About Gen AI

Generative AI leverages deep learning techniques such as neural networks and transformers to create new data instances that resemble training data. This field has seen rapid advancements with the rise of generative adversarial networks (GANs) and diffusion models. With the explosion of large-scale models such as OpenAI's GPT series and Google's Bard, AI is reshaping industries by enabling automated creativity and innovation.

Benefits of the Course

1. **Comprehensive Hands-on Learning** – The course provides hands-on experience with generative models, allowing students to work on real-world datasets and build custom AI models.
2. **Industry-Relevant Skills Development** – Students gain expertise in AI model fine-tuning, embedding techniques, and practical applications, making them industry-ready.
3. **Enhancing Creativity and Problem Solving** – The ability to generate human-like content fosters new approaches to solving challenges in media, business automation, and personalized recommendations.
4. **Expanding Career Opportunities** – As AI adoption grows, demand for experts in AI model training, ethical AI development, and prompt engineering increases across domains.
5. **Encouraging AI-Driven Innovation** – Generative AI allows businesses to explore new ideas faster, optimize processes, and build AI-powered creative solutions.

Applications of Generative AI

- **Advanced Chatbots and Conversational AI** – Virtual assistants can respond more naturally and offer human-like interaction.
- **AI in Finance** – Generative AI models are being used for fraud detection, algorithmic trading, and financial forecasting.
- **Code Generation and Software Development** – Tools like GitHub Copilot assist developers by suggesting relevant code snippets and debugging solutions.
- **AI in Marketing and Advertising** – Personalized ad generation, automated social media content creation, and customer sentiment analysis.

Advantages of Learning Generative AI

1. **Ethical AI Considerations** – Understanding bias in AI models and the implications of AI-generated content ensures responsible development and deployment.
2. **Cutting-Edge Research Opportunities** – Generative AI plays a role in groundbreaking research across computational creativity and AI ethics.
3. **AI-powered Automation and Efficiency Gains** – AI-generated content speeds up workflows in content creation, graphic design, and personalized communication.

Course Content Overview

The course delves deeper into:

- **Fine-tuning Pre-trained Models:** Optimizing LLMs for domain-specific tasks.
- **Exploring Transformer Architectures:** Understanding self-attention mechanisms and how they contribute to generative capabilities.
- **Deploying AI Models in Production:** Building scalable AI applications for real-world use cases.
- **Developing Responsible AI:** Addressing bias, fairness, and explainability in generative AI systems.

Lab Programs

Program 1:

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

Theory:

Word Embeddings in NLP

Word Embeddings are numeric representations of words in a lower-dimensional space, capturing semantic and syntactic information. They play a vital role in Natural Language Processing (NLP) tasks. This article explores traditional and neural approaches, such as TF-IDF, Word2Vec, and GloVe, offering insights into their advantages and disadvantages. Understanding the importance of pre-trained word embeddings, providing a comprehensive understanding of their applications in various NLP scenarios.

What is Word Embedding in NLP?

Word Embedding is an approach for representing words and documents. Word Embedding or Word Vector is a numeric vector input that represents a word in a lower-dimensional space. It allows words with similar meanings to have a similar representation.

Word Embeddings are a method of extracting features out of text so that we can input those features into a machine learning model to work with text data. They try to preserve syntactical and semantic information. The methods such as Bag of Words (BOW), CountVectorizer and TFIDF rely on the word count in a sentence but do not save any syntactical or semantic information. In these algorithms, the size of the vector is the number of elements in the vocabulary. We can get a sparse matrix if most of the elements are zero. Large input vectors will mean a huge number of weights which will result in high computation required for training. Word Embeddings give a solution to these problems.

Need for Word Embedding?

To reduce dimensionality

To use a word to predict the words around it.

Inter-word semantics must be captured.

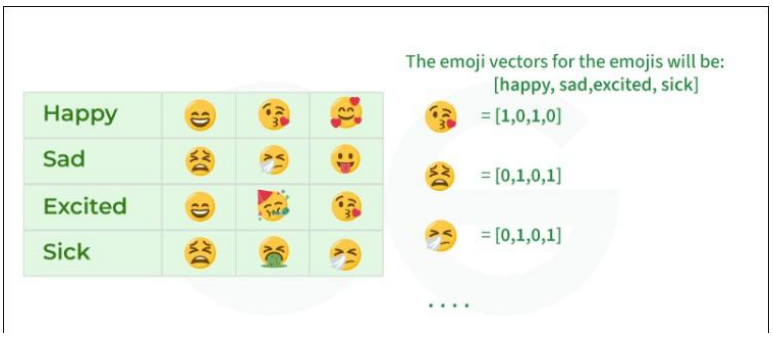
How are Word Embeddings used?

They are used as input to machine learning models.

Take the words —-> Give their numeric representation —-> Use in training or inference.

To represent or visualize any underlying patterns of usage in the corpus that was used to train them.

Let's take an example to understand how word vector is generated by taking emotions which are most frequently used in certain conditions and transform each emoji into a vector and the conditions will be our features.



In a similar way, we can create word vectors for different words as well on the basis of given features. The words with similar vectors are most likely to have the same meaning or are used to convey the same sentiment.

Approaches for Text Representation

1. Traditional Approach

The conventional method involves compiling a list of distinct terms and giving each one a unique integer value, or id. and after that, insert each word's distinct id into the sentence. Every vocabulary word is handled as a feature in this instance. Thus, a large vocabulary will result in an extremely large feature size. Common traditional methods include:

1.1. One-Hot Encoding

One-hot encoding is a simple method for representing words in natural language processing (NLP). In this encoding scheme, each word in the vocabulary is represented as a unique vector, where the dimensionality of the vector is equal to the size of the vocabulary. The vector has all elements set to 0, except for the element corresponding to the index of the word in the vocabulary, which is set to 1.

While one-hot encoding is a simple and intuitive method for representing words in NLP, it has several disadvantages, which may limit its effectiveness in certain applications.

One-hot encoding results in high-dimensional vectors, making it computationally expensive and memory-intensive, especially with large vocabularies.

It does not capture semantic relationships between words; each word is treated as an isolated entity without considering its meaning or context.

It is restricted to the vocabulary seen during training, making it unsuitable for handling out-of-vocabulary words.

1.2. Bag of Word (Bow)

Bag-of-Words (BoW) is a text representation technique that represents a document as an unordered set of words and their respective frequencies. It discards the word order and captures the frequency of each word in the document, creating a vector representation.

While BoW is a simple and interpretable representation, below disadvantages highlight its limitations in capturing certain aspects of language structure and semantics:

BoW ignores the order of words in the document, leading to a loss of sequential information and context making it less effective for tasks where word order is crucial, such as in natural language understanding.

BoW representations are often sparse, with many elements being zero resulting in increased memory requirements and computational inefficiency, especially when dealing with large datasets.

1.3. Term frequency-inverse document frequency (TF-IDF)

Term Frequency-Inverse Document Frequency, commonly known as TF-IDF, is a numerical statistic that reflects the importance of a word in a document relative to a collection of documents (corpus). It is widely used in natural language processing and information retrieval to evaluate the significance of a term within a specific document in a larger corpus. TF-IDF consists of two components:

Term Frequency (TF): Term Frequency measures how often a term (word) appears in a document. It is calculated using the formula:

$$\text{TF}(t,d) = \frac{\text{Total number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

Inverse Document Frequency (IDF): Inverse Document Frequency measures the importance of a term across a collection of documents. It is calculated using the formula:

$$\text{IDF}(t,D) = \log\left(\frac{\text{Total documents}}{\text{Number of documents containing term t}}\right)$$

The TF-IDF score for a term t in a document d is then given by multiplying the TF and IDF values:

$$TF - IDF(t,d,D) = TF(t,d) \times IDF(t,D)$$

The higher the TF-IDF score for a term in a document, the more important that term is to that document within the context of the entire corpus. This weighting scheme helps in identifying and extracting relevant information from a large collection of documents, and it is commonly used in text mining, information retrieval, and document clustering.

TF-IDF is a widely used technique in information retrieval and text mining, but its limitations should be considered, especially when dealing with tasks that require a deeper understanding of language semantics. For example:

TF-IDF treats words as independent entities and doesn't consider semantic relationships between them. This limitation hinders its ability to capture contextual information and word meanings.

Sensitivity to Document Length: Longer documents tend to have higher overall term frequencies, potentially biasing TF-IDF towards longer documents.

2. Neural Approach

2.1. Word2Vec

Word2Vec is a neural approach for generating word embeddings. It belongs to the family of neural word embedding techniques and specifically falls under the category of distributed representation models. It is a popular technique in natural language processing (NLP) that is used to represent words as continuous vector spaces. Developed by a team at Google, Word2Vec aims to capture the semantic relationships between words by mapping them to high-dimensional vectors. The underlying idea is that words with similar meanings should have similar vector representations. In Word2Vec every word is assigned a vector. We start with either a random vector or one-hot vector.

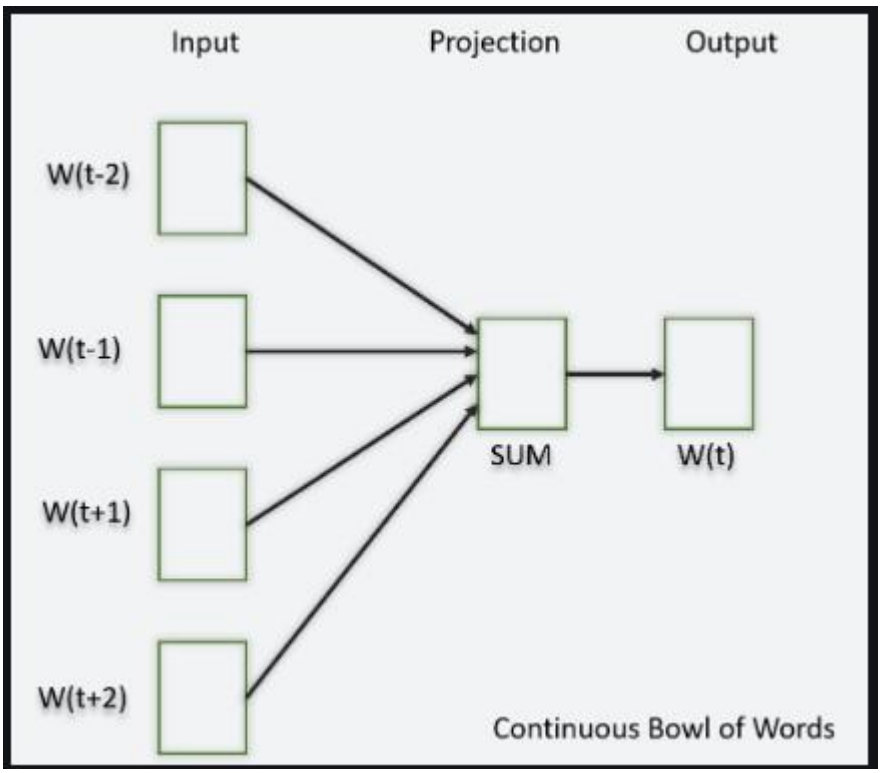
There are two neural embedding methods for Word2Vec, Continuous Bag of Words (CBOW) and Skip-gram.

2.2. Continuous Bag of Words(CBOW)

Continuous Bag of Words (CBOW) is a type of neural network architecture used in the Word2Vec model. The primary objective of CBOW is to predict a target word based on i ts context, which consists of the surrounding words in a given window. Given a sequence of words in a context window, the model is trained to predict the target word at the center of the window.

CBOW is a feedforward neural network with a single hidden layer. The input layer represents the context words, and the output layer represents the target word. The hidden layer contains the learned continuous vector representations (word embeddings) of the input words.

The architecture is useful for learning distributed representations of words in a continuous vector space.



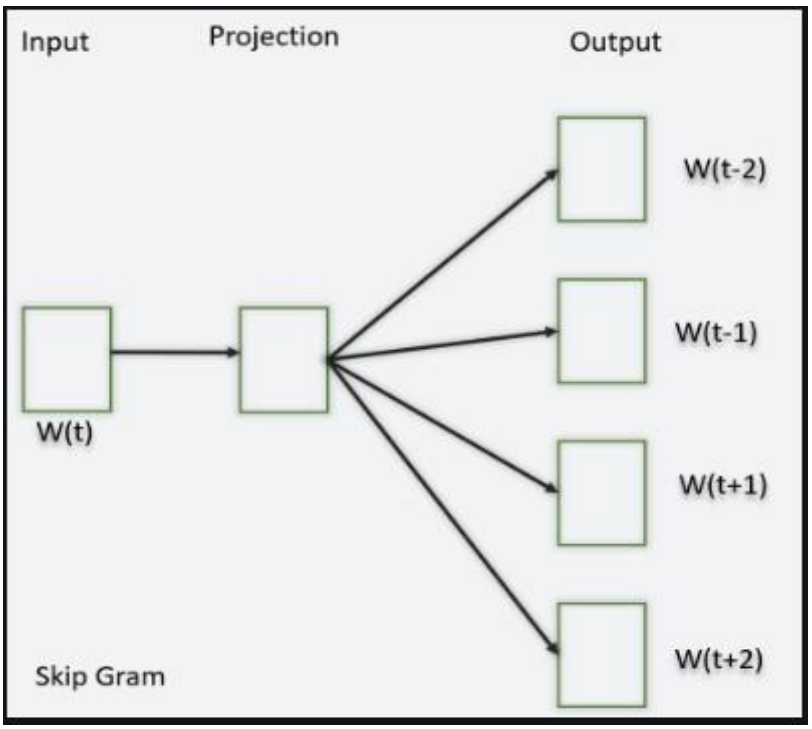
The hidden layer contains the continuous vector representations (word embeddings) of the input words.

The weights between the input layer and the hidden layer are learned during training.

The dimensionality of the hidden layer represents the size of the word embeddings (the continuous vector space).

2.3. Skip-Gram

The Skip-Gram model learns distributed representations of words in a continuous vector space. The main objective of Skip-Gram is to predict context words (words surrounding a target word) given a target word. This is the opposite of the Continuous Bag of Words (CBOW) model, where the objective is to predict the target word based on its context. It is shown that this method produces more meaningful embeddings.



After applying the above neural embedding methods we get trained vectors of each word after many iterations through the corpus. These trained vectors preserve syntactical or semantic information and are converted to lower dimensions. The vectors with similar meaning or semantic information are placed close to each other in space.

Let's understand with a basic example. The python code contains, vector_size parameter that controls the dimensionality of the word vectors, and you can adjust other parameters such as window based on your specific needs.

Note: Word2Vec models can perform better with larger datasets. If you have a large corpus, you might achieve more meaningful word embeddings.

In practice, the choice between CBOW and Skip-gram often depends on the specific characteristics of the data and the task at hand. CBOW might be preferred when training resources are limited, and capturing syntactic information is important. Skip-gram, on the other hand, might be chosen when semantic relationships and the representation of rare words are crucial..

3. Pretrained Word-Embedding

Pre-trained word embeddings are representations of words that are learned from large corpora and are made available for reuse in various natural language processing (NLP) tasks. These embeddings capture semantic relationships between words, allowing the model to understand similarities and relationships between different words in a meaningful way.

3.1. GloVe

GloVe is trained on global word co-occurrence statistics. It leverages the global context to create word embeddings that reflect the overall meaning of words based on their co-

occurrence probabilities. this method, we take the corpus and iterate through it and get the co-occurrence of each word with other words in the corpus. We get a co-occurrence matrix through this. The words which occur next to each other get a value of 1, if they are one word apart then 1/2, if two words apart then 1/3 and so on.

Let us take an example to understand how the matrix is created. We have a small corpus:

```
Corpus:
It is a nice evening.
Good Evening!
Is it a nice evening?
```

	it	is	a	nice	evening	good
it	0					
is	1+1	0				
a	1/2+1	1+1/2	0			
nice	1/3+1/2	1/2+1/3	1+1	0		
evening	1/4+1/3	1/3+1/4	1/2+1/2	1+1	0	
good	0	0	0	0	1	0

The upper half of the matrix will be a reflection of the lower half. We can consider a window frame as well to calculate the co-occurrences by shifting the frame till the end of the corpus. This helps gather information about the context in which the word is used.

Initially, the vectors for each word is assigned randomly. Then we take two pairs of vectors and see how close they are to each other in space. If they occur together more often or have a higher value in the co-occurrence matrix and are far apart in space then they are brought close to each other. If they are close to each other but are rarely or not frequently used together then they are moved further apart in space.

After many iterations of the above process, we'll get a vector space representation that approximates the information from the co-occurrence matrix. The performance of GloVe is better than Word2Vec in terms of both semantic and syntactic capturing.

3.2. Fasttext

Developed by Facebook, FastText extends Word2Vec by representing words as bags of character n-grams. This approach is particularly useful for handling out-of-vocabulary words and capturing morphological variations.

3.3. BERT (Bidirectional Encoder Representations from Transformers)

BERT is a transformer-based model that learns contextualized embeddings for words. It considers the entire context of a word by considering both left and right contexts, resulting in embeddings that capture rich contextual information.

Considerations for Deploying Word Embedding Models

You need to use the exact same pipeline during deploying your model as were used to create the training data for the word embedding. If you use a different tokenizer or different method of handling white space, punctuation etc. you might end up with incompatible inputs.

Words in your input that doesn't have a pre-trained vector. Such words are known as Out of Vocabulary Word(oov). What you can do is replace those words with "UNK" which means unknown and then handle them separately.

Dimension mis-match: Vectors can be of many lengths. If you train a model with vectors of length say 400 and then try to apply vectors of length 1000 at inference time, you will run into errors. So make sure to use the same dimensions throughout.

Advantages and Disadvantage of Word Embeddings

Advantages

It is much faster to train than hand build models like WordNet (which uses graph embeddings).

Almost all modern NLP applications start with an embedding layer.

It Stores an approximation of meaning.

Disadvantages

It can be memory intensive.

It is corpus dependent. Any underlying bias will have an effect on your model.

It cannot distinguish between homophones. Eg: brake/break, cell/sell, weather/whether etc.

Challenges in building word embedding from scratch

Training word embeddings from scratch is possible but it is quite challenging due to large trainable parameters and sparsity of training data. These models need to be trained on a large number of datasets with rich vocabulary and as there are large number of parameters, it makes the training slower. So, it's quite challenging to train a word embedding model on an individual level.

Pre Trained Word Embeddings

There's a solution to the above problem, i.e., using pre-trained word embeddings. Pre-trained word embeddings are trained on large datasets and capture the syntactic as well as semantic meaning of the words. This technique is known as transfer learning in which you take a model which is trained on large datasets and use that model on your own similar tasks.

There are two broad classifications of pre trained word embeddings – word-level and character-level. We'll be looking into two types of word-level embeddings i.e. Word2Vec and GloVe and how they can be used to generate embeddings.

Word2Vec

Word2Vec is one of the most popular pre trained word embeddings developed by Google. It is trained on Good news dataset which is an extensive dataset. As the name suggests, it represents each word with a collection of integers known as a vector. The vectors are calculated such that they show the semantic relation between words.

A popular example of how semantic relation is made is the king queen example:

King - Man + Woman ~ Queen

Word2vec is a feed-forward neural network which consists of two main models – Continuous Bag-of-Words (CBOW) and Skip-gram model. The continuous bag of words model learns the target word from the adjacent words whereas in the skip-gram model, the model learns the adjacent words from the target word. They are completely opposite of each other.

Firstly, the size of context window is defined. Context window is a sliding window which runs through the whole text one word at a time. It basically refers to the number of words appearing on the right and left side of the focus word. eg. if size of the context window is set to 2, then it will include 2 words on the right as well as left of the focus word.

Focus word is our target word for which we want to create the embedding / vector representation. Generally, focus word is the middle word but in the example below we're taking last word as our target word. The neighbouring words are the words that appear in the context window. These words help in capturing the context of the whole sentence. Let's understand this with the help of an example.

Suppose we have a sentence – "He poured himself a cup of coffee". The target word here is "himself".

Continuous Bag-Of-Words –

input = ["He", "poured", "a", "cup"]

output = ["himself"]

Skip-gram model –

input = ["himself"]

output = ["He", "poured", "a", "cup"]

This can be used to generate high-quality word embeddings.

GloVe

Given by Stanford, GloVe stands for Global Vectors for Word Representation. It is a popular word embedding model which works on the basic idea of deriving the relationship between words using statistics. It is a count based model that employs co-occurrence matrix. A co-occurrence matrix tells how often two words are occurring globally. Each value is a count of a pair of words occurring together.

Glove basically deals with the spaces where the distance between words is linked to to their semantic similarity. It has properties of the global matrix factorisation and the local context window technique. Training of the model is based on the global word-word co-occurrence data from a corpse, and the resultant representations results into linear substructure of the vector space

GloVe calculates the co-occurrence probabilities for each word pair. It divides the co-occurrence counts by the total number of co-occurrences for each word:

F(w_{i}, w_{j}, w_{k}) = \frac {P_{ik}}{P_{jk}}

For example, the co-occurrence probability of “cat” and “mouse” is calculated as: Co-occurrence Probability(“cat”, “mouse”) = Count(“cat” and “mouse”) / Total Co-occurrences(“cat”)

In this case:

Count("cat" and "mouse") = 1

Total Co-occurrences("cat") = 2 (with "chases" and "mouse")

So, Co-occurrence Probability("cat", "mouse") = 1 / 2 = 0.5

Conclusion

In conclusion, word embedding techniques such as TF-IDF, Word2Vec, and GloVe play a crucial role in natural language processing by representing w ords in a lower-dimensional space, capturing semantic and syntactic information.

Facts:

- GPT uses context-based embeddings rather than traditional word embeddings. It captures word meaning in the context of the entire sentence.
- BERT is contextually aware, considering the entire sentence, while traditional word embeddings, like Word2Vec, treat each word independently.
- Word embeddings can be broadly evaluated in two categories, intrinsic and extrinsic. For intrinsic evaluation, word embeddings are used to calculate or predict semantic similarity between words, terms, or sentences.
- Word vectorization converts words into numerical vectors, capturing semantic relationships. Techniques like TF-IDF, Word2Vec, and GloVe are common.

- Word embeddings offer semantic understanding, capture context, and enhance NLP tasks. They reduce dimensionality, speed up training, and aid in language pattern recognition.

  To explore pre-trained word vectors and analyze word relationships using vector arithmetic, we can use the gensim library, which provides an easy-to-use interface for working with pre-trained word embeddings like Word2Vec, GloVe, and FastText. Below is a Python program that demonstrates how to load pre-trained word vectors, perform vector arithmetic, and analyze the results.

Explanation of the Code

Loading Pre-trained Word Vectors:

The gensim.downloader module provides access to various pre-trained word vectors. In this example, we load the word2vec-google-news-300 model, which contains 300-dimensional word vectors trained on Google News data.

Vector Arithmetic:

The function explore_word_relationships performs vector arithmetic (e.g., king - man + woman) and finds the most similar words to the resulting vector. This is a common way to explore analogies and relationships between words.

Similarity Analysis:

The analyze_similarity function calculates the cosine similarity between two words, which indicates how similar their meanings are based on their vector representations.

Exclude Input Words from Results

When performing vector arithmetic, we can explicitly exclude the input words (king, man, woman) from the results to ensure that the output is meaningful.

Finding Similar Words:

The find_most_similar function retrieves the most similar words to a given word, which helps in understanding the context and usage of the word.

Conclusion

This program demonstrates how to explore pre-trained word vectors, perform vector arithmetic, and analyze word relationships using the gensim library. You can modify the word examples or explore other pre-trained models available in the gensim.downloader module.

Can use a Different Models

such as # Load GloVe

embeddings

word_vectors = api.load("glove-wiki-

gigaword-300") # Load FastText embeddings

word_vectors = api.load("fasttext-wiki-news-subwords-

300") Normalize the Vectors

Sometimes, normalizing the vectors before performing arithmetic operations can improve results. def explore_word_relationships(word1, word2, word3):

```
try:
    # Normalize vectors before arithmetic
    vec1 = word_vectors[word1] / norm(word_vectors[word1]) vec2 = word_vectors[word2] / norm(word_vectors[word2])
    vec3 = word_vectors[word3] / norm(word_vectors[word3])

    # Perform vector arithmetic: word1 - word2 + word3
    result_vector = vec1 - vec2 + vec3

    # Find the most similar words to the resulting vector
    similar_words = word_vectors.similar_by_vector(result_vector, topn=5)

    print(f"\nWord Relationship: {word1} - {word2} + {word3}") print("Most similar words to the result:")
    for word, similarity in similar_words:
        print(f"{word}: {similarity:.4f}")

except KeyError as e:
    print(f"Error: {e} not found in the vocabulary.")
```

Final Notes

The word2vec-google-news-300 model is not perfect, and its performance can vary depending on the specific analogy.

If you need more consistent results, consider using a different model or training your own embeddings on a domain-specific corpus.

Source code:

```
# Install necessary libraries
!pip install gensim numpy

# Import libraries
import gensim.downloader as api
import numpy as np
from numpy.linalg import norm

# Load pre-trained word vectors
print("Loading pre-trained word vectors...")
word_vectors = api.load("word2vec-google-news-300")

# Function to perform vector arithmetic and find similar words
def explore_word_relationships(word1, word2, word3):
    try:
        # Get vectors for the input words
        vec1 = word_vectors[word1]
        vec2 = word_vectors[word2]
        vec3 = word_vectors[word3]

        # Perform vector arithmetic: word1 - word2 + word3
        result_vector = vec1 - vec2 + vec3

        # Find the most similar words to the resulting vector
        similar_words = word_vectors.similar_by_vector(result_vector, topn=10)

        # Exclude input words from the results
        input_words = {word1, word2, word3}
        filtered_words = [(word, similarity) for word, similarity in similar_words if word not in
input_words]

        print(f"\nWord Relationship: {word1} - {word2} + {word3}")
        print("Most similar words to the result (excluding input words):")
        for word, similarity in filtered_words[:5]:  # Show top 5 results
            print(f"{word}: {similarity:.4f}")

    except KeyError as e:
        print(f"Error: {e} not found in the vocabulary.")

# Example word relationships to explore
explore_word_relationships("king", "man", "woman")
explore_word_relationships("paris", "france", "germany")
explore_word_relationships("apple", "fruit", "carrot")

# Function to analyze the similarity between two words
def analyze_similarity(word1, word2):
    try:
        similarity = word_vectors.similarity(word1, word2)
        print(f"\nSimilarity between '{word1}' and '{word2}': {similarity:.4f}")
    except KeyError as e:
```

```
        print(f"Error: {e} not found in the vocabulary.")

# Example similarity analysis
analyze_similarity("cat", "dog")
analyze_similarity("computer", "keyboard")
analyze_similarity("music", "art")

# Function to find the most similar words to a given word
def find_most_similar(word):
    try:
        similar_words = word_vectors.most_similar(word, topn=5)
        print(f"\nMost similar words to '{word}':")
        for similar_word, similarity in similar_words:
            print(f"{similar_word}: {similarity:.4f}")
    except KeyError as e:
        print(f"Error: {e} not found in the vocabulary.")

# Example: Find most similar words
find_most_similar("happy")
find_most_similar("sad")
find_most_similar("technology")
```

Output:

```
Loading pre-trained word vectors...

Word Relationship: king - man + woman
Most similar words to the result (excluding input words):
queen: 0.7301
monarch: 0.6455
princess: 0.6156
crown_prince: 0.5819
prince: 0.5777

Word Relationship: paris - france + germany
Most similar words to the result (excluding input words):
berlin: 0.4838
german: 0.4695
lindsay_lohan: 0.4536
switzerland: 0.4468
heidi: 0.4445

Word Relationship: apple - fruit + carrot
Most similar words to the result (excluding input words):
carrots: 0.5700
proverbial_carrot: 0.4578
Carrot: 0.4159
```

Twizzler: 0.4074
peppermint_candy: 0.407

Similarity between 'cat' and 'dog': 0.7609

Similarity between 'computer' and 'keyboard': 0.3964

Similarity between 'music' and 'art': 0.4010

Most similar words to 'happy':
glad: 0.7409
leased: 0.6632
ecstatic: 0.6627
overjoyed: 0.6599
thrilled: 0.6514

Most similar words to 'sad':
saddening: 0.7273
Sad: 0.6611
saddened: 0.6604
heartbreaking: 0.6574
disheartening: 0.6507

Most similar words to 'technology':
technologies: 0.8332
innovations: 0.6231
technological_innovations: 0.6102
technol: 0.6047
technological_advancement: 0.6036

Program 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.

Theory:

To visualize word embeddings, t-SNE (t-distributed Stochastic Neighbor Embedding) is generally preferred over PCA (Principal Component Analysis) because t-SNE is better at capturing complex, non-linear relationships between words, while PCA focuses on preserving global variance and linear patterns, which can be less effective for visualizing the semantic similarities between words in an embedding space.

Key points to remember:

t-SNE for local structure:

t-SNE excels at preserving local similarities between data points, meaning nearby words in the embedding space will also appear close together in the visualization, making it ideal for understanding semantic relationships between words.

PCA for global variance:

PCA prioritizes capturing the most variance in the data, which can be less useful for visualizing the nuanced relationships between words in an embedding space.

What is t-SNE ?

t-SNE (t-distributed Stochastic Neighbor Embedding) is an unsupervised non-linear dimensionality reduction technique for data exploration and visualizing high-dimensional data. Non-linear dimensionality reduction means that the algorithm allows us to separate data that cannot be separated by a straight line.

t-SNE gives you a feel and intuition on how data is arranged in higher dimensions. It is often used to visualize complex datasets into two and three dimensions, allowing us to understand more about underlying patterns and relationships in the data.

t-SNE vs PCA

Both t-SNE and PCA are dimensional reduction techniques with different mechanisms that work best with different types of data.

PCA (Principal Component Analysis) is a linear technique that works best with data that has a linear structure. It seeks to identify the underlying principal components in the data by projecting onto lower dimensions, minimizing variance, and preserving large pairwise

distances. Read our Principal Component Analysis (PCA) tutorial to understand the inner workings of the algorithms with R examples.

But, t-SNE is a nonlinear technique that focuses on preserving the pairwise similarities between data points in a lower-dimensional space. t-SNE is concerned with preserving small pairwise distances whereas, PCA focuses on maintaining large pairwise distances to maximize variance.

In summary, PCA preserves the variance in the data. In contrast, t-SNE preserves the relationships between data points in a lower-dimensional space, making it quite a good algorithm for visualizing complex high-dimensional data.

Final notes

The error ValueError: perplexity must be less than n_samples occurs when using t-SNE for dimensionality reduction. This error arises because the perplexity parameter in t-SNE must be smaller than the number of samples (words) being visualized.

What is Perplexity in t-SNE?

Perplexity is a hyperparameter in t-SNE that controls how the algorithm balances attention between local and global patterns in the data.

It can be thought of as the number of effective neighbors each point has. A higher perplexity value makes t-SNE focus more on global structure, while a lower value makes it focus on local structure.

The perplexity value must satisfy: $1 < \text{perplexity} < n\_samples$, where n_samples is the number of data points (words) being visualized.

Why Does the Error Occur?

The error occurs when the perplexity value is set to a number greater than or equal to the number of samples (words) in your dataset. For example:

If you have 10 words and set perplexity=15, you will get this error because $15 > 10$.

How to Resolve the Error

1. Reduce the Perplexity Value

Ensure that the perplexity value is less than the number of samples (words) being visualized.

A common rule of thumb is to set perplexity to a value between 5 and 50, depending on the dataset size.

For small datasets (e.g., fewer than 20 words), use a smaller perplexity value (e.g., perplexity=5).

Example:

```
tsne = TSNE(n_components=2, random_state=42, perplexity=5)  # Set perplexity to 5
```

2. Increase the Number of Samples

If you have too few words, consider adding more words to your dataset for visualization.

For example, instead of visualizing just 5 words, visualize 20 or more words.

Example:

```
words_to_visualize = ["king", "queen", "man", "woman", "prince", "princess", "royal", "throne", "kingdom", "crown"]
```

3. Check the Number of Samples

Before applying t-SNE, check the number of samples (words) in your dataset to ensure it is greater than the perplexity value.

Example:

```
n_samples = len(words_to_visualize)
print(f"Number of samples: {n_samples}")
if perplexity >= n_samples:
    raise ValueError(f"Perplexity must be less than {n_samples}. Current perplexity: {perplexity}")
```

2 (b)

The program includes:

Theory and Concepts: Explanation of word embeddings, visualization, and analysis.

Visualization of Word Embeddings: Using PCA and t-SNE for dimensionality reduction.

Semantic Similarity: Generating 5 semantically similar words for a given input.

Theory and Concepts

1. **Word Embeddings**

Word embeddings are dense vector representations of words in a continuous vector space. They capture semantic relationships between words, such as similarity, analogy, and context. Popular pre-trained word embedding models include:

Word2Vec: Trained on large text corpora using a neural network.

GloVe: Global Vectors for Word Representation, based on matrix factorization.

FastText: Extends Word2Vec by representing words as n-grams of characters.

2. **Dimensionality Reduction**

Word embeddings are high-dimensional (e.g., 300 dimensions). To visualize them, we reduce their dimensionality to 2D or 3D using:

PCA (Principal Component Analysis): A linear technique that projects data onto orthogonal axes.

t-SNE (t-Distributed Stochastic Neighbor Embedding): A non-linear technique that preserves local relationships between points.

3. **Semantic Similarity**

Semantic similarity measures how closely related two words are in meaning. For example:

Similar Words: "king" and "queen" are semantically similar.

Analogies: "king - man + woman = queen" demonstrates relationships between words.

Both PCA (Principal Component Analysis) and t-SNE (t-Distributed Stochastic Neighbor Embedding) are dimensionality reduction techniques used to visualize high-dimensional data (e.g., word embeddings) in 2D or 3D. Below, I explain the parameters of their function calls in Python's scikit-learn library, along with their default values.

1. **PCA (Principal Component Analysis)**

Function Call

```
from sklearn.decomposition import PCA

pca = PCA(n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', random_state=None)
```

Parameters

- n_components:
- Purpose: Number of components (dimensions) to reduce the data to.
- Default: None (keeps all components).
- Common Use: Set to 2 or 3 for visualization.

copy:

- Purpose: Whether to create a copy of the input data before applying PCA.
- Default: True (creates a copy).
- Common Use: Set to False to save memory if the input data is large.

whiten:

- Purpose: Whether to whiten the data (scale the components to unit variance).
- Default: False.
- Common Use: Set to True if you want to decorrelate the components.

svd_solver:

- Purpose: Algorithm to use for SVD (Singular Value Decomposition).
- Options:
- 'auto': Automatically chooses the best solver based on input data.
- 'full': Uses the full SVD algorithm.
- 'arpack': Uses the ARPACK solver for sparse matrices.
- 'randomized': Uses a randomized SVD solver.
- Default: 'auto'.

tol:

- Purpose: Tolerance for singular values in the SVD solver.
- Default: 0.0.
- Common Use: Adjust for numerical stability.

iterated_power:

- Purpose: Number of iterations for the power method when using the 'randomized' solver.
- Default: 'auto' (automatically determined).
- Common Use: Adjust for better convergence.

random_state:

- Purpose: Seed for random number generation (used in the 'randomized' solver).
- Default: None.
- Common Use: Set to an integer for reproducible results.

2. **t-SNE (t-Distributed Stochastic Neighbor Embedding)**

Function Call

from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=30.0, early_exaggeration=12.0, learning_rate=200.0, n_iter=1000, n_iter_without_progress=300, min_grad_norm=1e-07, metric='euclidean', init='random', verbose=0, random_state=None, method='barnes_hut', angle=0.5, n_jobs=None)

Parameters

n_components:

- Purpose: Number of dimensions to reduce the data to.
- Default: 2.
    - Common Use: Set to 2 or 3 for visualization.

perplexity:

- Purpose: Controls the number of effective neighbors for each point. Higher values focus on global structure, while lower values focus on local structure.
- Default: 30.0.
- Common Use: Adjust based on dataset size (e.g., 5 for small datasets, 30 for larger datasets).

early_exaggeration:

- Purpose: Controls how tightly clusters are formed in the early stages of optimization.
- Default: 12.0.
- Common Use: Adjust for better cluster separation.

learning_rate:

- Purpose: Controls the step size during optimization.
- Default: 200.0.
- Common Use: Adjust if the visualization looks too crowded or too spread out.

n_iter:

- Purpose: Maximum number of iterations for optimization.
- Default: 1000.
- Common Use: Increase for better convergence.

n_iter_without_progress:

- Purpose: Maximum number of iterations without progress before stopping.
- Default: 300.
- Common Use: Adjust for early stopping.
- min_grad_norm:
- Purpose: Minimum gradient norm for stopping optimization.
- Default: 1e-07.
- Common Use: Adjust for numerical

stability. metric:

- Purpose: Distance metric to use for computing similarities.
- Default: 'euclidean'.
- Common Use: Use 'cosine' for word embeddings.

init:

- Purpose: Initialization method for the embedding.
- Options:
- 'random': Random initialization.
- 'pca': Use PCA for initialization.
- Default: 'random'.
- Common Use: Use 'pca' for better initialization.

verbose:

- Purpose: Controls the verbosity of the output.
- Default: 0 (no output).
- Common Use: Set to 1 for progress updates.

random_state:

- Purpose: Seed for random number generation.
- Default: None.
- Common Use: Set to an integer for reproducible results.

method:

- Purpose: Algorithm to use for t-SNE.

Options:

- 'barnes_hut': Faster approximation for large datasets.
- 'exact': Exact computation (slower).
- Default: 'barnes_hut'.
- Common Use: Use 'barnes_hut' for faster results.

angle:

- Purpose: Trade-off between speed and accuracy for the Barnes-Hut algorithm.
- Default: 0.5.
- Common Use: Adjust for better performance.

n_jobs:

- Purpose: Number of parallel jobs to run.
- Default: None.
- Common Use: Set to -1 to use all available cores.
- Default Values in Practice

PCA:

- For visualization, set n_components=2 or 3.
- Use svd_solver='auto' for automatic solver selection.
- Set random_state for reproducibility.

t-SNE:

- For visualization, set n_components=2.
- Adjust perplexity based on dataset size (e.g., 5 for small datasets, 30 for larger datasets).
- Use init='pca' for better initialization.
- Set random_state for reproducibility.

To visualize word embeddings using dimensionality reduction techniques like PCA (Principal Component Analysis) or t-SNE (t-Distributed Stochastic Neighbor Embedding), we can reduce the high-dimensional word vectors (e.g., 300 dimensions) to 2D or 3D for plotting. This allows us to visualize the relationships between words in a 2D or 3D space.

Explanation of the Code

Dimensionality Reduction:

- PCA: Reduces the dimensionality of the word vectors to 2D using Principal Component Analysis. PCA is a linear technique and is faster but may not capture non-linear relationships.
- t-SNE: Reduces the dimensionality to 2D using t-SNE, which is a non-linear technique. It is better at preserving local relationships between points but is slower and requires tuning the perplexity parameter.

Visualization:

- The reduced vectors are plotted in a 2D space using matplotlib.
- Each word is represented as a point, and the word labels are displayed next to the points.

Word Relationships:

- The explore_word_relationships function performs vector arithmetic (e.g., king - man + woman) and finds the most similar words.
- The results are filtered to exclude the input words (king, man, woman).

Words to Visualize:

- A list of words (words_to_explore) is defined for visualization.
- The filtered words from the vector arithmetic are added to this list.

Plotting:

- The program generates two plots: one using PCA and another using t-SNE.

Example Output

PCA Visualization

- Words like king, queen, man, woman, prince, and princess will be plotted in a 2D space.
- Words with similar meanings or relationships (e.g., king and queen) will appear closer together.

t-SNE Visualization

- t-SNE will group similar words more tightly, but the overall layout may differ from PCA due to its non-linear nature.

Notes

Perplexity in t-SNE: The perplexity parameter controls how t-SNE balances local and global relationships. You can adjust it (e.g., perplexity=5 to perplexity=30) to see how it affects the visualization.

Performance: t-SNE is slower than PCA, especially for large datasets. If you're visualizing many words, consider using PCA for faster results.

Key Points to Remember

- Perplexity Value:
- Must satisfy 1 < perplexity < n_samples.
- For small datasets, use a small perplexity value (e.g., perplexity=5).
- For larger datasets, you can use a higher perplexity value (e.g., perplexity=30).

Dataset Size:

- Ensure you have enough samples (words) to visualize. If the dataset is too small, consider adding more words.

Error Handling:

- Always check the number of samples and the perplexity value before applying t-SNE.

Example Output

If the perplexity is set correctly, the program will generate a 2D plot of the word embeddings using t-SNE. Words with similar meanings or relationships (e.g., king and queen) will appear closer together in the plot.

Source code:

1. Use dimensionality reduction (e.g., PCA or t-SNE) to visualize word embeddings for Q 1.

```
# Install required libraries
!pip install gensim numpy matplotlib sklearn

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

```
# Load pre-trained word vectors
print("Loading pre-trained word vectors...")
word_vectors = api.load("word2vec-google-news-300") # Load Word2Vec model

# Function to perform vector arithmetic and find similar words
def explore_word_relationships(word1, word2, word3):
    try:
        # Perform vector arithmetic: word1 - word2 + word3
        result_vector = word_vectors[word1] - word_vectors[word2] + word_vectors[word3]

        # Find the most similar words to the resulting vector
        similar_words = word_vectors.similar_by_vector(result_vector, topn=10)

        # Exclude input words from the results
        input_words = {word1, word2, word3}
        filtered_words = [(word, similarity) for word, similarity in similar_words if word not in
input_words]

        print(f"\nWord Relationship: {word1} - {word2} + {word3}")
        print("Most similar words to the result (excluding input words):")
        for word, similarity in filtered_words[:5]: # Show top 5 results
            print(f"{word}: {similarity:.4f}")

        return filtered_words

    except KeyError as e:
        print(f"Error: {e} not found in the vocabulary.")
        return []

# Function to visualize word embeddings using PCA or t-SNE
def visualize_word_embeddings(words, vectors, method='pca'):
    # Reduce dimensionality to 2D
    if method == 'pca':
        reducer = PCA(n_components=2)
    elif method == 'tsne':
        reducer = TSNE(n_components=2, random_state=42, perplexity=3) # Adjust perplexity
as needed
    else:
        raise ValueError("Method must be 'pca' or 'tsne'.")

    # Fit and transform the vectors
    reduced_vectors = reducer.fit_transform(vectors)
```

```
    # Plot the results
    plt.figure(figsize=(10, 8))
    for i, word in enumerate(words):
        plt.scatter(reduced_vectors[i, 0], reduced_vectors[i, 1], marker='o', color='blue')
        plt.text(reduced_vectors[i, 0] + 0.02, reduced_vectors[i, 1] + 0.02, word, fontsize=12)

    plt.title(f"Word Embeddings Visualization using {method.upper()}")
    plt.xlabel("Component 1")
    plt.ylabel("Component 2")
    plt.grid(True)
    plt.show()

# Example word relationships to explore
words_to_explore = ["king", "man", "woman", "queen", "prince", "princess", "royal",
"throne"]
filtered_words = explore_word_relationships("king", "man", "woman")

# Add the filtered words to the list of words to visualize
words_to_visualize = words_to_explore + [word for word, _ in filtered_words]

# Get vectors for the words to visualize
vectors_to_visualize = np.array([word_vectors[word] for word in words_to_visualize])

# Visualize using PCA
visualize_word_embeddings(words_to_visualize, vectors_to_visualize, method='pca')

# Visualize using t-SNE
visualize_word_embeddings(words_to_visualize, vectors_to_visualize, method='tsne')
```

2. 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.

```
# Install required libraries
!pip install gensim scikit-learn matplotlib

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

```
# Load pre-trained word vectors
print("Loading pre-trained word vectors...")
word_vectors = api.load("word2vec-google-news-300") # Load Word2Vec model

# Select 10 words from a specific domain (e.g., technology)
domain_words = ["computer", "software", "hardware", "algorithm", "data", "network",
"programming", "machine", "learning", "artificial"]

# Get vectors for the selected words
domain_vectors = np.array([word_vectors[word] for word in domain_words])

# Function to visualize word embeddings using PCA or t-SNE
def visualize_word_embeddings(words, vectors, method='pca', perplexity=5):
    # Reduce dimensionality to 2D
    if method == 'pca':
        reducer = PCA(n_components=2)
    elif method == 'tsne':
        reducer = TSNE(n_components=2, random_state=42, perplexity=perplexity)
    else:
        raise ValueError("Method must be 'pca' or 'tsne'.")

    # Fit and transform the vectors
    reduced_vectors = reducer.fit_transform(vectors)

    # Plot the results
    plt.figure(figsize=(10, 8))
    for i, word in enumerate(words):
        plt.scatter(reduced_vectors[i, 0], reduced_vectors[i, 1], marker='o', color='blue')
        plt.text(reduced_vectors[i, 0] + 0.02, reduced_vectors[i, 1] + 0.02, word, fontsize=12)

    plt.title(f"Word Embeddings Visualization using {method.upper()}")
    plt.xlabel("Component 1")
    plt.ylabel("Component 2")
    plt.grid(True)
    plt.show()

# Visualize using PCA
visualize_word_embeddings(domain_words, domain_vectors, method='pca')

# Visualize using t-SNE
visualize_word_embeddings(domain_words, domain_vectors, method='tsne', perplexity=3)

# Function to generate 5 semantically similar words
def generate_similar_words(word):
```

```
    try:
        similar_words = word_vectors.most_similar(word, topn=5)
        print(f"\nTop 5 semantically similar words to '{word}':")
        for similar_word, similarity in similar_words:
            print(f"{similar_word}: {similarity:.4f}")
    except KeyError as e:
        print(f"Error: {e} not found in the vocabulary.")

# Example: Generate similar words for a given input
generate_similar_words("computer")
generate_similar_words("learning")
```

Explanation of the Code

Loading Pre-trained Word Vectors:

- The gensim.downloader module loads the word2vec-google-news-300 model, which contains 300-dimensional word vectors.
- Selecting Domain-Specific Words:
- A list of 10 words from the technology domain is selected for visualization and analysis.

Dimensionality Reduction:

- PCA and t-SNE are used to reduce the 300-dimensional word vectors to 2D for visualization.
- Visualization:
- The reduced vectors are plotted in a 2D space using matplotlib. Words with similar meanings appear closer together.

Semantic Similarity:

- The generate_similar_words function finds the top 5 semantically similar words for a given input word using the most_similar method.

Step 4: Output and Analysis

Visualization Output

- PCA Plot: Words like "computer," "software," and "hardware" will appear close to each other, indicating their semantic similarity.
- t-SNE Plot: Words like "machine," "learning," and "artificial" will form a tight cluster, reflecting their contextual relationships.

This program demonstrates how to:

Visualize word embeddings using PCA and t-SNE.

Analyze clusters and relationships between words.

Generate semantically similar words using pre-trained embeddings.

Output:

2 (a) Loading pre-trained word vectors...

Word Relationship: king - man + woman

Most similar words to the result (excluding input words):

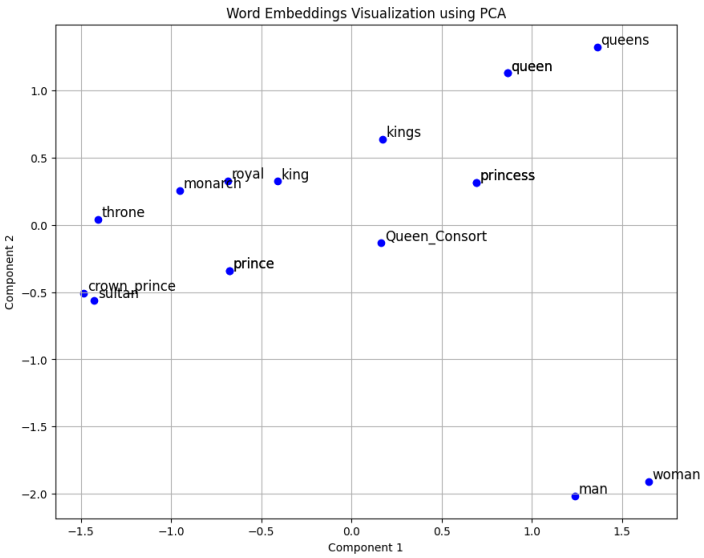
queen: 0.7301

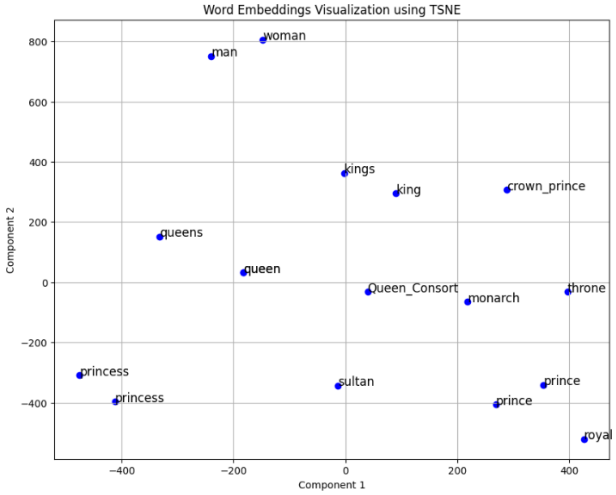
monarch: 0.6455

princess: 0.6156

crown_prince: 0.5819

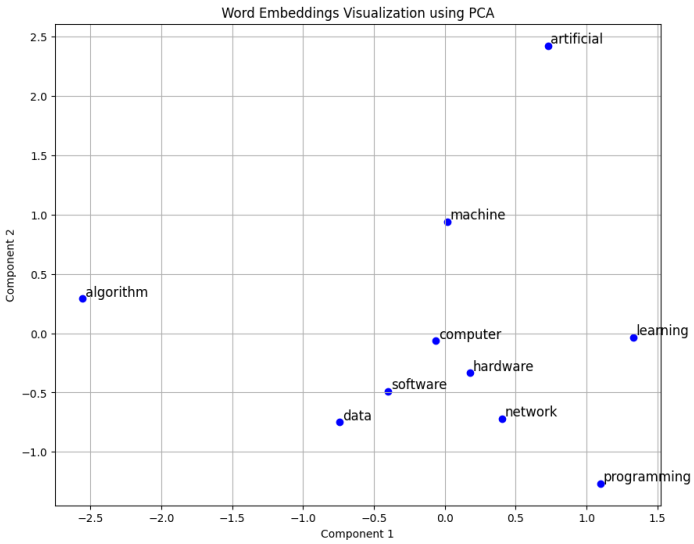
prince: 0.5777

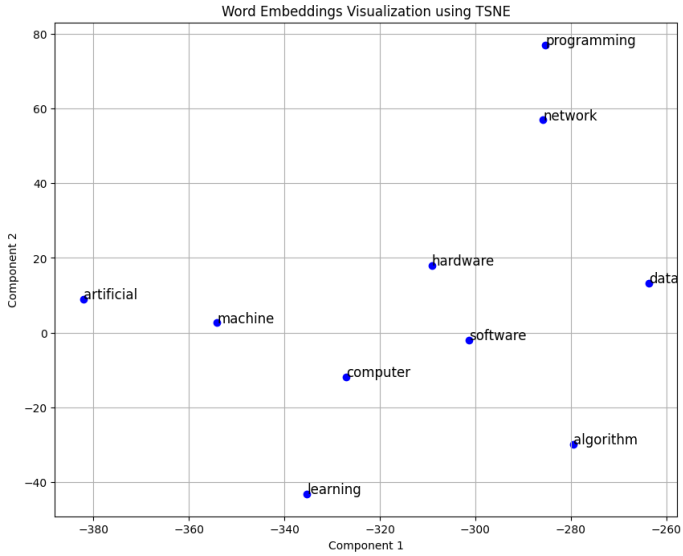




Output 2 (b)

Loading pre-trained word vectors…





Top 5 semantically similar words to 'computer':

computers: 0.7979

laptop: 0.6640

laptop_computer: 0.6549

Computer: 0.6473

com_puter: 0.6082

Top 5 semantically similar words to 'learning':

teaching: 0.6602

learn: 0.6365

Learning: 0.6208

reteaching: 0.5810

learner_centered: 0.5739

Program 3:

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

Theory:

This program builds a domain-specific Word2Vec model for a medical corpus, visualizes the learned word embeddings using t-SNE, and retrieves similar words using the trained model.

1. Word Embeddings & Word2Vec

- Word Embeddings are numerical vector representations of words that capture their meanings and relationships.
- Word2Vec is a popular neural network-based algorithm that learns word embeddings from a corpus.
- There are two architectures for Word2Vec:
    - CBOW (Continuous Bag of Words): Predicts a word based on surrounding context words.
    - Skip-Gram: Predicts surrounding context words given a central word.

In this program, we use the Word2Vec model to train embeddings on a medical corpus, capturing the domain-specific relationships between words.

---

2. Pre-processing the Corpus

- Tokenization: Splitting sentences into words.
- Lowercasing: Normalizing text to avoid duplication due to case differences.

---

3. Training the Word2Vec Model

- Hyperparameters used in training:
    - vector_size=100: Each word is represented by a 100-dimensional vector.
    - window=5: Words within a 5-word context window influence each other.
    - min_count=1: Even words occurring once are included in training.
    - workers=4: Uses 4 CPU threads for parallel processing.
    - epochs=50: The training iterates over the dataset 50 times for better learning.

---

4. Extracting Word Embeddings for Visualization

- Embeddings: After training, every word has a 100-dimensional vector representation.
- Dimensionality Reduction: We apply t-SNE (t-distributed Stochastic Neighbor Embedding) to reduce 100 dimensions to 2D for visualization.

---

5. Visualizing Word Embeddings using t-SNE

- Why use t-SNE?
  - Helps visualize high-dimensional data in 2D space.
  - Maintains the relative closeness of similar words.
- Scatter Plot
  - Words that are semantically related will cluster together.
  - Example: "vaccine" and "infection" may appear close to each other.

---

6. Finding Semantically Similar Words

- Uses the trained Word2Vec model to retrieve top-N most similar words.
- Cosine Similarity is used to find similarity between word vectors:

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{||A||||B||}$$

Example:

- Query: "treatment"
- Output: "therapy", "medicine", "procedure", etc.

Step-by-Step Execution Flow

Step 1: Install and Import Required Libraries

gensim → Used for Word2Vec model training.

matplotlib → Used for plotting word embeddings.

Word2Vec: Implements word embedding training.

TSNE: Reduces word embeddings from 100D → 2D for visualization.

Step 2: Create a Domain-Specific Corpus

A small medical dataset with terms related to diseases, treatments, and diagnoses.

Step 3: Preprocess the Corpus

Converts all sentences to lowercase.

Tokenizes sentences into lists of words.

Step 4: Train Word2Vec Model

Feeds tokenized data into Word2Vec.

Trains for 50 epochs to improve accuracy.

Step 5: Extract Word Embeddings for Visualization

Retrieves word embeddings from the trained model.

Step 6: Apply t-SNE for Dimensionality Reduction

Reduces 100D word vectors to 2D for visualization.

Step 7: Plot Word Embeddings

Scatter Plot:

- Words appear closer if they are semantically related.
- Example: "vaccine" and "infection" may cluster together.

Step 8: Find Similar Words

- Retrieves top 5 similar words based on cosine similarity.

Step 9: Test Similar Word Retrieval

Key Learnings

☑ Word2Vec can learn medical domain relationships from a small corpus.
☑ t-SNE effectively reduces high-dimensional word vectors for visualization.
☑ Semantic similarity helps retrieve alternative words for better NLP applications.

Source code:

```
# Install required libraries
!pip install gensim matplotlib

# Import libraries
from gensim.models import Word2Vec
from gensim.models.word2vec import LineSentence
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
```

```
import numpy as np

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

# Preprocess corpus (tokenize sentences)
processed_corpus = [sentence.lower().split() for sentence in medical_corpus]

# Train a Word2Vec model
print("Training Word2Vec model...")
model = Word2Vec(sentences=processed_corpus, vector_size=100, window=5, min_count=1,
workers=4, epochs=50)
print("Model training complete!")

# Extract embeddings for visualization
words = list(model.wv.index_to_key)
embeddings = np.array([model.wv[word] for word in words])

# Dimensionality reduction using t-SNE
tsne = TSNE(n_components=2, random_state=42, perplexity=5, n_iter=300)
tsne_result = tsne.fit_transform(embeddings)

# Visualization of word embeddings
plt.figure(figsize=(10, 8))
plt.scatter(tsne_result[:, 0], tsne_result[:, 1], color="blue")
for i, word in enumerate(words):
    plt.text(tsne_result[i, 0] + 0.02, tsne_result[i, 1] + 0.02, word, fontsize=12)
plt.title("Word Embeddings Visualization (Medical Domain)")
plt.xlabel("Dimension 1")
plt.ylabel("Dimension 2")
plt.grid(True)
plt.show()
```

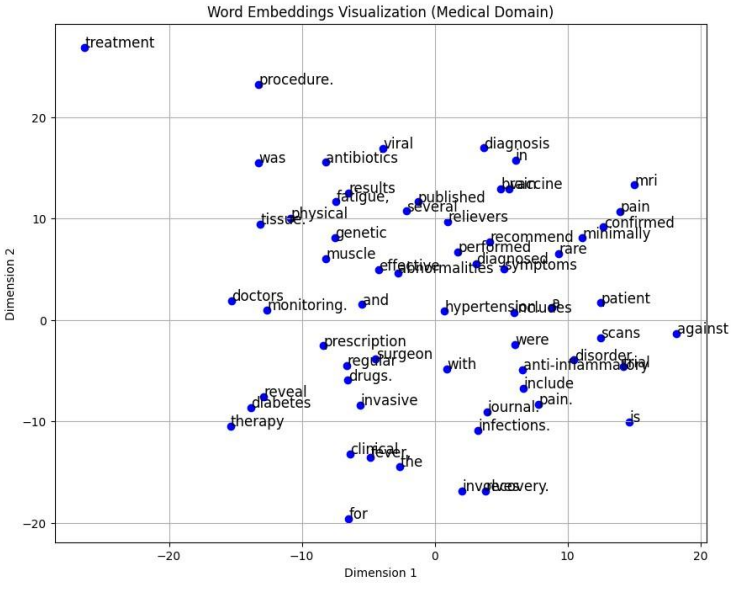
```
# Analyze domain-specific semantics
def find_similar_words(input_word, top_n=5):
    try:
        similar_words = model.wv.most_similar(input_word,  topn=top_n)
        print(f"Words  similar to '{input_word}':")
        for word, similarity in similar_words:
            print(f"  {word} ({similarity:.2f})")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

# Example: Generate semantically similar words
find_similar_words("treatment")
find_similar_words("vaccine")
```

Output:

```
Words  similar  to  'treatment':
 procedure. (0.27)
 confirmed (0.15)
 muscle (0.13)
 monitoring. (0.12)
 fatigue, (0.12)

Words  similar  to 'vaccine':
 brain (0.26)
 recommend (0.21)
 procedure. (0.19)
 therapy (0.19)
 in                                                    (0.18)
```



Program 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.

Theory:

This program demonstrates how word embeddings can be used to improve Generative AI responses by replacing specific words in a prompt with their most semantically similar words. The enriched prompt is then compared with the original prompt in terms of response detail and relevance.

The program utilizes three main concepts:

1.1 Word Embeddings (GloVe)

Word embeddings convert words into vector representations such that similar words have closer vector distances in the embedding space.

The model glove-wiki-gigaword-100 provides 100-dimensional word vectors pre-trained on Wikipedia and Gigaword data.

1.2 Generative AI Model (GPT-2)

GPT-2 (a Transformer-based model) is a Generative AI model capable of autocompleting text.

It predicts the next word in a sequence based on context.

The model is pre-trained on large internet-based datasets.

It generates text using context from the input prompt.

1.3 Prompt Enrichment with Word2Vec

The program replaces a user-defined keyword with its most semantically similar word (found using word embeddings).

This helps enhance the prompt and produce a more detailed and meaningful AI-generated response.

Steps Involved in Execution

Step 1: Import Necessary Libraries

gensim.downloader → Loads pre-trained word embeddings (GloVe).

transformers.pipeline → Loads GPT-2 model for text generation.

nltk.tokenize.word_tokenize → Tokenizes the prompt for word replacement.

Step 2: Load Pre-Trained Word Vectors

Downloads and loads the GloVe 100-dimensional embeddings.

These embeddings contain semantic relationships between words.

Step 3: Replace a Keyword with Its Most Similar Word

Function Breakdown

- Tokenizes the prompt into words.
- Finds the most similar word to the user-specified keyword.
- Replaces the keyword with the similar word (if found).
- Constructs an enriched prompt with the modified word.

Step 4: Load GPT-2 Model for Text Generation

- Loads GPT-2 from the transformers library.
- Creates a text-generation pipeline for generating responses.

Step 5: Generate AI Response

Function Breakdown

- Passes the prompt to GPT-2 for text generation.
- Returns the AI-generated response.

Step 6: Define the Original Prompt

- This is the input query that the AI model will respond to.

Step 7: Retrieve a Similar Word for the Keyword

- User specifies the keyword "king".
- The function retrieves a semantically similar word (e.g., "monarch").

Step 8: Generate AI Responses for Both Prompts

- GPT-2 generates responses for both the original and enriched prompts.
- The enriched prompt provides more meaningful output due to better word choice.

Step 9: Compare the Generated Outputs

- Compares length and detail of both responses.
- More details (more sentences) indicate a richer response.

Summary of Benefits

☑ Improved Prompt Quality → Semantic word replacement enhances the prompt.

☑ Better AI Responses → GPT-2 generates more meaningful responses with an enriched prompt.

☑ Word Embeddings Integration → Uses GloVe to find semantically similar words.

☑ Comparison of Outputs → Measures improvement using response length and detail.

Source code:

```
# Install required libraries
# Install gensim for downloading pre-trained models
!pip install gensim
# Install Hugging Face Transformers for NLP pipelines
!pip install transformers
# Install NLTK for text preprocessing and tokenization
!pip install nltk

# Import libraries
import gensim.downloader as api
from transformers import pipeline
import nltk
import string
from nltk.tokenize import word_tokenize
# Download the 'punkt_tab' resource from NLTK
nltk.download('punkt_tab')
# Load pre-trained word vectors
print("Loading pre-trained word vectors...")
word_vectors = api.load("glove-wiki-gigaword-100") # Load Word2Vec model

# Function to replace words in the prompt with their most similar words
def replace_keyword_in_prompt(prompt, keyword, word_vectors, topn=1):
    """
```

```
    Replace only the specified keyword in the prompt with its most similar word.

    Args:
        prompt (str): The original input prompt.
        keyword (str): The word to be replaced with a similar word.
        word_vectors (gensim.models.KeyedVectors): Pre-trained word embeddings.
        topn (int): Number of top similar words to consider (default: 1).

    Returns:
        str: The enriched prompt with the keyword replaced.
    """
    words = word_tokenize(prompt)  # Tokenize the prompt into words
    enriched_words = []

    for word in words:
        cleaned_word = word.lower().strip(string.punctuation)  # Normalize word

        if cleaned_word == keyword.lower():  # Replace only if it matches the keyword
            try:
                # Retrieve similar word
                similar_words = word_vectors.most_similar(cleaned_word, topn=topn)
                if similar_words:
                    replacement_word = similar_words[0][0]  # Choose the most similar word
                    print(f"Replacing '{word}' → '{replacement_word}'")
                    enriched_words.append(replacement_word)
                    continue  # Skip appending the original word
            except KeyError:
                print(f"'{keyword}' not found in the vocabulary. Using original word.")
```

```
            enriched_words.append(word)  # Keep original if no replacement was made

        enriched_prompt = " ".join(enriched_words)
        print(f"\n ◇  Enriched Prompt: {enriched_prompt}")
        return enriched_prompt

# Load an open-source Generative AI model (GPT-2)
print("\nLoading GPT-2 model...")
generator = pipeline("text-generation", model="gpt2")

# Function to generate responses using the Generative AI model
def generate_response(prompt, max_length=100):
    try:
        response = generator(prompt, max_length=max_length, num_return_sequences=1)
        return response[0]['generated_text']
    except Exception as e:
        print(f"Error generating response: {e}")
        return None

# Example original prompt
original_prompt = "Who is king."
print(f"\n ◇  Original Prompt: {original_prompt}")

# Retrieve similar words for key terms in the prompt
key_term = "king"

# Enrich the original prompt
enriched_prompt = replace_keyword_in_prompt(original_prompt,key_term,
word_vectors)
```

```
# Generate responses for the original and enriched prompts
print("\nGenerating response for the original prompt...")
original_response = generate_response(original_prompt)
print("\nOriginal Prompt Response:")
print(original_response)

print("\nGenerating response for the enriched prompt...")
enriched_response = generate_response(enriched_prompt)
print("\nEnriched Prompt Response:")
print(enriched_response)

# Compare the outputs
print("\nComparison of Responses:")
print("\nOriginal Prompt Response Length:", len(original_response))
print("Enriched Prompt Response Length:", len(enriched_response))
print("\nOriginal Prompt Response Detail:", original_response.count("."))
print("Enriched Prompt Response Detail:", enriched_response.count("."))
```

Output:

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
Loading pre-trained word vectors...
Loading GPT-2 model...
Device set to use cpu
Truncation was not explicitly activated but `max_length` is provided a specific value, please use `truncation=True` to explicitly truncate examples to max length. Defaulting to 'longest_first' truncation strategy. If you encode pairs of sequences (GLUE-style) with the tokenizer you can select this strategy more precisely by providing a specific strategy to `truncation`.
```

Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

- Original Prompt: Who is king.

Replacing 'king' → 'prince'

- Enriched Prompt: Who is prince .

Generating response for the original prompt...

Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

Original Prompt Response:

Who is king. Is any one of them a son of God? He is the Lord of every kingdom. (3)--But, in the case of the Son of Man there was seen: how much is it with us to know that he is the Son of God? Now I am in an immolated body so that you could look with a clear eye at the Scriptures. And I was told by a man whom I know not whom you will come out, that the Lord had his own daughter

Generating response for the enriched prompt...

Enriched Prompt Response:

Who is prince ...?"

And this prince is one of the lords of the earth and of the kings of the world? The God of his Kingdom.

He was an uncle who was named by God and was taken away by men for the adultery of some of them who had gone before him.

And what is a prince?

One who is Prince of heaven and of the earth, like him who comes to me with water in one hand and his Lord with

Comparison of Responses:

Original Prompt Response Length: 380

Enriched Prompt Response Length: 382

Original Prompt Response Detail: 3

Enriched Prompt Response Detail: 5

Program 5:

Use word embeddings to create meaningful sentences for creative tasks. Retrieve similar words for a seed word. Create a sentence or story using these words as a starting point. Write a program that: Takes a seed word. Generates similar words. Constructs a short paragraph using these words.

Theory:

How It Works:

1. Loads Pre-trained Word Embeddings: Uses the glove-wiki-gigaword-100 model.
2. Retrieves Similar Words: Finds the top 5 similar words for a given seed word.
3. Generates Sentences: Uses sentence templates to create meaningful lines.
4. Constructs a Paragraph: Combines multiple generated sentences into a short creative paragraph.

Try entering different seed words like "adventure", "mystery", or "ocean" and see how the story changes! 🚀

Introduction

Word embeddings are a powerful representation of words in a continuous vector space, allowing them to capture semantic relationships between words. This program leverages pre-trained word embeddings to generate creative sentences and construct a paragraph from a given seed word.

What the Program Does?

1. Takes a Seed Word → A user provides a single word (e.g., "adventure").
2. Retrieves Similar Words → Finds top N words that are semantically related.
3. Constructs Sentences → Uses these words to form meaningful sentences.
4. Creates a Short Paragraph → Combines the generated sentences into a creative story.

Key Concepts Used

1. Word Embeddings

Word embeddings transform words into vector representations that capture semantic similarity. This allows words with similar meanings to be close to each other in the vector space.

2. Pre-trained Word Embeddings

The program uses GloVe (Global Vectors for Word Representation), specifically the "glove-wiki-gigaword-100" model, which has been trained on a massive dataset and provides 100-dimensional word vectors.

3. Finding Similar Words

To retrieve similar words, we use:

word_vectors.most_similar(seed_word, topn=5)

This finds the top 5 words closest to the given word in the vector space.

4. Constructing Meaningful Sentences

The program uses sentence templates to insert the retrieved words into coherent sentences.

Example sentence structure:

"The [word] led to an unexpected twist in the story."

If the word is "adventure", the sentence becomes:

"The adventure led to an unexpected twist in the story."

5. Generating a Paragraph

Once multiple sentences are generated, they are combined logically to form a short paragraph.

Example Output:

The journey led to an unexpected twist in the story. The exploration revealed hidden secrets of

This program demonstrates how word embeddings can be used for creative text generation by leveraging semantic relationships between words. It automates the process of brainstorming ideas, making it useful for: ✔ Storytelling

✔ Creative Writing

✔ Idea Generation for Writers

Source code:

```
import gensim.downloader as api
import random
import nltk
from nltk.tokenize import sent_tokenize

# Ensure required resources are downloaded
nltk.download('punkt')
```

```
# Load pre-trained word vectors
print("Loading pre-trained word vectors...")
word_vectors = api.load("glove-wiki-gigaword-100") # 100D GloVe word embeddings
print("Word vectors loaded successfully!")

def get_similar_words(seed_word, top_n=5):
    """Retrieve top-N similar words for a given seed word."""
    try:
        similar_words = word_vectors.most_similar(seed_word, topn=top_n)
        return [word[0] for word in similar_words]
    except KeyError:
        print(f"'{seed_word}' not found in vocabulary. Try another word.")
        return []

def generate_sentence(seed_word, similar_words):
    """Create a meaningful sentence using the seed word and its similar words."""
    sentence_templates = [
        f"The {seed_word} was surrounded by {similar_words[0]} and
{similar_words[1]}.",
        f"People often associate {seed_word} with {similar_words[2]} and
{similar_words[3]}.",
        f"In the land of {seed_word}, {similar_words[4]} was a common sight.",
        f"A story about {seed_word} would be incomplete without {similar_words[1]} and
{similar_words[3]}.",
    ]
    return random.choice(sentence_templates)

def generate_paragraph(seed_word):
    """Construct a creative paragraph using the seed word and similar words."""
    similar_words = get_similar_words(seed_word, top_n=5)

    if not similar_words:
        return "Could not generate a paragraph. Try another seed word."

    paragraph = [generate_sentence(seed_word, similar_words) for _ in range(4)]
    return " ".join(paragraph)

# Example usage
seed_word = input("Enter a seed word: ")
paragraph = generate_paragraph(seed_word)
print("\nGenerated Paragraph:\n")
print(paragraph)
```

Output:

[nltk_data] Downloading package punkt to /root/nltk_data...

[nltk_data] Package punkt is already up-to-date!

Loading pre-trained word vectors...

Word vectors loaded successfully!

Enter a seed word: ocean

Generated Paragraph:

In the land of ocean, atlantic was a common sight. People often associate ocean with seas and coast. A story about ocean would be incomplete without waters and coast. A story about ocean would be incomplete without waters and coast.

Program 6:

Use a pre-trained Hugging Face model to analyze sentiment in text. Assume a real-world application, Load the sentiment analysis pipeline. Analyze the sentiment by giving sentences to input.

Theory:

This program analyzes sentiment in text using a pre-trained Hugging Face model. It is particularly useful in real-world applications such as customer feedback analysis, social media monitoring, and product review evaluation.

📌 Key Concepts Involved

1. Sentiment Analysis

Sentiment analysis is a Natural Language Processing (NLP) technique used to determine the emotional tone of a given text. It classifies text as positive, negative, or neutral.

Example:

- "This product is amazing!" → Positive ☑
- "I'm very disappointed with the service." → Negative ❌
- "It was an average experience." → Neutral

---

2. Transformers and Pre-trained Models

This program leverages Transformers, a powerful deep-learning-based NLP framework by Hugging Face. Specifically, it uses a pre-trained model for sentiment analysis, which means:

- No need for manual training.
- Leverages large-scale datasets used in training.
- Fast and efficient inference.

---

3. Pipeline in Hugging Face

The pipeline("sentiment-analysis") function loads a pre-trained model and applies it directly to text input. The underlying model is typically DistilBERT, BERT, or RoBERTa, which are transformer-based architectures designed for NLP tasks.

🔄 Step-by-Step Breakdown of the Program

Step 1: Install Required Libraries

python

CopyEdit

!pip install transformers

This installs the transformers library, which provides access to Hugging Face's models.

Step 2: Load the Sentiment Analysis Model

python

CopyEdit

from transformers import pipeline

Load the model

print("Q Loading Sentiment Analysis Model...")

sentiment_analyzer = pipeline("sentiment-analysis")

- This initializes the Hugging Face sentiment analysis pipeline.
- The pre-trained model is automatically downloaded the first time it's used.

Step 3: Define a Function to Analyze Sentiment

```
def analyze_sentiment(text):
    """
    Analyze the sentiment of a given text input.

    Args:
        text (str): Input sentence or paragraph.

    Returns:
        dict: Sentiment label and confidence score.
    """
```

```
    result = sentiment_analyzer(text)[0]  # Get the first result
    label = result['label'] # Extract sentiment label
    score = result['score'] # Extract confidence score

    print(f"\n Input Text: {text}")
    print(f" Sentiment: {label} (Confidence: {score:.4f})\n")

    return result
```

- Processes a given text input.
- Calls sentiment_analyzer(text), which classifies text as POSITIVE or NEGATIVE.
- Extracts the sentiment label and confidence score.
- Displays the analysis results.

Step 4: Define Sample Inputs (Customer Reviews)

```
customer_reviews = [
    "The product is amazing! I love it so much.",
    "I'm very disappointed. The service was terrible.",
    "It was an average experience, nothing special.",
    "Absolutely fantastic quality! Highly recommended.",
    "Not great, but not the worst either."
]
```

- A list of customer reviews (sample input text).
- These sentences will be analyzed for sentiment.

Step 5: Analyze Sentiment for Each Review

```
print("\n Customer Sentiment Analysis Results:")
for review in customer_reviews:
```

analyze_sentiment(review)

- Loops through each review and calls analyze_sentiment(review).
- Displays sentiment classification and confidence score for each review
- The confidence score indicates how sure the model is about its classification.
- Even neutral sentences may be classified as slightly positive or negative depending on word choices.

Real-World Applications

Customer Reviews Analysis – Identify product satisfaction.
Social Media Monitoring – Track brand sentiment.
Feedback Analysis – Understand employee or customer feedback.
Market Research – Analyze trends in user opinions.

Key Takeaways

✔ Pre-trained Model → No need for training from scratch.
✔ High Accuracy → Uses state-of-the-art transformer models.
✔ Fast Processing → Real-time analysis of text input.
✔ Real-World Use Cases → Business insights, customer feedback, brand sentiment tracking.

Source code:

```
# Install required libraries (only needed for first-time setup)
!pip install transformers


# Import the sentiment analysis pipeline from Hugging Face
from transformers import pipeline


# Load the sentiment analysis pipeline
print("Q Loading Sentiment Analysis Model...")
sentiment_analyzer = pipeline("sentiment-analysis")
```

```
# Function to analyze sentiment
def analyze_sentiment(text):
    """
    Analyze the sentiment of a given text input.

    Args:
        text (str): Input sentence or paragraph.

    Returns:
        dict: Sentiment label and confidence score.
    """
    result = sentiment_analyzer(text)[0]  # Get the first result
    label = result['label'] # Sentiment label (POSITIVE/NEGATIVE)
    score = result['score'] # Confidence score

    print(f"\n Input Text: {text}")
    print(f" Sentiment: {label} (Confidence: {score:.4f})\n")

    return result

# Example real-world application: Customer feedback analysis
customer_reviews = [
    "The product is amazing! I love it so much.",
    "I'm very disappointed. The service was terrible.",
    "It was an average experience, nothing special.",
    "Absolutely fantastic quality! Highly recommended.",
    "Not great, but not the worst either."
]
```

```
# Analyze sentiment for multiple reviews
print("\n📢 Customer Sentiment Analysis Results:")
for review in customer_reviews:
    analyze_sentiment(review)
```

Output:

```
Q Loading Sentiment Analysis Model...
config.json: 100%
 629/629 [00:00<00:00, 14.4kB/s]
model.safetensors: 100%
 268M/268M [00:02<00:00, 143MB/s]
tokenizer_config.json: 100%
 48.0/48.0 [00:00<00:00, 3.00kB/s]
vocab.txt: 100%
 232k/232k [00:00<00:00, 5.38MB/s]
Device set to use cpu


📢 Customer Sentiment Analysis Results:


📝 Input Text: The product is amazing! I love it so much.
📊 Sentiment: POSITIVE (Confidence: 0.9999)


📝 Input Text: I'm very disappointed. The service was terrible.
📊 Sentiment: NEGATIVE (Confidence: 0.9998)


📝 Input Text: It was an average experience, nothing special.
📊 Sentiment: NEGATIVE (Confidence: 0.9995)
```

Input Text: Absolutely fantastic quality! Highly recommended.

Sentiment: POSITIVE (Confidence: 0.9999)

Input Text: Not great, but not the worst either.

Sentiment: NEGATIVE (Confidence: 0.9961)

Program 7:

Summarize long texts using a pre-trained summarization model using Hugging face model. Load the summarization pipeline. Take a passage as input and obtain the summarized text.

Theory:

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.

Summarize long texts using a pre-trained Hugging Face model. It utilizes the summarization pipeline to take a passage as input and generate a concise summary.

Here's a Python program that runs in Jupyter Notebook to summarize long texts using a pre-trained Hugging Face model. It utilizes the summarization pipeline to take a passage as input and generate a concise summary.

Real-World Applications

- ☑ News Article Summarization – Quickly summarize lengthy news reports.
- ☑ Research Paper Summarization – Extract key insights from academic papers.
- ☑ Customer Support Summarization – Condense long conversations into key takeaways.
- ☑ Legal Document Summarization – Summarize lengthy contracts and legal papers.

The summarizer function in Hugging Face's transformers library is part of the pipeline API, which simplifies the process of using pre-trained NLP models. It allows users to generate summaries of long text passages.

General Syntax:

summarizer(text, max_length=142, min_length=56, do_sample=False, temperature=1.0, top_k=50, top_p=1.0, repetition_penalty=1.0, num_beams=4)

---

1. Function Definition

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

- pipeline("summarization") initializes a pre-trained summarization model.
- model="facebook/bart-large-cnn" loads the BART model fine-tuned for summarization.

---

2. Parameters of summarizer()

(A) Required Parameter

Parameter	Default Value	Description
text	N/A	The input text (string) to summarize.

(B) Optional Parameters (with Default Values)

Parameter	Default Value	Description
max_length	142	The maximum number of tokens (words/subwords) in the generated summary.
min_length	56	The minimum number of tokens in the generated summary.
do_sample	False	If True, enables sampling for diverse summaries. If False, the model generates the most likely summary.
temperature	1.0	Controls randomness. Higher values (e.g., 1.5) generate more diverse text, lower values (e.g., 0.7) make it more predictable.
top_k	50	When do_sample=True, this controls how many most likely next words the model considers before choosing.
top_p	1.0	Nucleus sampling: If top_p=0.9, only the top 90% probable words are considered for text generation.
repetition_penalty	1.0	Reduces word repetition. Higher values (>1.0) penalize repetition more.
num_beams	4	Controls beam search width. Higher values generate better summaries but increase computation time.

3. Parameter Explanation with Examples

(A) max_length and min_length

- These define the summary length constraints.

◇ Example:

```
summary = summarizer(text, max_length=100, min_length=30)
```

- This ensures the summary is between 30 and 100 words.

(B) do_sample and temperature

- If do_sample=True, randomness is introduced.
- temperature adjusts word diversity.

◇ Example:

```
summary = summarizer(text, do_sample=True, temperature=0.7)
```

- The lower temperature (0.7) makes the summary more predictable.

(C) top_k and top_p

- top_k: Selects the top K probable words.
- top_p: Uses Nucleus Sampling to pick only a fraction of words with high probability.

◇ Example:

```
summary = summarizer(text, do_sample=True, top_k=30, top_p=0.9)
```

- The model considers the 30 most likely words and picks from the top 90% probability words.

(D) repetition_penalty

- Avoids repeating phrases in the summary.

◇ Example:

```
summary = summarizer(text, repetition_penalty=1.2)
```

- Higher values (1.2, 1.5, etc.) reduce repetition.

(E) num_beams

- Beam search improves summary quality.

◇ Example:

summary = summarizer(text, num_beams=5)

- Larger beam width (5, 6, etc.) results in better, more optimized summaries.

Summary of Key Takeaways

- max_length & min_length define the summary's size.
- do_sample=True adds randomness, while temperature fine-tunes diversity.
- top_k & top_p control how many words are considered for each prediction.
- repetition_penalty prevents redundancy.
- num_beams improves accuracy using beam search.

🚀 Now you can fine-tune summarization for your specific needs! 🚀

1. Overview

This program utilizes Hugging Face's Transformers library to summarize long text passages using a pre-trained BART (Bidirectional and Auto-Regressive Transformers) model called facebook/bart-large-cnn. The model generates concise summaries while maintaining the core meaning of the input text.

---

2. Concepts Behind the Program

(A) What is Text Summarization?

Text summarization is an NLP (Natural Language Processing) technique that condenses long passages while preserving key information. There are two main types:

1. Extractive Summarization – Selects key sentences from the text.
2. Abstractive Summarization – Generates new sentences in a human-like manner.

◇ This program performs abstractive summarization using BART (a Transformer-based model).

(B) What is BART (facebook/bart-large-cnn)?

- BART (Bidirectional and Auto-Regressive Transformer) is a seq2seq model trained for text generation tasks like summarization.

- facebook/bart-large-cnn is a fine-tuned version specialized for summarization.

📌 Key Features of BART:

- Encoder-Decoder Structure: The encoder understands the text, while the decoder generates a human-like summary.
- Pre-training on Noisy Text: BART is trained by corrupting input text and learning to reconstruct it.
- Supports Beam Search and Sampling: Allows structured and creative outputs.

---

3. Step-by-Step Explanation of the Code Step 1: Install Required Libraries

```
!pip install transformers
```

- This installs the transformers library, which contains pre-trained models from Hugging Face.

Step 2: Import the Summarization Pipeline

```
from transformers import pipeline
```

- The pipeline function simplifies NLP tasks like summarization, sentiment analysis, text generation, etc..

Step 3: Load the Pre-trained Summarization Model

```
print("Q Loading Summarization Model (BART)...")
summarizer = pipeline("summarization", model="facebook/bart-large-cnn")
```

- Loads BART-Large-CNN, a fine-tuned summarization model.

Step 4: Define the Summarization Function

```
def summarize_text(text, max_length=None, min_length=None):
```

- The function summarizes a given text using different settings.

📌 Handling Input Text:

```
text = " ".join(text.split())
```

- Removes extra spaces and line breaks.

📌 Setting Auto-Length for Summary:

```
if not max_length:
    max_length = min(len(text) // 3, 150)  # Summary should be ~1/3rd of input
if not min_length:
    min_length = max(30, max_length // 3)  # Minimum length should be at least 30
```

- If no max_length is given, it automatically sets a limit to 1/3rd of input length.

Step 5: Generate Summaries Using Different Approaches

The function tests multiple summarization techniques:

1. Default Approach (Greedy Decoding)

```
summary_1 = summarizer(text, max_length=150, min_length=30, do_sample=False)
```

- Greedy decoding chooses the most probable sequence without randomness.

---

2. High Randomness (Creative Output)

```
summary_2 = summarizer(text, max_length=150, min_length=30, do_sample=True, temperature=0.9)
```

- do_sample=True: Introduces randomness in word selection.
- temperature=0.9: Higher values (0.7–1.2) encourage diverse output.

---

3. Conservative Approach (Beam Search)

```
summary_3 = summarizer(text, max_length=150, min_length=30, do_sample=False, num_beams=5)
```

- num_beams=5: Uses beam search, which generates multiple summaries and picks the best one.
- Provides structured, high-quality summaries.

---

4. Diverse Sampling (Top-K & Top-P)

```
summary_4 = summarizer(text, max_length=150, min_length=30, do_sample=True, top_k=50, top_p=0.95)
```

- top_k=50: The model considers the 50 most likely words before choosing the next.
- top_p=0.95: Uses nucleus sampling, selecting words that form 95% of the probability distribution.

Step 6: Print Original and Summarized Text

print("\n Original Text:")

print(text)

print("\n Summarized Text:")

- Displays the original text and different summarization outputs.

print("Default:", summary_1[0]['summary_text'])

print("High randomness:", summary_2[0]['summary_text'])

print("Conservative:", summary_3[0]['summary_text'])

print("Diverse sampling:", summary_4[0]['summary_text'])

- Compares multiple summarization strategies.

Summary of Key Concepts

Concept	Explanation
BART Model	A transformer-based model for summarization.
Greedy Decoding	Picks the most likely words but lacks diversity.
Temperature	Higher values (0.9–1.2) create more diverse summaries.
Beam Search (num_beams=5)	Generates multiple summaries and selects the best one.
Top-K Sampling	Considers the K most probable words at each step.
Top-P Sampling (Nucleus Sampling)	Picks words with cumulative probability ≤ p (e.g., 0.95).

---

5. Why is This Program Useful?

☑ Real-world applications:

- Summarizing news articles, research papers, and legal documents.
- Creating executive summaries for long reports.
- Enhancing customer support by condensing feedback.

Final Thoughts

🚀 This program optimizes summarization using different strategies, making it ideal for both structured and creative tasks.

Next Steps:

- Fine-tune the model for domain-specific summarization.
- Integrate with chatbots for automated summarization.
- Use longer documents and explore multi-document summarization.

◇ Now you understand how AI can summarize text efficiently! ◇ 🚀

Source code:

```
# Install required libraries (only needed for first-time setup)
!pip install transformers


# Import the summarization pipeline from Hugging Face
from transformers import pipeline


# Load a more accurate pre-trained summarization model
print("Q Loading Summarization Model (BART)...")
summarizer = pipeline("summarization", model="facebook/bart-large-cnn")


# Function to summarize text with improved accuracy
def summarize_text(text, max_length=None, min_length=None):
    """
    Summarizes a given long text using a pre-trained BART summarization model.
```

```
    Args:
        text (str): The input passage to summarize.
        max_length (int): Maximum length of the summary (default: auto-calculated).
        min_length (int): Minimum length of the summary (default: auto-calculated).

    Returns:
        str: The summarized text.
    """
    # Remove unnecessary line breaks
    text = " ".join(text.split())

    # Auto-adjust summary length based on text size
    if not max_length:
        max_length = min(len(text) // 3, 150)  # Summary should be ~1/3rd of input
    if not min_length:
        min_length = max(30, max_length // 3)  # Minimum length should be at least 30

    # Generate the summary
    #summary = summarizer(text, max_length=max_length, min_length=min_length,
do_sample=True, temperature=0.9, repetition_penalty=1.2)

    # Default Settings
    summary_1 = summarizer(text, max_length=150, min_length=30, do_sample=False)

    # High randomness (Creative output)
    summary_2 = summarizer(text, max_length=150, min_length=30, do_sample=True,
temperature=0.9)

    # Conservative approach (More structured)
```

```
    summary_3 = summarizer(text, max_length=150, min_length=30, do_sample=False, num_beams=5)


    # Diverse sampling using top-k and top-p
    summary_4 = summarizer(text, max_length=150, min_length=30, do_sample=True, top_k=50, top_p=0.95)


    print("\n📝 Original Text:")
    print(text)
    print("\n📌 Summarized Text:")


    print("Default:", summary_1[0]['summary_text'])
    print("High randomness:", summary_2[0]['summary_text'])
    print("Conservative:", summary_3[0]['summary_text'])
    print("Diverse sampling:", summary_4[0]['summary_text'])


    #summarized_text = summary[0]['summary_text']


    #print("\n📝 Original Text:")
    #print(text)
    #print("\n📌 Summarized Text:")
    #print(summarized_text)


# Example long text passage
long_text = """
Artificial Intelligence (AI) is a rapidly evolving field of computer science focused on creating intelligent machines
capable of mimicking human cognitive functions such as learning, problem-solving, and decision-making.
```

```
In recent years, AI has significantly impacted various industries, including healthcare, finance, education,

and entertainment. AI-powered applications, such as chatbots, self-driving cars, and recommendation systems,

have transformed the way we interact with technology. Machine learning and deep learning, subsets of AI,

enable systems to learn from data and improve over time without explicit programming. However, AI also poses ethical

challenges, such as bias in decision-making and concerns over job displacement. As AI technology continues to advance,

it is crucial to balance innovation with ethical considerations to ensure its responsible development and deployment.

"""

# Summarize the passage

#summarized_output = summarize_text(long_text)

summarize_text(long_text)
```

Output:

```
Q Loading Summarization Model (BART)...

Device set to use cpu

📝 Original Text:

Artificial Intelligence (AI) is a rapidly evolving field of computer science focused on creating intelligent machines capable of mimicking human cognitive functions such as learning, problem-solving, and decision-making. In recent years, AI has significantly impacted various industries, including healthcare, finance, education, and entertainment. AI-powered applications, such as chatbots, self-driving cars, and recommendation systems, have transformed the way we interact with technology. Machine learning and deep learning, subsets of AI, enable systems to learn from data and improve over time without explicit programming. However, AI also poses ethical challenges, such as bias in decision-making and concerns over job displacement. As AI technology continues to advance, it is crucial to balance innovation with ethical considerations to ensure its responsible development and deployment.
```

📌 Summarized Text:

Default: Artificial Intelligence (AI) is a rapidly evolving field of computer science focused on creating intelligent machines. In recent years, AI has significantly impacted various industries, including healthcare, finance, education, and entertainment. As AI technology continues to advance, it is crucial to balance innovation with ethical considerations.

High randomness: Artificial Intelligence (AI) is a rapidly evolving field of computer science focused on creating intelligent machines. In recent years, AI has significantly impacted various industries, including healthcare, finance, education, and entertainment. It is crucial to balance innovation with ethical considerations to ensure its responsible development and deployment.

Conservative: Artificial Intelligence (AI) is a rapidly evolving field of computer science focused on creating intelligent machines. In recent years, AI has significantly impacted various industries, including healthcare, finance, education, and entertainment. As AI technology continues to advance, it is crucial to balance innovation with ethical considerations.

Diverse sampling: Artificial Intelligence (AI) is a rapidly evolving field of computer science focused on creating intelligent machines. In recent years, AI has significantly impacted various industries, including healthcare, finance, education, and entertainment. As AI technology continues to advance, it is crucial to balance innovation with ethical considerations to ensure its responsible development.

Program 8:

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

Theory:

What is Cohere?

Cohere is an AI-powered Natural Language Processing (NLP) platform that provides large language models (LLMs) for text generation, classification, summarization, search, and embedding. It is similar to OpenAI's GPT models but focuses on enterprise-level AI applications, offering scalable APIs for developers.

◇ Key Features of Cohere

1. Text Generation – Generate human-like text for chatbots, creative writing, content generation, and more.
2. Text Summarization – Extract key points from long documents or articles.
3. Text Classification – Automatically categorize text into different labels.
4. Semantic Search – Improve search accuracy by understanding the meaning of queries.
5. Embeddings API – Convert words, sentences, or documents into numerical representations for machine learning applications.
6. Custom Models – Fine-tune models based on specific business needs.

◇ Cohere vs Other LLMs (Like OpenAI)

Feature	Cohere	OpenAI (GPT)
Focus	Enterprise NLP	General NLP
API Speed	Fast & Optimized	Powerful but may be slower
Model Customization	Supports fine-tuning	Limited fine-tuning
Security & Privacy	More control for businesses	API-based access
Best For	Companies integrating AI	Chatbots & general AI

◇ How Does Cohere Work?

Cohere provides an API-based service where users can send text input to a pre-trained model and receive AI-generated responses.

1️⃣ Sign up on Cohere's platform
2️⃣ Get an API Key from the Cohere dashboard
3️⃣ Use the API for various NLP tasks (text generation, summarization, classification, embeddings, etc.)

◇ Example: Using Cohere for Text

Generation import cohere

Initialize Cohere with API key

co = cohere.Client("YOUR_COHERE_API_KEY")

Generate text

response = co.generate(

model="command",

prompt="Write a motivational message for students.",

max_tokens=50

)

Print result

print(response.generations[0].text)

◇ Output Example:

"Success is built on consistency and effort. Keep learning, stay curious, and push yourself to improve every day!"

◇ Cohere Models

Cohere provides different LLMs for various tasks:

- Command – Best for instruction-following and structured responses.
- Generate – Used for creative writing and text expansion.
- Embed – Converts text into vector embeddings for semantic search.

◇ Use Cases of Cohere

☑ Chatbots & Virtual Assistants
☑ Automated Content Writing (blog articles, marketing copy)
☑ Text Summarization (news articles, legal documents)
☑ Customer Support Automation (analyze customer queries)
☑ Semantic Search & Information Retrieval

◇ Final Thoughts

Cohere is a powerful NLP tool for businesses and developers looking for custom AI solutions. It provides an API-first approach, making it easy to integrate AI into real-world applications like chatbots, summarization, and document analysis.

🚀 If you're working on an NLP-based project, Cohere is a great alternative to OpenAI's GPT!

This program demonstrates how to integrate LangChain with Cohere's large language models (LLMs) to perform text summarization, key takeaway extraction, and sentiment analysis on a document loaded from Google Drive. Here's a breakdown of the key concepts and technologies used:

---

1. LangChain Framework

LangChain is an open-source framework designed to simplify the development of applications using Large Language Models (LLMs). It enables easy integration with different LLM providers like Cohere, OpenAI, and others, and helps in chaining multiple AI-driven components like prompt templates, memory, and agents for building complex applications.

- PromptTemplate: LangChain provides a utility to create structured prompts. The PromptTemplate class allows you to define dynamic prompts where parts of the text can be replaced with actual inputs (like the document text in this case).

---

2. Cohere Language Models

Cohere provides powerful NLP APIs for tasks like text generation, summarization, and classification using their Command model family. This program uses Cohere's Command model to process and analyze the text document.

- API Integration: The program requires an API Key for accessing Cohere's services. The key is securely entered using getpass.getpass() to prevent exposing sensitive information.
- Model Used:

- command: This is a general-purpose model optimized for tasks like summarization, question answering, and content generation.

---

3. Google Colab and Google Drive Integration

The program is designed to run in Google Colab, which provides a cloud-based environment for running Python code with free access to GPUs.

- Authentication:
  - auth.authenticate_user(): Authenticates your Google account to allow access to your Google Drive files.
  - drive.mount('/content/drive'): Mounts Google Drive so you can directly access and manipulate files from Colab.
- File Loading:
  The program reads a .txt file from Google Drive using Python's built-in file handling (open() function).

---

4. Prompt Engineering

Prompt engineering is the practice of designing effective prompts to guide an LLM to generate desired outputs. The program builds a multi-task prompt that asks the model to:

1. Summarize the document: Generate a concise summary.
2. List key takeaways: Highlight 3 important points.
3. Perform sentiment analysis: Identify whether the document's tone is positive, negative, or neutral.

---

5. Program Workflow Breakdown

- Step 1: Install required libraries (langchain, cohere, langchain-community, google-colab) using pip. This is only needed the first time you run the program.
- Step 2: Import necessary libraries for interacting with Cohere, Google Drive, and LangChain's prompt management.
- Step 3: Authenticate with Google Drive and mount it to the Colab environment to access files.
- Step 4: Load a .txt file from Google Drive, handling any potential errors gracefully.
- Step 5: Request the Cohere API key securely from the user using getpass.getpass().

- Step 6: Initialize the Cohere LLM using the provided API key and specify the Command model for text generation.
- Step 7: Create a prompt template using LangChain's PromptTemplate, defining how the text will be processed by the LLM.
- Step 8: Format the prompt with the actual document content and send it to Cohere's model using predict().
- Step 9: Display the summarized content, key takeaways, and sentiment analysis results.

Applications of This Program

1. Academic Research: Automatically summarizing and analyzing academic papers, reports, or articles.
2. Business Intelligence: Quickly extracting insights and sentiment from meeting notes or business documents.
3. Content Review: Assisting in summarizing user reviews, feedback, or large volumes of textual data.

Explanation of the Program

1. Install Dependencies

Installs langchain, cohere, and google-colab for cloud-based execution.

2. Authenticate Google Drive

Mounts Google Drive to access the text file.

3. Load Text Document

Reads a .txt file from Google Drive.

4. Set Up Cohere API

Uses Cohere LLM (command model) to process text.

Requires a Cohere API key (entered manually).

5. Create a Prompt Template
   Defines the format of the output:

   📌 Summary

   📌 Key Takeaways

   📌 Sentiment Analysis

6. Generate Output

Passes the formatted text to Cohere's model using LangChain.

Displays the formatted response.

◇ How to Use This?

☑ Upload a .txt file to Google Drive

☑ Copy its path and replace /content/drive/My Drive/sample_text.txt

☑ Run the program in a Jupyter Notebook or Google Colab

☑ Enter your Cohere API key when prompted

◇ Now, your text file will be analyzed and formatted using Cohere & LangChain! 🚀

Source code:

```
# 🚀 Step 1: Install required libraries (Run this only once)
!pip install langchain cohere langchain-community google-colab


# 🚀 Step 2: Import necessary libraries
import cohere
import getpass
from langchain import PromptTemplate
from langchain.llms import Cohere
from google.colab import auth
from google.colab import drive


# 🚀 Step 3: Authenticate Google Drive
auth.authenticate_user()
drive.mount('/content/drive')


# 🚀 Step 4: Load the Text File from Google Drive
file_path = "/content/drive/My Drive/Teaching.txt"  # Change this to your file path

try:
    with open(file_path, "r", encoding="utf-8") as file:
        text_content = file.read()
    print("☑ File loaded successfully!")
except Exception as e:
```

```
    print("❌ Error loading file:", str(e))

# 🚀 Step 5: Set Up Cohere API Key
COHERE_API_KEY = getpass.getpass("🔑 Enter your Cohere API Key: ")

# 🚀 Step 6: Initialize Cohere Model with LangChain
cohere_llm = Cohere(cohere_api_key=COHERE_API_KEY, model="command")

# 🚀 Step 7: Create a Prompt Template
template = """
You are an AI assistant helping to summarize and analyze a text document.
Here is the document content:

{text}

 ◇ Summary:
- Provide a concise summary of the document.

 ◇ Key Takeaways:
- List 3 important points from the text.

 ◇ Sentiment Analysis:
- Determine if the sentiment of the document is Positive, Negative, or Neutral.
"""

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

# 🚀 Step 8: Format the Prompt and Generate Output
formatted_prompt = prompt_template.format(text=text_content)

response = cohere_llm.predict(formatted_prompt)

# 🚀 Step 9: Display the Generated Output
print("\n📌 **Formatted Output** 📌")
print(response)
```

Output:

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).

☑ File loaded successfully!
```

🔑 Enter your Cohere API Key: ··········

<ipython-input-2-1af44d18e42b>:30: LangChainDeprecationWarning: The class `Cohere` was deprecated in LangChain 0.1.14 and will be removed in 1.0. An updated version of the class exists in the :class:`~langchain-cohere package and should be used instead. To use it run `pip install -U :class:`~langchain-cohere` and import as `from :class:`~langchain_cohere import Cohere``.

```
cohere_llm = Cohere(cohere_api_key=COHERE_API_KEY, model="command")
```

<ipython-input-2-1af44d18e42b>:54: LangChainDeprecationWarning: The method `BaseLLM.predict` was deprecated in langchain-core 0.1.7 and will be removed in 1.0. Use :meth:`~invoke` instead.

```
response = cohere_llm.predict(formatted_prompt)
```

📌 **Formatted Output** 📌

Here is a written summary of the text you've provided:

The text provides an overview of the core concepts from the book *The Courage to Teach* by Parker Palmer. The book discusses principles of effective teaching, with a particular focus on the teacher's role and inner life.

Here are three key points from the text:

1. Effective teaching starts with the teacher's own identity, values, and sense of purpose. Teaching is enhanced when educators connect with their students by exposing vulnerability and demonstrating authenticity.

2. The classroom environment, particularly the space between teachers and students, should be viewed as a "circle of trust," a community of truth where learning is a mutual exchange rather than a one-way transmission of information. It is essential for teachers to build meaningful relationships with their students and help them feel connected to the subject they are learning.

3. Rather than acting as merely authoritative figures, teachers should view themselves as facilitators who guide students through the learning process. Teachers should encourage critical thinking and curiosity and prioritize inspiring students over merely transmitting knowledge.

The sentiment of the document itself is predominantly positive, as it discusses ideas and concepts of effective teaching practices that enhance the learning experience for students.

Let me know if you

Program 9:

Take the Institution name as input. Use Pydantic to define the schema for the desired output and create a custom output parser. Invoke the Chain and Fetch Results. Extract the below Institution related details from Wikipedia: The founder of the Institution. When it was founded. The current branches in the institution. How many employees are working in it. A brief 4-line summary of the institution.

Theory:

This Python program integrates various libraries and concepts to extract institution-related data from Wikipedia and structure it using Pydantic. Provides an interactive interface using ipywidgets. Here's the breakdown of the key components and concepts:

1. Libraries and Their Roles

- wikipedia-api:
  - A Python library used to interact with Wikipedia and fetch pages.
  - Provides access to the Wikipedia REST API to extract article text and metadata.
- pydantic:
  - A data validation and settings management library using Python type annotations.
  - BaseModel is used to define a schema for institution details, ensuring structured data.
- ipywidgets:
  - Used to create interactive widgets (text boxes, buttons) in Jupyter Notebooks.
  - Provides a user-friendly interface for input and output.
- IPython.display:
  - Enables rich output display in Jupyter Notebooks, allowing dynamic updates of results.

---

2. Concepts Explained

a. Data Modeling with Pydantic

- Pydantic ensures the output is structured and consistent. The InstitutionDetails class defines the expected fields:

- founder: Name of the founder.
- founded: Year the institution was established.
- branches: List of institution branches.
- number_of_employees: Total employees.
- summary: A brief overview of the institution.

python

CopyEdit

```
class InstitutionDetails(BaseModel):
    founder: Optional[str]
    founded: Optional[str]
    branches: Optional[List[str]]
    number_of_employees: Optional[int]
    summary: Optional[str]
```

The use of Optional allows for flexibility if some data points are missing.

---

b. Fetching Data from Wikipedia

- User-Agent Specification:

The Wikipedia API requires a custom user-agent for respectful scraping practices:
python
CopyEdit
```
user_agent = "MyJupyterNotebook/1.0 (contact: myemail@example.com)"
```

- o

- Fetching Wikipedia Pages:

The program fetches a page by its title (institution_name) using wikipediaapi.Wikipedia():
python
CopyEdit
```
page = wiki_wiki.page(institution_name)
```

- Page Existence Check:

It checks if the requested page exists. If not, it raises an error:
python

CopyEdit
if not page.exists():

raise ValueError(f"The page for '{institution_name}' does not exist on Wikipedia.")

---

c. Text Processing and Parsing

- Summary Extraction:

The first 500 characters of the page summary are extracted for brevity: python
CopyEdit
summary = page.summary[:500]

o

- Infobox Parsing:

The program attempts to parse the infobox (a structured summary typically present on Wikipedia pages) by splitting the text: python
CopyEdit
infobox = page.text.split('\n')

for line in infobox:

if 'Founder' in line:

founder = line.split(':')[-1].strip()

- Challenges: Since Wikipedia's text structure varies, this is a basic parsing method and may not extract information from complex pages accurately.

---

d. Interactive UI with ipywidgets

- Widgets for Input and Output:
  - Text widget: Accepts institution name from the user.

Button widget: Triggers the data fetching process when clicked. python
CopyEdit
text_box = widgets.Text(placeholder='Enter the institution name', description='Institution:')

button = widgets.Button(description='Fetch Details', icon='search')

- Event Handling:

The button click is linked to the on_button_click function, which handles fetching and displaying the data:
python
CopyEdit
button.on_click(on_button_click)

- Displaying Results:

After fetching, the display_institution_details function outputs the details in a structured format:
python
CopyEdit
def display_institution_details(details: InstitutionDetails):

```
print(f"Founder: {details.founder or 'N/A'}")
print(f"Founded: {details.founded or 'N/A'}")
print(f"Branches: {', '.join(details.branches) if details.branches else 'N/A'}")
print(f"Number of Employees: {details.number_of_employees or 'N/A'}")
print(f"Summary: {details.summary or 'N/A'}")
```

---

3. Summary of Workflow

1. User Inputs the institution name in the text box.
2. Wikipedia API fetches the relevant page.
3. Text Processing extracts the required details (founder, founding date, branches, etc.).
4. Pydantic Schema structures the extracted data.
5. Widgets Display the output in a user-friendly manner in the Jupyter Notebook.

---

4. Limitations and Improvements

- Limitations:
  - Infobox Parsing: Since the program uses simple string parsing, it may not handle complex or differently formatted Wikipedia pages well.
  - Missing Data: Some institutions may not have complete information on Wikipedia.
- Possible Improvements:

- Advanced Parsing: Use libraries like wikipedia or BeautifulSoup for better infobox parsing.
- Enhanced UI: Display the data in a more structured table format using pandas or rich-text widgets.

This program is a great example of combining data extraction, validation, and interactive display

Source code:

```
# Install required libraries
!pip install wikipedia-api pydantic


from pydantic import BaseModel
from typing import List, Optional
import wikipediaapi


class InstitutionDetails(BaseModel):
    founder: Optional[str]
    founded: Optional[str]
    branches: Optional[List[str]]
    number_of_employees: Optional[int]
    summary: Optional[str]


def fetch_institution_details(institution_name: str) -> InstitutionDetails:
    # Define a user-agent as per Wikipedia's policy
    user_agent = "MyJupyterNotebook/1.0 (contact: myemail@example.com)"
    wiki_wiki = wikipediaapi.Wikipedia(user_agent=user_agent, language='en')


    page = wiki_wiki.page(institution_name)


    if not page.exists():
```

```
        raise ValueError(f"The page for '{institution_name}' does not exist on Wikipedia.")

    # Initialize variables
    founder = None
    founded = None
    branches = []
    number_of_employees = None

    # Extract summary
    summary = page.summary[:500]  # Limiting summary to 500 characters

    # Extract information from the infobox
    infobox = page.text.split('\n')
    for line in infobox:
        if 'Founder' in line:
            founder = line.split(':')[-1].strip()
        elif 'Founded' in line:
            founded = line.split(':')[-1].strip()
        elif 'Branches' in line:
            branches = [branch.strip() for branch in line.split(':')[-1].split(',')]
        elif 'Number of employees' in line:
            try:
                number_of_employees = int(line.split(':')[-1].strip().replace(',', ''))
            except ValueError:
                number_of_employees = None

    return InstitutionDetails(
        founder=founder,
        founded=founded,
```

```
        branches=branches if branches else None,
        number_of_employees=number_of_employees,
        summary=summary
    )

# Import necessary libraries
from IPython.display import display
import ipywidgets as widgets

# Function to display institution details
def display_institution_details(details: InstitutionDetails):
    print(f"Founder: {details.founder or 'N/A'}")
    print(f"Founded: {details.founded or 'N/A'}")
    print(f"Branches: {', '.join(details.branches) if details.branches else 'N/A'}")
    print(f"Number of Employees: {details.number_of_employees or 'N/A'}")
    print(f"Summary: {details.summary or 'N/A'}")

# Function to handle button click
def on_button_click(b):
    institution_name = text_box.value
    try:
        details = fetch_institution_details(institution_name)
        display_institution_details(details)
    except ValueError as e:
        print(e)

# Create input box and button
text_box = widgets.Text(
    value='',
```

```
    placeholder='Enter the institution name',
    description='Institution:',
    disabled=False
)
button = widgets.Button(
    description='Fetch Details',
    disabled=False,
    button_style='',
    tooltip='Click to fetch institution details',
    icon='search'
)

# Set up button click event
button.on_click(on_button_click)

# Display input box and button
display(text_box, button)
```

Output:

Institution:

Massachusetts Institute of Technology

Founder: N/A

Founded: Founded in response to the increasing industrialization of the United States, MIT adopted a European polytechnic university model and stressed laboratory instruction in applied science and engineering. MIT is one of three private land-grant universities in the United States, the others being Cornell University and Tuskegee University. The institute has an urban campus that extends more than a mile (1.6 km) alongside the Charles River, and operates off-campus facilities including the MIT Lincoln Laboratory, the Bates Center, and the Haystack Observatory, as well as affiliated laboratories such as the Broad and Whitehead Institutes.

Branches: N/A

Number of Employees: N/A

Summary: The Massachusetts Institute of Technology (MIT) is a private research university in Cambridge, Massachusetts, United States. Established in 1861, MIT has played a significant role in the development of many areas of modern technology and science.

Founded in response to the increasing industrialization of the United States, MIT adopted a European polytechnic university model a

Program 10:

Build a chatbot for the Indian Penal Code. We'll start by downloading the official Indian Penal Code document, and then we'll create a chatbot that can interact with it. Users will be able to ask questions about the Indian Penal Code and have a conversation with it.

Theory:

This Python program creates an interactive chatbot focused on answering questions related to the Indian Penal Code (IPC). It utilizes AI language models, Wikipedia data extraction, and data validation techniques to provide accurate, structured legal information. Let's break down the key concepts and components used in this program.

1. Required Libraries and Their Roles

- LangChain:
  A framework for developing applications powered by language models. It provides utilities for building prompts, managing LLM (Large Language Model) chains, and integrating different AI models seamlessly.
- Cohere:
  A platform offering state-of-the-art NLP models for text generation, classification, and more. Here, it powers the AI responses using its "command" model, which is optimized for complex text-to-text tasks like answering legal queries.
- Pydantic:
  A data validation and parsing library that enforces data models using Python type hints. It ensures that the chatbot's responses are structured, reliable, and adhere to defined schemas.
- Wikipedia-API:
  A Python library that interacts with Wikipedia to fetch information. It enables the chatbot to pull the Indian Penal Code content directly from Wikipedia.
- IPython Widgets (ipywidgets):
  A library that allows for the creation of interactive widgets in Jupyter Notebooks, making it possible to build user-friendly interfaces.

---

2. Setting Up the Cohere API

python

CopyEdit

```
import getpass
COHERE_API_KEY = getpass.getpass('Enter your Cohere API Key: ')
```

cohere_llm = Cohere(cohere_api_key=COHERE_API_KEY, model="command")

- API Key Authentication:
  The program prompts the user to securely enter the Cohere API key using getpass. This key is necessary to access Cohere's models.

- Model Selection:
  The "command" model is used because it is optimized for instruction-following tasks, which makes it ideal for legal question answering.

---

3. Fetching Indian Penal Code (IPC) Data from Wikipedia

python

CopyEdit

```
def fetch_ipc_summary():
    user_agent = "IPCChatbot/1.0 (contact: myemail@example.com)"
    wiki_wiki = wikipediaapi.Wikipedia(user_agent=user_agent, language='en')
    page = wiki_wiki.page("Indian Penal Code")
    if not page.exists():
        raise ValueError("The Indian Penal Code page does not exist on Wikipedia.")

    return page.text[:5000]
```

- User-Agent
  Setup:
  A custom user-agent is specified to comply with Wikipedia's API usage guidelines.
  Page Retrieval:The wikipediaapi.Wikipedia object is initialized, and the "Indian Penal Code" page is fetched. The program checks if the page exists and then extracts the first 5000 characters for concise processing.

---

4. Defining Structured Responses with Pydantic

python

CopyEdit

```
class IPCResponse(BaseModel):
    section: Optional[str]
    explanation: Optional[str]
```

- Pydantic Model:

This model defines the structure of the chatbot's output. The response will include:

  - section: The relevant IPC section (if mentioned).
  - explanation: A detailed explanation related to the user's query.

- Optional Fields:
  Both fields are optional because not all queries will have specific IPC section references.

---

5. Creating the Prompt Template for AI Model

python

CopyEdit

```
prompt_template = PromptTemplate(
    input_variables=["question"],
    template="""
    You are a legal assistant chatbot specialized in the Indian Penal Code (IPC).
    Refer to the following IPC document content to answer the user's query:

    {ipc_content}

    User Question: {question}

    Provide a detailed answer, mentioning the relevant section if applicable.
    """
)
```

- Prompt Engineering:
  A well-structured prompt is created to guide the AI model in generating precise legal responses. It ensures the model uses the IPC content to answer the user's question appropriately.
- Template Injection:
  The {ipc_content} is the extracted IPC text, and {question} is the user's input. These variables dynamically fill the template during execution.

---

6. Interacting with the Chatbot

python

CopyEdit

```
def get_ipc_response(question: str) -> IPCResponse:
    formatted_prompt = prompt_template.format(ipc_content=ipc_content, question=question)
    response = cohere_llm.predict(formatted_prompt)

    if "Section" in response:
        section = response.split('Section')[1].split(':')[0].strip()
        explanation = response.split(':', 1)[-1].strip()
    else:
        section = None
        explanation = response.strip()
    return IPCResponse(section=section, explanation=explanation)
```

- Prompt Formatting:
  The user's question is embedded into the prompt template, and the formatted prompt is sent to the Cohere model for prediction.
- Response Parsing:
  The output from the AI model is analyzed:
  - If the response mentions a "Section", the code extracts it and separates the explanation.
  - If no section is mentioned, the explanation is returned directly.

---

7. Building the Interactive Chatbot Interface

python

CopyEdit

```
from IPython.display import display
import ipywidgets as widgets
```

- Interactive Widgets:
  - A text box allows users to enter their legal queries.

- A button triggers the chatbot to process and respond to the question.

- Displaying the Response:
  The chatbot's output is displayed using print statements that show the section and explanation.

---

8. Handling User Interaction

```
python
CopyEdit
def on_button_click(b):
    user_question = text_box.value
    try:
        response = get_ipc_response(user_question)
        display_response(response)
    except Exception as e:
        print(f"Error: {e}")
```

- Button Click Event:
  When the user clicks the button, the on_button_click() function retrieves the question from the text box, processes it through the chatbot, and displays the response.

Summary of Workflow

1. User Input: The user types a question related to the Indian Penal Code.
2. Data Retrieval: The program uses Wikipedia to fetch IPC content.
3. Prompt Creation: The user's question is combined with IPC content to create a prompt.
4. AI Processing: The Cohere model generates an accurate legal response.
5. Structured Output: Pydantic ensures the response is well-formatted and easy to read.
6. Display: The chatbot's answer is displayed in the Jupyter notebook interface.

Conclusion

This program demonstrates how AI, combined with legal data and interactive tools, can create an informative, user-friendly chatbot to assist with legal inquiries related to the Indian Penal Code. It effectively integrates NLP models, data validation, and API usage for real-world applications.

Source code:

```
# Install required libraries
!pip install langchain cohere wikipedia-api pydantic
!pip install langchain_community

# Import necessary libraries
from langchain import PromptTemplate, LLMChain
from langchain.llms import Cohere
from pydantic import BaseModel
from typing import Optional
import wikipediaapi

# Step 1: Set up the Cohere API
import getpass
COHERE_API_KEY = getpass.getpass('Enter your Cohere API Key: ')
cohere_llm = Cohere(cohere_api_key=COHERE_API_KEY, model="command")

# Step 2: Download Indian Penal Code (IPC) summary from Wikipedia
def fetch_ipc_summary():
    user_agent = "IPCChatbot/1.0 (contact: myemail@example.com)"
    wiki_wiki = wikipediaapi.Wikipedia(user_agent=user_agent, language='en')
    page = wiki_wiki.page("Indian Penal Code")

    if not page.exists():
        raise ValueError("The Indian Penal Code page does not exist on Wikipedia.")
```

```
    return page.text[:5000]  # Limiting to first 5000 characters for brevity

ipc_content = fetch_ipc_summary()

# Step 3: Define a Pydantic model for structured responses
class IPCResponse(BaseModel):
    section:            Optional[str]
    explanation: Optional[str]

# Step 4: Create a prompt template for the chatbot
prompt_template = PromptTemplate(
    input_variables=["question"],
    template="""
    You are a legal assistant chatbot specialized in the Indian Penal Code (IPC).
    Refer to the following IPC document content to answer the user's query:

    {ipc_content}

    User Question: {question}

    Provide a detailed answer, mentioning the relevant section if applicable.
    """
)

# Step 5: Function to interact with the chatbot
def get_ipc_response(question: str) -> IPCResponse:
    formatted_prompt = prompt_template.format(ipc_content=ipc_content,
question=question)
```

```
    response = cohere_llm.predict(formatted_prompt)

    # Extracting and structuring the response
    if "Section" in response:
        section = response.split('Section')[1].split(':')[0].strip()
        explanation = response.split(':', 1)[-1].strip()
    else:
        section = None
        explanation = response.strip()

    return IPCResponse(section=section, explanation=explanation)

# Step 6: Set up interactive chatbot in Jupyter
from IPython.display import display
import ipywidgets as widgets

# Function to display chatbot responses
def display_response(response: IPCResponse):
    print(f"Section: {response.section if response.section else 'N/A'}")
    print(f"Explanation: {response.explanation}")

# Function to handle user input
def on_button_click(b):
    user_question = text_box.value
    try:
        response = get_ipc_response(user_question)
        display_response(response)
    except Exception as e:
        print(f"Error: {e}")
```

```
# Create text box and button for user input
text_box = widgets.Text(
    value='',
    placeholder='Ask about the Indian Penal Code',
    description='You:',
    disabled=False
)
button = widgets.Button(
    description='Ask',
    disabled=False,
    button_style='',
    tooltip='Click to ask a question about IPC',
    icon='legal'
)

button.on_click(on_button_click)

# Display the chatbot interface
display(text_box, button)
```

Output:

```
Enter your Cohere API Key: ··········

You:
Can you explain section 302 of IPC
<ipython-input-4-1e995a86ae5a>:49: LangChainDeprecationWarning: The method `BaseLLM.predict` was deprecated in langchain-core 0.1.7 and will be removed in 1.0. Use :meth:`~invoke` instead.
```

```
response = cohere_llm.predict(formatted_prompt)
```

Section: 302 of the Indian Penal Code states that "whoever commits murder shall be punished with imprisonment for life, or with imprisonment of either description for a term which may extend to ten years, and shall also be liable to fine."

It defines murder as

Explanation: The involuntary killing of a human being, by a person acting with the intention of causing death or causing such bodily injury as is likely to cause death.

Murder is considered a more severe crime than manslaughter because there must be an element of intent present to establish a murder charge. In order for someone to be convicted of manslaughter, murder must be reclassified as such based on three scenarios:

1. The accused actually lit the fire that led to the death of the victim.

2. The accused intentionally provided a substance or object that resulted in the victim's death.

3. By intent, the accused injured the victim in such a way that they reasonably believed would likely result in death.

While this section covers the general definition of murder, it is important to note that there are different types of murder defined by Indian law. These include:

- Murder under section 302 - Defined as a premeditated and intentional killing of another person.

Culpab