



**Maharaja Education Trust (R), Mysuru**

**MAHARAJA INSTITUTE OF TECHNOLOGY MYSORE**

An Autonomous Institute affiliated to Visvesvaraya Technological University, Belagavi

Department of Computer Science and Engineering



**LAB Manual**  
**For**  
**Generative AI Lab**  
**(BAIL657C)**  
**6<sup>th</sup> Semester**

**Prepared by,**

**Prakruthi S**

**Assistant Professor**

**Dept. of Computer Science and Engineering**

**MIT Mysore.**

## 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.

## Experiments 1

**LAB 1: Explore pre-trained word vectors. Explore word relationships using vector arithmetic. Perform arithmetic operations and analyze results.**

### *# Step 1: Import Required Libraries*

```
import os
import gensim.downloader as api
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from gensim.models import KeyedVectors
from google.colab import drive
```

### *# Step 2: Mount Google Drive*

```
drive.mount('/content/drive')
```

### *# Define model path in Google Drive*

```
model_path = "/content/drive/My Drive/word2vec-google-news-300.model"
```

### *# Step 3: Load or Download the Word2Vec Model*

```
if os.path.exists(model_path):
    print("Model found in Google Drive..Loading")
    word_vectors = KeyedVectors.load(model_path)
else:
    print("Model not found. Downloading Word2Vec model...")
    word_vectors = api.load("word2vec-google-news-300")
    print("Saving model to Google Drive for future use...")
    word_vectors.save(model_path)
    print("Model saved successfully")
    print("\nModel Loaded Successfully\n")
```

### *# Step 4: Find Similar Words*

```
print("Top 5 words similar to 'computer':")
```

```
similar_words = word_vectors.most_similar("computer", topn=5)
for word, similarity in similar_words:
    print(f"{word}: {similarity:.4f}")
```

### ***# Step 5: Word Vector Arithmetic***

```
print("\nPerforming Vector Arithmetic: 'king - man + woman'")
# king - man + woman = ?
result = word_vectors.most_similar(positive=['king', 'woman'], negative=['man'], topn=1)
print(f"Result: {result[0][0]}") # Expected output: 'queen'
```

### ***# Step 6: More Arithmetic Operations***

```
print("\n More Examples of Vector Arithmetic:")
examples = [
    ("Paris", "France", "Italy"),
    ("Einstein", "scientist", "painter")
]
for w1, w2, w3 in examples:
    result = word_vectors.most_similar(positive=[w1, w3], negative=[w2], topn=1)
    print(f"{w1} - {w2} + {w3} = {result[0][0]}")
```

## **OUTPUT**

Mounted at /content/drive

Model found in Google Drive! Loading...

Model Loaded Successfully

Top 5 words similar to 'computer':

*computers: 0.7979*

*laptop: 0.6640*

*laptop\_computer: 0.6549*

*Computer: 0.6473*

*com\_puter: 0.6082*

Performing Vector Arithmetic: 'king - man + woman'

Result: queen

### **More Examples of Vector Arithmetic:**

Paris - France + Italy = Milan

Einstein - scientist + painter = Picasso

## Experiments 2

**Use dimensionality reduction (e.g., PCA or t-SNE) to visualize word embeddings for Q1. 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.**

```
import os
import gensim.downloader as api
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from gensim.models import KeyedVectors
from sklearn.metrics.pairwise import cosine_similarity
from gensim.models import KeyedVectors
from google.colab import drive

# Step 2: Mount Google Drive
drive.mount('/content/drive')

# Define model path in Google Drive
model_path = "/content/drive/My Drive/word2vec-google-news-300.model"

# Step 3: Load or Download the Word2Vec Model
if os.path.exists(model_path):
    print("Model found in Google Drive..Loading")
    word_vectors = KeyedVectors.load(model_path)
else:
    print("Model not found. Downloading Word2Vec model...")
    word_vectors = api.load("word2vec-google-news-300")
    print("Saving model to Google Drive for future use...")
    word_vectors.save(model_path)
    print("Model saved successfully")
```

```
print("\nModel Loaded Successfully\n")
word2vec_model = KeyedVectors.load('/content/drive/My Drive/word2vec-google-news-300.model')
```

**#Step 4: Get embeddings for the selected words**

```
words = ["investment", "stocks", "bonds", "bank", "loan", "interest", "equity", "mortgage", "risk", "dividend"]
```

```
word_vectors = np.array([word2vec_model[word] for word in words])
```

```
# Step 5 Apply dimensionality reduction (PCA and t-SNE)
```

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

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

```
plt.figure(figsize=(10, 7))
```

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

```
plt.scatter(reduced_vectors[i, 0], reduced_vectors[i, 1])
```

```
plt.text(reduced_vectors[i, 0] + 0.01, reduced_vectors[i, 1] + 0.01, word, fontsize=12)
```

```
plt.title('Word Embeddings Visualized using PCA')
```

```
plt.xlabel('PCA Component 1')
```

```
plt.ylabel('PCA Component 2')
```

```
plt.grid(True)
```

```
plt.show()
```

**# Step 6 : Find 5 semantically similar words for a given input word**

```
def find_similar_words(input_word, model, top_n=5):
```

```
    similar_words = model.most_similar(input_word, topn=top_n)
```

```
    return [word for word, similarity in similar_words]
```

**# Test the function with "bank"**

```
similar_words_bank = find_similar_words('bank', word2vec_model)
```

```
print("5 Semantically Similar Words to 'bank':", similar_words_bank)
```



## Experiment 3

Train a custom Word2Vec model on a small dataset. Train embedding's on a domain-specific corpus (e.g., legal, medical) and analyse how embedding's capture domain-specific semantics.

### Introduction

Word embeddings are numerical vector representations of words that capture their semantic meaning. **Word2Vec**, introduced by Mikolov et al., is a popular algorithm for learning such embeddings. It is used in **Natural Language Processing (NLP)** to represent words in a continuous vector space, where similar words have similar representations.

In this experiment, we train a **custom Word2Vec model** on a **medical corpus** to generate meaningful word embeddings. To improve the quality of embeddings, **bigrams (multi-word phrases)** are detected and incorporated into the model. The trained model is then analyzed by checking similar words and visualizing word relationships using **Principal Component Analysis (PCA)**.

### Objectives

1. **Train a Word2Vec model** on a **medical domain-specific** text corpus.
2. **Remove stopwords** to improve the quality of embeddings.
3. **Detect bigrams (word pairs)** such as "blood sugar" and "high pressure", ensuring better representation.
4. **Analyze the trained embeddings** by finding words similar to "diabetes" and "hypertension".

In Natural Language Processing (NLP), a corpus (plural: corpora) refers to a large collection of text data that is used for training machine learning models, especially for language-based tasks like text classification, machine translation, and word embeddings.

### *Corpus in the Word2Vec Experiment*

A corpus **in this context means** a set of sentences related to a specific domain **that is used to** train the Word2Vec model. **It provides context for words so that the model can learn their meanings and relationships.**

## Prerequisites and Key Concepts

To understand this experiment fully, familiarity with the following concepts is beneficial:

### 1. Word Representation in NLP

One-hot encoding vs. dense embeddings

Need for vectorization of text data

### 2. Word2Vec Model

**CBOW (Continuous Bag of Words):** Predicts a target word from its context.

**Skip-gram Model:** Predicts surrounding words for a given target word.

### 3. Text Preprocessing Techniques

Tokenization (splitting sentences into words)

Stopword removal (removing common words like "the", "is")

Lemmatization (converting words to their root forms)

By the end of this experiment, students will develop an intuitive understanding of how word embeddings capture domain-specific relationships.

## *# Experiment 3: Word2Vec Training on Medical Corpus with Bigram Detection*

### *# Import libraries*

```
from gensim.models import Word2Vec
from gensim.models.phrases import Phrases, Phraser
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
```

### *# Read medical corpus from an external file*

```
with open("medical_corpus.txt", "r") as file:
    corpus = [line.strip() for line in file if line.strip()]
```

### *# Refined stopword list*

```
stopwords = {"the", "a", "is", "for", "with", "to", "of", "and", "in", "can", "are"}
```

```
# Tokenization with stopwords removal
```

```
tokenized_sentences = [
```

```
[word for word in sentence.lower().split() if word not in stopwords]
for sentence in corpus
]
```

***# Detect bigrams to capture word combinations (e.g., "blood sugar", "high pressure")***

```
bigram = Phrases(tokenized_sentences, min_count=2, threshold=5)
bigram_phraser = Phraser(bigram)
tokenized_sentences = [bigram_phraser[sentence] for sentence in tokenized_sentences]
```

***# Train Word2Vec model with improved parameters***

```
model = Word2Vec(tokenized_sentences, vector_size=150, window=5, min_count=2,
epochs=300, sg=1, hs=1, negative=0)
```

***# Display similar words***

```
diabetes_similar = [(word, round(sim, 2)) for word, sim in
model.wv.most_similar("diabetes", topn=5)]
print("Words similar to 'diabetes':", diabetes_similar)
hypertension_similar = [(word, round(sim, 2)) for word, sim in
model.wv.most_similar("hypertension", topn=5)]
print("Words similar to 'hypertension':", hypertension_similar)
```

### **Output:**

Words similar to 'diabetes': [('lifestyle', 0.12), ('disease.', 0.11), ('diet', 0.08), ('sugar', 0.07), ('diabetes.', 0.06)]

Words similar to 'hypertension': [('disease.', 0.09), ('risk', 0.05), ('high', 0.03), ('diet', 0.01), ('blood', 0.01)]

### Experiment 4

**Use Word Embedding's to improve prompts for Generative AI model.**

- (i) **Retrieve similar words using word embedding's.**
- (ii) **Use the similar words to enrich a GenAI prompt.**
- (iii) **Use the AI model to generate responses for the original and enriched prompts.**
- (iv) **Compare the outputs in terms of detail and relevance.**

**Step 1: Installs necessary libraries, imports modules, and sets up Google Colab environment to work with word embedding's using Genism.**

```
%pip install numpy
%pip install scipy
%pip install gensim
import os
import gensim.downloader as api
from gensim.models import KeyedVectors
from google.colab import drive
```

**Step 2: Connect your Google Drive to Google Colab, allowing you to access, save, and load files from your Drive directly in the Colab environment.**

```
drive.mount('/content/drive')
```

**Step 3: Check if Word2Vec model is already saved in Google Drive, load it if found, or downloads and save it if not available.**

```
model_path = "/content/drive/My Drive/word2vec-google-news-300.model"
if os.path.exists(model_path):
    print("Model found in Google Drive..Loading")
    word_vectors = KeyedVectors.load(model_path)
    print("Loading Completed")
else:
    print("Model not found. Downloading Word2Vec model...")
    word_vectors = api.load("word2vec-google-news-300")
    print("Saving model to Google Drive for future use...")
```

```
word_vectors.save(model_path)
print("Model saved successfully")
print("\nModel Loaded Successfully\n")
```

**Step 4: Retrieve words that are most similar to "king" based on the pre-trained Word2Vec model. It measures similarity using cosine similarity between word vectors.**

```
print(word_vectors.most_similar("king"))
```

**[Output:** *[('kings', 0.7138045430183411), ('queen', 0.6510956883430481), ('monarch', 0.6413194537162781), ('crown\_prince', 0.6204220056533813), ('prince', 0.6159993410110474), ('sultan', 0.5864824056625366), ('ruler', 0.5797567367553711), ('princes', 0.5646552443504333), ('Prince\_Paras', 0.5432944297790527), ('throne', 0.5422105193138123)]]*

**Step 5: Take user input to create a structured prompt for an AI system.**

**1. The original prompt (a question or instruction for AI).**

**2. Key terms (words that help refine the AI response).**

```
original_prompt = input("Enter the original prompt: ")
```

*# Get key terms from user (comma-separated)*

```
key_terms_input = input("Enter key terms (comma-separated): ")
```

```
key_terms = [term.strip() for term in key_terms_input.split(",")]
```

**Output:**

*Enter the original prompt: Importance of engineering*

*Enter key terms (comma-separated): Job, career*

**Step 6: Enrich the given user prompt by finding semantically similar words for key terms using pre-trained word embedding's (Word2Vec).**

```
similar_terms = []
```

```
for term in key_terms:
```

```
    if term in word_vectors.key_to_index:
```

```
        similar_terms.extend({word for word, _ in word_vectors.most_similar(term, topn=2)})
```

```
if similar_terms:
    enriched_prompt = f"{original_prompt} Consider aspects like: {' '.join(similar_terms)}."
else:
    enriched_prompt = original_prompt
print("Original Prompt:", original_prompt)
print("Enriched Prompt:", enriched_prompt)
```

**Output:**

*Original Prompt: Importance of engineering.*

*Enriched Prompt: Importance of engineering Consider aspects like: Employment, job, career.*

**Step 7: Prompt the user to enter the API key (API key of Google AI used here), stores it in an environment variable, and assigns it to a Python variable for later use.**

**Note: Create the Google API using the link, <https://aistudio.google.com/>**

```
import getpass
import os
GOOGLE_API_KEY=os.environ["GOOGLE_API_KEY"]=getpass.getpass("Enter your
Google AI API key: ")
```

**Step 8: Install the necessary LangChain libraries and set up ChatGoogleGenerativeAI to interact with Google's Gemini-2.0 Flash model for text generation. Import the required module, initialize the model with key parameters like temperature (0.3 for balanced responses), max tokens (512 for response length), timeout (30 seconds to prevent hanging requests), and max retries (2 for handling failures).**

**The api\_key ensures secure access to Google's AI services, enabling efficient and controlled AI-powered text generation.**

```
%pip install langchain-google-genai
%pip install langchain-core
%pip install langchain-community
%pip install -qU langchain-google-genai
```

```
%pip install --upgrade langchain
from langchain_google_genai import ChatGoogleGenerativeAI
llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash-exp",
    temperature=0.3,
    api_key=GOOGLE_API_KEY,
    max_tokens=512,
    timeout=30,
    max_retries=2,
)
```

### **Step 8: Invoking the AI Model for a Response**

```
llm.invoke("Hi")
```

The command `llm.invoke("Hi")` sends the input "Hi" to the Google Gemini AI model via LangChain and returns the model's response. Since `llm` is an instance of `ChatGoogleGenerativeAI`, this method invokes the AI model, processes the input, and generates a response based on the configured parameters (e.g., temperature, max tokens, timeout). The output will typically be a greeting or a conversational response like

**"Hi! How can I assist you today?"**

### **Step 9: Send the original prompt to the Google Gemini AI model (Large Language Model)**

```
print(llm.invoke(original_prompt).content)
```

This command **sends the** `original_prompt` to the Google Gemini AI model (`llm`), **retrieves the response**, extracts the **text content**, and prints it.

**Output:** Engineering is incredibly important to modern society, playing a vital role in shaping our world and improving our lives in countless ways.

Here's a breakdown of its importance:

### **Experiment 5:**

**Use word embedding's to create meaningful sentences for creative tasks.**

- (i) Retrieve similar words for a seed word.**
- (ii) Create a sentence or story using these words as a starting point. Write a program that: (a) Takes a seed word. (b) Generates similar words. (c) Constructs a short paragraph using these words.**

**AIM:** To build a user friendly application that takes a seed word as input, retrieves semantically or creatively similar words using Googles Gemini LLM via LangChain, and generates creative paragraph variations based on those words.

### **Introduction:**

#### **Large Language Models (LLMs)**

LLMs are advanced AI systems trained on vast amounts of text data to generate human like text. They understand context, semantics, and syntax, allowing them to write essays, generate code, answer questions, and even compose poetry.

#### **Gemini's Flash Model**

Google's Gemini Flash is a lightweight, fast, and efficient version of the Gemini LLM family. It is optimized for quick response times and is ideal for real time applications such as chatbots, text generation, and summarization. In this lab, we use gemini-2.0-flash-exp, an experimental fast-response model.

#### **LangChain Library**

LangChain is a framework that simplifies the development of applications powered by language models. It provides tools to connect to different LLMs (like OpenAI, Cohere, Gemini, etc.), manage conversations, chain together prompts, and integrate memory, tools, and agents. We use LangChain to send prompts to Gemini and receive structured responses.

#### **Gradio Web Interface**

Gradio is an open-source Python library that allows us to quickly create web apps to showcase machine learning models or functions. It provides components like text boxes, buttons, and file downloaders so users can interact with the model in a browser with no coding required.



Large Language Models (LLMs) such as Google's Gemini can generate human-like responses and creative content. In this lab, we use the LangChain library to interact with Gemini's Flash model and create a simple Gradio web interface for creative paragraph generation. The application takes a single word (the seed word) and builds imaginative content based on related terms.

### **Objectives of the Program**

1. Understand how to integrate and interact with Google's Gemini LLM in a Python application using the LangChain framework.
2. Retrieve semantically or creatively related words for a given seed word using the Gemini language model.
3. Generate multiple creative paragraph variations that meaningfully use the seed word and its related words.
4. Design and implement a user friendly browser based interface using the Gradio library to showcase the application.

### **Step 1: Install Required Libraries**

```
!pip install -q langchain-google-genai gradio
```

### **Step 2: Imports, Configure and Gemini LLM setup (Initialize Gemini LLM)**

#### **# Imports and Gemini LLM setup**

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.messages import HumanMessage
import gradio as gr
import getpass
import io
```

#### **# Get Google API key securely**

```
GOOGLE_API_KEY = getpass.getpass("Enter your Google API key: ")
```

#### **# Initialize Gemini 2.0 LLM**

```
llm = ChatGoogleGenerativeAI(
```

```
model="gemini-2.0-flash-exp",
temperature=0.8,
api_key=GOOGLE_API_KEY,
max_tokens=512,
timeout=30,
max_retries=2,
)
print("Gemini LLM is ready.")
```

Output: Enter your Google API key: .....

Gemini LLM is ready.

### **Step 3: Retrieve Similar Words from Gemini LLM for the Given Seed Words**

#### **# Get similar words from Gemini**

```
def get_similar_words(seed):
    prompt =(
        f"Give me 5 English words that are semantically or creatively similar to '{seed}'. "
        f"Return the words as a comma separated list without numbers or explanations."
    )
    response = llm.invoke([HumanMessage(content=prompt)])
    return [word.strip() for word in response.content.split(',') if word.strip()]
```

### **Step 4: Generate creative paragraphs using seed words**

#### **def create\_paragraph(seed, words):**

```
word_list = ', '.join(words)
prompt = (
    f"Write a short, creative paragraph using the words '{seed}' and the following related words: "
    f"{word_list}. The paragraph should be imaginative and meaningful."
)
response = llm.invoke([HumanMessage(content=prompt)])
return response.content
```

**Step 5: Generate and Format Paragraph Variations Using the Seed Words and Similar Words**  
**def generate\_paragraphs(seed\_word):**

```
try:
    seed_word = seed_word.strip()
    if not seed_word:
        return "Please enter a valid seed word."
    similar_words = get_similar_words(seed_word)
    if len(similar_words) < 3:
        return "Could not find similar words. Try a different seed word."
    output_text = f"Seed Word: {seed_word}\nSimilar Words: {' '.join(similar_words)}\n\n"
    for i in range(1, 4):
        paragraph=create_paragraph(seed_word, similar_words) or f"(Variation{i})Could not generate paragraph."
        output_text += f"--- Variation {i} ---\n{paragraph.strip()}\n\n"
    return output_text
except Exception as e:
    return f"Error: {str(e)}"
```

**Step 6: Build and Launch the Gradio User Interface for Paragraph Generation**

```
gr.Interface(
    fn=generate_paragraphs,
    inputs=gr.Textbox(label="Enter a Seed Word"),
    outputs=gr.Markdown(label="Generated Paragraphs"),
    title="Creative Writer",
    description="Enter a seed word. This app will find similar words using Gemini and generate 3 creative paragraph variations.",
    theme="default",
).launch(debug=False)
```

## 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.

**AIM:** To use a pre-trained Hugging Face model to analyze sentiment in text by loading the sentiment analysis pipeline and providing real-world input sentences.

### Introduction:

- Sentiment analysis is one of the most common applications of **Natural Language Processing (NLP)**, which helps machines understand and interpret human emotions from written text. It is widely used in real-world applications such as analyzing customer reviews, monitoring opinions on social media, and gaining insights into user satisfaction. For example, companies use sentiment analysis to understand how customers feel about their products or services.
- In this lab, we use the Hugging Face Transformers library, an open-source toolkit that provides access to thousands of pre-trained language models, including BERT, GPT, and DistilBERT. These models have already been trained on massive amounts of text data, so we can use them without training from scratch.
- To perform sentiment analysis, we need to convert raw text into a format that the model can understand. This process involves:

**Tokenization:** Breaking a sentence into smaller pieces called tokens (words or subwords).

**Tokenizer:** A tool that converts tokens into numerical IDs that the model can process.

**Model:** A pre-trained deep learning model that understands the structure and meaning of language and classifies the sentiment.

**Pipeline:** A simplified interface from Hugging Face that bundles the tokenizer and model into one easy-to-use function for a specific task like sentiment

analysis. **Pipeline** bundles together preprocessing, model inference, and output formatting.

In this experiment, we use the *distilbert-base-uncased-finetuned-sst-2-english* model, which is a lighter and faster version of BERT. It has been fine-tuned specifically for sentiment classification using the Stanford Sentiment Treebank (SST-2) dataset. We also use **pandas**, a powerful Python library for displaying data in a structured table format. This lab introduces key AI/ML concepts while showing how easy it is to apply state-of-the-art NLP with just a few lines of code using Hugging Face.

Sentiment analysis is a Natural Language Processing (NLP) technique used to determine whether a piece of text is positive, negative, or neutral. It is widely used in real-world applications such as product reviews, social media monitoring, and customer feedback analysis.

In this experiment, we use the Hugging Face transformers library, which provides easy access to powerful pre-trained models for various NLP tasks. One of the simplest ways to perform sentiment analysis is by using the pipeline interface, which abstracts away the complexities of tokenization and model handling.

### Key Concepts Introduced:

**Hugging Face Transformers:** An open-source library offering thousands of pre-trained models for NLP tasks.

**Pre-trained Model:** A machine learning model that has already been trained on a large dataset and can be reused.

**DistilBERT:** A smaller, faster version of BERT that retains most of its performance. We use the `'distilbert-base-uncased-finetuned-sst-2-english'` model, fine-tuned on sentiment classification.

By using the Hugging Face pipeline, we simplify the steps of loading the model, preparing the input, and interpreting the result—all done in a few lines of code.

### Confidence Score in Sentiment Analysis.

In sentiment analysis using machine learning models, the confidence score indicates how sure the model is about its prediction. For example, if a review is classified as POSITIVE with a

confidence score of 0.98, it means: The model is 98% confident that this sentence expresses positive sentiment.

### **Objectives of the Program**

1. To understand and implement a pre-trained sentiment analysis model using Hugging Face.
2. To process and classify text into positive or negative sentiment.
3. To create a real-world dataset of customer reviews.
4. To display sentiment results in a structured table using pandas.
5. To perform overall sentiment aggregation and derive recommendation insights.

### **Step 1: Mount Google Drive to store the downloaded model**

**This step connects your Google Drive to Colab so you can store the model permanently and reuse it later.**

```
from google.colab import drive
drive.mount('/content/drive')
```

### **Step 2: Create a directory in Google Drive to cache Hugging Face models**

**This sets up a folder in your Google Drive to store the model files.**

```
import os
cache_dir = "/content/drive/MyDrive/transformers_cache"
os.makedirs(cache_dir, exist_ok=True)
os.environ['TRANSFORMERS_CACHE'] = cache_dir
```

### **Step 3: Install Hugging Face Transformers and Pandas**

**This installs the necessary libraries: transformers (for the model) and pandas (for table display).**

```
!pip install transformers pandas --quiet
```

### **Step 4: Import Python libraries for NLP and data handling**

**This imports the Python libraries needed for model handling and data display.**

```
import pandas as pd
from transformers import pipeline, AutoTokenizer, AutoModelForSequenceClassification
```

### **Step 5: Load pre-trained model and tokenizer from Hugging Face**

```
model_name = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(model_name, cache_dir=cache_dir)
model      =      AutoModelForSequenceClassification.from_pretrained(model_name,
cache_dir=cache_dir)
```

### **# Create the sentiment analysis pipeline**

```
sentiment_pipeline = pipeline(
    "sentiment-analysis",
    model=model,
    tokenizer=tokenizer
)
```

### **Step 6: Create a list of example product reviews (real-world application)**

```
sample_reviews = [
    "I absolutely loved this product, it exceeded my expectations!",
    "Great experience, the product quality and delivery were excellent.",
    "Highly recommended! I'm very happy with the purchase.",
    "The design is sleek and the features work perfectly.",
    "Terrible experience. The product stopped working in two days.",
    "Not worth the money — very disappointed with the quality."
]
```




### **Step 7: Analyze sentiment of each review using the pipeline**

```
sentiment_results = sentiment_pipeline(sample_reviews)
```

Step 8: Show results in a table with sentiment and confidence score

```
df_results = pd.DataFrame({
    "Review": sample_reviews,
    "Sentiment": [result["label"] for result in sentiment_results],
    "Confidence Score": [result["score"] for result in sentiment_results]
})
df_results
```

**Output:**

	Review	Sentiment	Confidence Score	
0	I absolutely loved this product, it exceeded m...	POSITIVE	0.999877	
1	Great experience — the product quality and del...	POSITIVE	0.999864	
2	Highly recommended! I'm very happy with the pu...	POSITIVE	0.999876	
3	The design is sleek and the features work perf...	POSITIVE	0.999879	
4	Terrible experience. The product stopped worki...	NEGATIVE	0.999792	
5	Not worth the money — very disappointed with t...	NEGATIVE	0.999807	

**Step 9: Count how many reviews are positive and negative**

```

num_positive = sum(1 for res in sentiment_results if res["label"] == "POSITIVE")
num_negative = sum(1 for res in sentiment_results if res["label"] == "NEGATIVE")
total_reviews = len(sentiment_results)
positive_percentage = (num_positive / total_reviews) * 100
negative_percentage = (num_negative / total_reviews) * 100

```

**# Determine overall sentiment**

```

if num_positive > num_negative:
    overall_sentiment = "Positive"
    recommendation = "We recommend this product based on the positive reviews."
elif num_negative > num_positive:
    overall_sentiment = "Negative"
    recommendation = "We do not recommend this product based on the negative reviews."
else:
    overall_sentiment = "Mixed"
    recommendation = "The reviews are mixed. Consider additional factors before deciding."

```

**Step 10: Print summary of the analysis**

```

print("\n--- Overall Analysis ---")
print(f"Total Reviews Analyzed: {total_reviews}")
print(f"Positive Reviews: {num_positive} ({positive_percentage:.1f}%)")
print(f"Negative Reviews: {num_negative} ({negative_percentage:.1f}%)")

```



```
print(f"Overall Sentiment: {overall_sentiment}")  
print(f"Recommendation: {recommendation}")
```

### **Output:**

--- Overall Analysis ---

Total Reviews Analyzed: 6

Positive Reviews: 4 (66.7%)

Negative Reviews: 2 (33.3%)

Overall Sentiment: Positive

Recommendation: We recommend this product based on the positive reviews.

## 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.

AIM: To summarize long text using a pre-trained Hugging Face model by loading the summarization pipeline, providing a passage as input, and generating a concise output.

### Introduction:

Text summarization is a key task in Natural Language Processing (NLP) that condenses a large block of text into a short, meaningful summary. It is especially useful for applications like news summarization, document previews, and information retrieval. In this lab, we explore summarization using an abstractive model, which generates new phrases and sentences rather than merely extracting existing ones.

The model used is facebook/bart-large-cnn, an advanced transformer-based sequence-to-sequence model developed by Facebook AI. It excels at summarization by encoding the input text and decoding it into a shorter, human-like version.

We use the transformers library provided by Hugging Face, a powerful open-source platform that simplifies the use of state-of-the-art machine learning models. It offers a high-level API called pipeline() that abstracts the complexity of loading pre-trained models, managing tokenization, and performing inference, all in a few lines of code. A public online repository of pre-trained model where we can browse, search, and test models can be found in URL: <https://huggingface.co/models>

### Using Hugging Face offers several benefits:

Access to Pre-trained Models:

Hugging Face provides thousands of models that are already trained on large datasets. This saves time and resources—you don't have to train models from scratch.

Easy-to-use Pipeline API:

With just one line of code (e.g., pipeline("summarization")), you can load a full NLP system including the model and tokenizer. This simplifies development and reduces coding effort.

High Accuracy with Less Effort:

The models are trained by top research labs (like Facebook AI, Google, etc.), meaning they are well-optimized and deliver strong performance out-of-the-box.

Supports Multiple Languages and Tasks:

Whether you're summarizing English text, analyzing Hindi tweets, or translating French paragraphs—Hugging Face has models for it.

Integration with Google Colab and Google Drive:

You can easily run these models in the cloud (Colab), and even store them in Google Drive for reusability.

Community and Documentation:

Hugging Face has a large community of contributors and excellent documentation, making it easier for beginners and researchers to learn and experiment.

### **Key Concepts Introduced:**

1. Text Summarization: The process of shortening a long document into its essential meaning.
2. Abstractive Summarization: Generating new sentences rather than extracting directly.
3. Hugging Face Transformers: An open-source Python library with access to thousands of pre-trained NLP models.
4. BART: Facebook's encoder-decoder transformer for text generation and summarization.
5. Google Colab + Google Drive: Cloud-based coding with persistent model storage.

Objectives of the Program

1. To understand abstractive summarization using a pre-trained model.
2. To use the Hugging Face pipeline for summarization.
3. To read long text content from a file.
4. To generate and display the summarized version of the input.
5. To store the model in Google Drive and avoid repeated downloads.

### **Step 1: Install Required Libraries**

```
!pip install transformers sentencepiece --quiet
```

### **Step 2: Import libraries**

```
from transformers import pipeline
from google.colab import files, drive
import os
```

### Step 3: Mount Google Drive

```
drive.mount('/content/drive')
```

**Output: Mounted at /content/drive**

### Step 4: Define model save path inside Google Drive

```
model_dir = "/content/drive/MyDrive/bart_summarizer"
```

### Step 5: Load or download the model

```
if os.path.exists(model_dir):  
    print("Loading model from Google Drive...")  
    summarizer = pipeline("summarization", model=model_dir, tokenizer=model_dir)  
else:  
    print("Downloading model from Hugging Face for the first time...")  
    summarizer = pipeline("summarization", model="facebook/bart-large-cnn")  
    summarizer.model.save_pretrained(model_dir)  
    summarizer.tokenizer.save_pretrained(model_dir)  
    print("Model downloaded and saved to Google Drive")
```

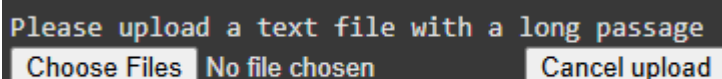
**Output: Downloading model from Hugging Face for the first time...**

**Model downloaded and saved to Google Drive**

### Step 6: Upload a text file

```
print("\nPlease upload a text file with a long passage")  
uploaded_file = files.upload()
```

### Output



```
Please upload a text file with a long passage  
Choose Files No file chosen Cancel upload
```

**Step 7: Read the uploaded file**

```
file_name = list(uploaded_file.keys())[0]
with open(file_name, 'r') as file:
    input_text = file.read()
```

**Step 8: Summarize the text file**

```
print("\n Summarizing... please wait.")
summary = summarizer(input_text, max_length=150, min_length=40, truncation=True)
summary_text = summary[0]['summary_text']
print("Summarization completed successfully")
```

**Output: Summarizing... please wait.**

**Summarization completed successfully**

**Step 9: Display output**

```
print("\n--- Original Text (First 500 characters) ---")
print(input_text[:500] + "..." if len(input_text) > 500 else input_text)
print("\n--- Summarized Text ---")
print(summary_text)
```

**--- Original Text (First 500 characters) ---**

**Artificial Intelligence (AI) is rapidly transforming the landscape of modern education, introducing a paradigm shift in how knowledge is delivered, absorbed, and assessed. One of the most significant contributions of AI is the advent of adaptive learning techniques. These systems analyze vast amounts of student data, including individual learning paces, existing strengths, and areas of weakness, to create truly personalized educational experiences. This level of customization allows students to ...**

**--- Summarized Text ---**

**Artificial Intelligence (AI) is rapidly transforming the landscape of modern education. AI-powered chatbots and virtual assistants are becoming increasingly common within academic institutions. AI plays a crucial role in enhancing accessibility and inclusivity in education.**

**Final Note:**

In this experiment, we use the pre-trained model facebook/bart-large-cnn for text summarization. The model is based on the BART transformer architecture, developed and trained by Facebook AI. This trained model is hosted publicly on the Hugging Face Model Hub, can be found in URL: <https://huggingface.co/models>

The program uses the Hugging Face transformers library to access and run the model with the following components.

- **Tokenizer and tokenization logic:** Provided by Hugging Face.

It converts input text into numerical token IDs that the model can understand (encoding), and later converts the model's output tokens back into readable text (decoding).

- **Model architecture and weights:** Developed and pre-trained by Facebook AI. These weights represent the learning the model has acquired and are specific to the summarization task. They are downloaded from Hugging Face's model hub using the model name facebook/bart-large-cnn.

- **Pipeline function:** Provided by Hugging Face's transformers library.

It bundles the tokenizer and model into a single easy-to-use interface, abstracting away complex setup steps.

The tokenizer and code logic come from Hugging Face, while the core model and trained weights come from Facebook AI, all integrated seamlessly using the transformers library and accessed via the Hugging Face platform.

## Experiment 8

### Custom Prompt Template with LangChain and Cohere

1. Install langchain, cohere (for key), langchain-community.
2. Get the API key (By logging into Cohere and obtaining the cohere key).
3. Load a text document from your google drive.
4. Create a prompt template to display the output in a particular manner.

#### AIM:

To install LangChain and Cohere, obtain a Cohere API key, load a text file from Google Drive, and use a LangChain prompt template to display the output in a structured format.

#### Introduction:

LangChain and Cohere are tools that help us work with large language models (LLMs). A large language model is an advanced AI system that can read, understand, and generate human-like text based on input. These models can answer questions, summarize content, and even carry out conversations.

Cohere is a company that provides access to such models via an API key. Think of an API key as a password that lets us connect to their service. Once connected, we can send text to their model and get intelligent responses.

LangChain is a Python based framework that allows us to easily structure interactions with LLMs. One of its features is the **PromptTemplate** that helps us decide exactly how we want our input or output text to look.

A **custom prompt template** is a **user-defined format** used to instruct a language model (LLM) to generate output in a specific way. It is built using the PromptTemplate class in LangChain and allows you to:

- **Control how the model responds** (e.g., as bullet points, Q&A, tables, etc.).
- Insert **dynamic content** (like document text) into a fixed structure using **placeholders** such as {content}.
- Make prompts **clear, reusable, and structured** to improve the quality and consistency of model output.

**For example**, we can ask the model to summarize text in bullet points or explain a topic in a question answer format.

**In this lab program, we will learn how to:**

- Install the required libraries.
- Use your Google Drive to load a document.
- Enter your Cohere API key securely.
- Create a template to control the output format from the model.
- Generate a response using Cohere's language model and LangChain.

**Objectives of the Program**

1. Install the required libraries (langchain, cohere, langchain-community).
2. Load a text document from Google Drive.
3. Configure the Cohere API key securely in Colab.
4. Use LangChain's PromptTemplate to format and display content from the document in a structured way.

**Step 1: Install LangChain, Cohere, and LangChain-Cohere Plugin**

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

**Step 2: Import required Python libraries**

```
from langchain import PromptTemplate
from langchain_community.llms import Cohere
from google.colab import drive
import os
from getpass import getpass
```

**Step 3: Mount your Google Drive to access text files**

```
drive.mount('/content/drive')
```



#### Step 4: Load and Read the text file content from Google Drive

```
file_path = "/content/drive/MyDrive/input.txt"
with open(file_path, 'r') as file:
    document_text = file.read()
print("Document loaded, length:", len(document_text), "characters")
```

#### Step 5: Set your Cohere API key securely (input will be hidden)

```
os.environ["COHERE_API_KEY"] = getpass("Enter your Cohere API key: ")
print("Cohere API key configured.")
```

To obtain a Cohere API key, follow these steps:

##### 1. Create a Cohere Account:

- Visit the Cohere Dashboard : <https://dashboard.cohere.com/welcome/login>
- If you don't have an account, sign up by providing the required details.
- Verify your email address to activate your account.

##### 2. Access the API Keys Section:

- After logging in, navigate to the API Keys page.

##### 3. Generate a New API Key:

- Click on “New Trial Key”.
- Assign a name to your key for easy identification (e.g., “MyFirstKey”).
- Click on “Generate Trial Key”.
- Your new API key will be displayed. **Copy and store it securely**, as you won't be able to view it again later.

#### Step 6: Define the prompt format using LangChain's PromptTemplate

```
template = """
Summarize the following document in three bullet points highlighting the key ideas:
{content}
Bullet Point Summary:
"""

prompt = PromptTemplate(input_variables=["content"], template=template)
```

### Step 7: Use the Cohere LLM to generate a response from the formatted prompt

```
llm = Cohere(max_tokens=150, temperature=0) # temperature=0 for consistent
output
formatted_prompt = prompt.format(content=document_text)
response = llm(formatted_prompt)

# Displaying the formatted response
print("Formatted Output:\n", response)
```

### Output

Here are the key ideas from the document about AI in education:

1. AI in education primarily impacts how knowledge is delivered, absorbed, and assessed through personalized learning, AI-powered chatbots, and virtual assistants, and AI-graded assessments.
2. AI also increases accessibility and inclusivity in education through language translation services and disability assistance technology to break down communication barriers and support diverse needs.
3. However, the integration of AI in education comes with challenges such as data privacy concerns, the risk of AI biases, and the need for educator training. Institutions are urged to adopt ethical AI frameworks for optimal outcomes and equitable learning environments.

### Final Note:

In this lab, we learned how to use LangChain and Cohere to create a custom prompt template for summarizing text. We used Google Drive to load a sample text file, entered our API key securely, and generated output using the LLM with a prompt structure we designed. This experiment demonstrates the power of using AI to format and control output, a key requirement in many real-world applications.