50, min_length=30, do_sample=False)[0]['summary_text']
12     print("\nSummary:", summary)
```

Output :

Device set to use cpu

Enter text to summarize:

Generative AI is a type of artificial intelligence that creates new content, such as text, images, music, and videos, based on patterns learned from existing data. It uses advanced algorithms and deep learning models to mimic human-like creativity and generate unique outputs. Here's a more detailed explanation: Key Features: Content Creation: Generative AI can produce various forms of content, including text, images, audio, video, and even code. Learning from Data: It learns patterns and structures from large datasets to understand and replicate existing content. Mimicking Human Creativity: Generative AI aims to generate outputs that resemble human-created content, such as writing a poem or designing a logo. Natural Language Input: Users can often provide prompts or instructions in natural language to guide the AI in generating specific content. Deep Learning Models: Generative AI relies on sophisticated machine learning models, particularly deep learning algorithms, to analyze and interpret data. Evolving Technology: Generative AI is constantly evolving as new data and algorithms are developed, leading to more sophisticated and versatile content creation. Examples of Generative AI Applications: Text

Generation: Creating articles, stories, scripts, and other written content. Image Generation: Producing realistic or stylized images, modifying existing images, or designing logos. Music Generation: Composing original musical pieces or remixing existing music. Video Creation: Generating video content, creating virtual or augmented reality experiences, or producing game assets. Code Generation: Writing or modifying

Summary: Generative AI is a type of artificial intelligence that creates new content. It uses advanced algorithms and deep learning models to mimic human-like creativity. Generative AI can produce various forms of content, including text, images, audio, video

Explanation :

Step 1: Load the Summarization Pipeline

```
from transformers import pipeline
```

```
summarizer = pipeline("summarization", model="facebook/bart-large-cnn")
```

Loads Facebook's BART model, fine-tuned for summarization.

BART is a powerful NLP model trained on large datasets.

Step 2: Take User Input

```
text = input("Enter text to summarize:\n")
```

The user types or pastes in a long paragraph or article.

Step 3: Validate Input Length

```
if len(text.split()) < 30:
```

```
    print("Text is too short to summarize.")
```

If the input has fewer than 30 words, it's too short to summarize meaningfully.

Step 4: Generate Summary

```
summary = summarizer(text, max_length=50, min_length=30,  
do_sample=False)[0]['summary_text']
```

The model generates a condensed version of the input text.

max_length and min_length control the summary size.

do_sample=False ensures deterministic output (not random).

- 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.**

```

1 #r8 |
2 !pip install langchain cohere langchain-community
3 from langchain_community.llms import Cohere
4 from langchain_community.document_loaders import TextLoader
5 from langchain.prompts import PromptTemplate
6 from langchain.chains import LLMChain
7 text = TextLoader("/content/sample text.txt").load()[0].page_content
8 prompt = PromptTemplate(
9     input_variables=["text"],
10    template="Summarize this text:\n\n{text}\n\nSummary:"
11 )
12 llm = Cohere(cohere_api_key="JuJktEEFgPDKDmzaUDfmmEaIMldTroCQcvwiUInx", temperature=0.7)
13 summary = LLMChain(llm=llm, prompt=prompt).run(text=text)
14 print("\nSummary:\n", summary)

```

Output :

Step 1: Install Required Libraries

```
!pip install langchain cohere langchain-community
```

Installs LangChain, Cohere, and community support modules.

Import Libraries

```
from langchain_community.llms import Cohere
```

```
from langchain_community.document_loaders import TextLoader
```

```
from langchain.prompts import PromptTemplate
```

```
from langchain.chains import LLMChain
```

Brings in components for:

Language model (LLM)

Document loading

Prompt formatting

Chaining LLM with input/output

Step 2: Load Your Text

```
text = TextLoader("/content/sample text.txt").load()[0].page_content
```

Loads plain text from a file.

Step 3: Extracts the raw content for processing.

Prepare the Prompt Template

```
prompt = PromptTemplate(
```

```
    input_variables=["text"],
```

```
    template="Summarize this text:\n\n{text}\n\nSummary:"
```

```
)
```


Step 4: Defines how the input is structured.

The LLM will be told: "Summarize this text..."

initialize the Cohere Language Model

```
llm = Cohere(cohere_api_key="YOUR_API_KEY", temperature=0.7)
```

Step 5: Connects to Cohere's AI.

temperature=0.7 allows slightly creative output.

Run the Chain

```
summary = LLMChain(llm=llm, prompt=prompt).run(text=text)
```

Step 6: Combines your input text and prompt.

Sends it to Cohere and retrieves the summary.

Step 7: Display Output

```
print("\nSummary:\n", summary)
```

Prints the generated summary of the file content.

- 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.**

```
1 #r9 |
2 !pip install wikipedia
3 import wikipedia, requests
4 from bs4 import BeautifulSoup
5 def fetch_info(name):
6     try:
7         p, s = wikipedia.page(name), wikipedia.summary(name, sentences=2)
8         rows = BeautifulSoup(requests.get(p.url).text, 'html.parser').select('.infobox tr')
9     except wikipedia.exceptions.DisambiguationError as e:
10         return f"Disambiguation Error: {e.options}"
11     except wikipedia.exceptions.PageError:
12         return f"No page found for '{name}'"
13     f, h = "Not Found", "Not Found"
14     for r in rows:
15         th, td = r.find("th"), r.find("td")
16         if not th or not td: continue
17         t = th.text.lower()
18         if "founded" in t or "established" in t: f = td.text.strip()
19         if "headquarters" in t or "location" in t: h = td.text.strip()
20     b = [c for c in ["New York", "London", "Tokyo", "Bangalore", "Davangere"]
21          if c.lower() in p.content.lower()] or ["Not Found"]
22     return f"Name: {name}\nFounded: {f}\nHeadquarters: {h}\nBranches: {'', '.join(b)}\nSummary: {s}"
23 print(fetch_info(input("Enter institution name: ")))
24
```

Output :

```
Requirement already satisfied: wikipedia in /usr/local/lib/python3.11/dist-packages (1.4.0)
Requirement already satisfied: BeautifulSoup4 in /usr/local/lib/python3.11/dist-packages (from wikipedia) (4.13.4)
Requirement already satisfied: requests<3.0.0,>2.0.0 in /usr/local/lib/python3.11/dist-packages (from wikipedia) (2.32.3)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests<3.0.0,>2.0.0->wikipedia) (3.4.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests<3.0.0,>2.0.0->wikipedia) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests<3.0.0,>2.0.0->wikipedia) (2.4.0)
Requirement already satisfied: certifi>2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests<3.0.0,>2.0.0->wikipedia) (2025.4.26)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.11/dist-packages (from BeautifulSoup4->wikipedia) (2.7)
Requirement already satisfied: typing-extensions>4.0.0 in /usr/local/lib/python3.11/dist-packages (from BeautifulSoup4->wikipedia) (4.13.2)
Enter institution name: jain institute of technology
Name: jain institute of technology
Founded: 2011
Headquarters: Davangere, Karnataka, India
Branches: Davangere
Summary: In the academic year 2011-2012, a new engineering college was started in Davangere called Jain Institute of Technology. It is affiliated to Visvesvaraya Technological University and approved by the All India Council for Technical Education.
```

Explanation :

Step 1: Install and Import Required Libraries

`!pip install wikipedia`

`import wikipedia, requests`

`from bs4 import BeautifulSoup`

wikipedia: For searching and retrieving summaries and URLs.

requests: To fetch HTML content of Wikipedia pages.

BeautifulSoup: For parsing HTML and extracting data from the infobox.

Step 2: Define Function: `fetch_info(name)`

`def fetch_info(name):`

Takes the institution's name as input and gathers structured info.

Step 3: Get Wikipedia Page + Summary

`p, s = wikipedia.page(name), wikipedia.summary(name, sentences=2)`

p: Full page object (to get URL and content).

s: 2-sentence summary.

Step 4: Scrape the Infobox

`rows = BeautifulSoup(requests.get(p.url).text, 'html.parser').select('.infobox tr')`

Parses the page's HTML.

Selects rows (`<tr>`) inside the infobox (side table).

Step 5: Search for Key Info

`if "founded" in t or "established" in t: f = td.text.strip()`

`if "headquarters" in t or "location" in t: h = td.text.strip()`

Looks for keywords to extract:

Founded/Established

Headquarters/Location

Step 6: Optional: Look for Known Branches

`b = [c for c in ["New York", "London", "Tokyo", "Bangalore", "Davangere"]`

```
if c.lower() in p.content.lower()]
```

Scans the full page content for these known cities.

You can add more cities here if needed.

Step 7: Handle Errors Gracefully

```
except wikipedia.exceptions.DisambiguationError...
```

```
except wikipedia.exceptions.PageError...
```

Handles pages that are ambiguous or not found.

Step 8: Final Output

```
return f'Name: {name}\nFounded: {f}\nHeadquarters: {h}\nBranches: ...\nSummary: ...'
```

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.

```
1 #r10 |
2 !pip install pymupdf
3 # Install faiss-cpu explicitly
4 !pip install faiss-cpu
5 import fitz, faiss, numpy as np
6 from sentence_transformers import SentenceTransformer
7 from sklearn.metrics.pairwise import cosine_similarity
8
9 text = "".join(p.get_text() for p in fitz.open("ipc.pdf"))
10 chunks = [text[i:i+1000] for i in range(0, len(text), 1000)]
11
12 model = SentenceTransformer("all-MiniLM-L6-v2")
13 emb = model.encode(chunks, convert_to_numpy=True).astype("float32")
14
15 idx = faiss.IndexFlatL2(emb.shape[1])
16 idx.add(emb)
17
18 print("✅ IPC chatbot ready. Type 'bye' to exit.")
19 while True:
20     q = input("You: ")
21     if q.lower() == "bye":
22         print("Bot: Goodbye!"); break
23     qe = model.encode([q], convert_to_numpy=True).astype("float32")
24     D, I = idx.search(qe, 1)
25     sim = cosine_similarity(qe, [emb[I[0][0]]])[0][0]
26     print("Bot:", chunks[I[0][0]].strip() if sim > 0.6 else "No relevant info found.")
```

Output :

```

Requirement already satisfied: faiss-cpu in /usr/local/lib/python3.11/dist-packages (1.11.0)
Requirement already satisfied: numpy<3.0,>=1.25.0 in /usr/local/lib/python3.11/dist-packages (from faiss-cpu) (2.0.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from faiss-cpu) (24.2)
Requirement already satisfied: pymupdf in /usr/local/lib/python3.11/dist-packages (1.25.5)
Requirement already satisfied: faiss-cpu in /usr/local/lib/python3.11/dist-packages (from faiss-cpu) (1.11.0)
Requirement already satisfied: numpy<3.0,>=1.25.0 in /usr/local/lib/python3.11/dist-packages (from faiss-cpu) (2.0.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from faiss-cpu) (24.2)
modules.json: 100% 349/349 [00:00<00:00, 8.07kB/s]
config_sentence_transformers.json: 100% 116/116 [00:00<00:00, 1.60kB/s]
README.md: 100% 10.5k/10.5k [00:00<00:00, 263kB/s]
sentence_bert_config.json: 100% 53.0/53.0 [00:00<00:00, 1.26kB/s]
config.json: 100% 612/612 [00:00<00:00, 11.4kB/s]
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better performance, install the package with: 'pip install huggingface_hub[hf_xet]' or 'pip install hf_xet'
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better performance, install the package with: 'pip install huggingface_hub[hf_xet]' or 'pip install hf_xet'
model.safetensors: 100% 90.9M/90.9M [00:00<00:00, 143MB/s]
tokenizer_config.json: 100% 350/350 [00:00<00:00, 25.3kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 1.29MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 1.30MB/s]
special_tokens_map.json: 100% 112/112 [00:00<00:00, 5.58kB/s]
config.json: 100% 190/190 [00:00<00:00, 16.6kB/s]
[+] IPC chatbot ready. Type 'bye' to exit.
You: what is the punishment for theft?
Bot: of having committed, or attempted to commit, an offence punishable with death or with [imprisonment for life], or with imprisonment for a term which may extend to ten years, shall be punished with imprisonment of either description for a term which may extend to ten years, and shall also be liable to fine; and, if the offence be punishable under section 377 of this Code, may be punished with [imprisonment for life].
Of Robbery and dacoity
398. Robbery.-In all robbery there is either theft or extortion.
1. Subs. by Act 26 of 1955, s. 117 and the Sch., for "transportation for life" (w.e.f. 1-1-1956).
98
When theft is robbery.-Theft is "robbery" if, in order to the committing of the theft, or in committing the theft, or in carrying away or attempting to carry away property obtained by the theft, the offender, for that end voluntarily causes or attempts to cause to any person death or hurt or wrongful restraint, or
You: bye
Bot: Goodbye!

```

Step 1: Install Required Libraries

`!pip install pymupdf faiss-cpu`

pymupdf: To extract text from PDF files (fitz module).

faiss-cpu: Fast similarity search library.

sentence-transformers: For converting text to embeddings.

Step 2: Read PDF with PyMuPDF

`import fitz`

`text = "".join(p.get_text() for p in fitz.open("ipc.pdf"))`

Opens the IPC PDF and extracts all text.

Step 3: Split Text into Chunks

`chunks = [text[i:i+1000] for i in range(0, len(text), 1000)]`

Chunks the full text into pieces of 1000 characters (to maintain context during search).

Step 4: Convert Text to Vectors

`from sentence_transformers import SentenceTransformer`

`model = SentenceTransformer("all-MiniLM-L6-v2")`

`emb = model.encode(chunks, convert_to_numpy=True).astype("float32")`

Each chunk is embedded (converted into numerical vectors) using a pre-trained model.

These vectors capture semantic meaning.

Step 5: Create FAISS Index

`import faiss`

`idx = faiss.IndexFlatL2(emb.shape[1])`

`idx.add(emb)`

FAISS stores these embeddings to allow fast similarity-based search.

Step 6: Interactive Chat Loop

while True:

 q = input("You: ")

Continuously waits for user queries.

Step 7: Search for Best-Matching Chunk

qe = model.encode([q], convert_to_numpy=True).astype("float32")

D, I = idx.search(qe, 1)

Embeds the user's question and searches the closest match in the FAISS index.

Step 8: Check Semantic Similarity

from sklearn.metrics.pairwise import cosine_similarity

sim = cosine_similarity(qe, [emb[I[0][0]]])[0][0]

Ensures the matched text is relevant enough (cosine similarity > 0.6).

Step 9: Return the Answer

print("Bot:", chunks[I[0][0]].strip() if sim > 0.6 else "No relevant info found.")

If similarity is good, returns the most relevant IPC passage.