Code explanation summary

NLP2.ipynb
File

explain code

This notebook is designed to demonstrate basic Natural Language Processing (NLP) techniques using Python. Here's a breakdown of the first few code cells:

Cell 1: Install Required Libraries

```python
pip install numpy pandas scikit-learn gensim
```

This ensures the required libraries are installed:

- **`numpy`**, **`pandas`**: for numerical and tabular data handling.
- **`scikit-learn`**: for vectorizing text (CountVectorizer, TfidfVectorizer).
- **`gensim`**: for Word2Vec modeling.

Cell 2: Imports and User Input

```python
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from gensim.models import Word2Vec
```

These import the needed modules. Then, a function gathers user input:

```python
def get_user_input():
    print("Enter your sentences (type 'exit' to finish):")
    documents = []
    while True:
        sentence = input()
        if sentence.lower() == 'exit':
            break
        documents.append(sentence)
    return documents
```

After collecting input, it checks whether any documents were entered.

Bag-of-Words and TF-IDF Vectorization

The next part processes the text with different vectorization techniques:

CountVectorizer

```python
count_vectorizer = CountVectorizer()
count_matrix = count_vectorizer.fit_transform(documents)
```

Creates a word frequency matrix (Bag-of-Words model).

Normalized Word Frequencies

```python
normalized_count_df = count_df.div(count_df.sum(axis=1), axis=0)
```

Shows how often words appear relative to the total words in each sentence.

TF-IDF Vectorization

```python
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)
```

Transforms the text to a matrix based on Term Frequency-Inverse Document Frequency, highlighting words unique to each sentence.

Word2Vec Embeddings

```python
tokenized_docs = [doc.lower().split() for doc in documents]
word2vec_model = Word2Vec(sentences=tokenized_docs, vector_size=100, window=5, min_count=1, workers=
```

This trains a Word2Vec model from scratch on your sentences, where:

- `vector_size`: dimensionality of the word embeddings
- `window`: context window size
- `min_count`: ignore words with total frequency lower than this
- `workers`: parallelism level

It then allows the user to input a word and returns:

- Its vector representation
- A list of similar words based on cosine similarity

Would you like me to summarize the rest of the notebook as well?

write viva questions with answers

Here are some **viva questions with answers** based on your NLP notebook, which includes CountVectorizer, TF-IDF, and Word2Vec:

🔹 1. What is the purpose of this project?

Answer:
The project demonstrates basic NLP techniques such as Bag-of-Words (BoW), TF-IDF, and Word2Vec on user-input sentences. It shows how text can be converted into numerical features for further processing in machine learning models.

🔹 2. What is CountVectorizer and how does it work?

Answer:
`CountVectorizer` converts a collection of text documents into a matrix of token counts. It simply counts the number of times each word appears in a document (BoW model), without considering the importance of the word.

🔹 3. What is TF-IDF and why is it better than BoW in some cases?

Answer:
`TF-IDF` stands for Term Frequency–Inverse Document Frequency. It adjusts the raw frequency of a word by how common it is across all documents. Words that are frequent in one document but rare in others get higher scores, making it better at highlighting unique, informative words.

🔹 4. What is Word2Vec and how is it different from TF-IDF?

Answer:
`Word2Vec` is a neural network model that creates word embeddings — dense vector representations capturing semantic relationships. Unlike TF-IDF, which is sparse and based on frequency, Word2Vec captures context and similarity between words (e.g., "king" and "queen" are close in vector space).

🔹 5. What are the main parameters in the Word2Vec model?

Answer:

- `vector_size`: Dimension of the word embeddings.
- `window`: Number of context words considered to the left and right.
- `min_count`: Ignores words with total frequency less than this.
- `workers`: Number of threads for training.

🔹 6. How is the input text preprocessed before training Word2Vec?

Answer:
The input sentences are lowercased and split into words (tokenized) using Python's `` `split()` `` method, resulting in a list of token lists suitable for training Word2Vec.

🔹 7. What kind of output do you get from the Word2Vec model?

Answer:
You can extract:

- The vector representation of any word in the vocabulary.
- A list of most similar words to a given word, based on cosine similarity.

🔹 8. Can you explain the difference between sparse and dense vector representations?

Answer:

- **Sparse vectors** (like BoW and TF-IDF) are mostly zeros and represent word counts.
- **Dense vectors** (like Word2Vec) contain real numbers that encode semantic information, making them more efficient and meaningful for deep learning.

🔹 9. What happens if you input a word that's not in the Word2Vec vocabulary?

Answer:
The model will not be able to return a vector or find similar words because it hasn't learned anything about that word during training.

🔹 10. What are the limitations of this approach?

Answer:

- It only works well on small inputs and may not generalize.
- No preprocessing beyond basic tokenization (e.g., no stemming or stopword removal).
- Word2Vec training is based on user input, so vocabulary is very limited.

Would you like a printable PDF of these Q&A as well?

give more advance questions

Here are some **advanced viva questions with answers** related to your NLP project, focusing on deeper understanding and real-world implications of the techniques used:

🔹 1. Why is normalization used in Bag-of-Words models, and how is it implemented here?

Answer:
Normalization ensures each document's word frequencies are on the same scale, making comparisons fair across documents of different lengths. In this project, it is done using:

```python
normalized_count_df = count_df.div(count_df.sum(axis=1), axis=0)
```

This divides each word count by the total word count in that document (row-wise normalization).

🔹 2. How does Word2Vec use the skip-gram or CBOW architecture, and which one is better?

Answer:
Word2Vec has two training methods:

- **CBOW (Continuous Bag of Words):** Predicts a word based on its context.
- **Skip-gram:** Predicts the context based on a word.

`` `gensim.Word2Vec` `` uses skip-gram by default when `` `sg=1` ``. Skip-gram performs better with smaller datasets and captures rare word relationships more effectively.

🔹 3. How do TF-IDF and Word2Vec handle the issue of word semantics differently?

Answer:

- **TF-IDF** treats words independently and doesn't capture meaning or word order.
- **Word2Vec** learns from context, capturing semantic similarity (e.g., "doctor" and "nurse" might be close).

Word2Vec embeddings group semantically similar words together, which TF-IDF cannot do.

🔹 4. What are the limitations of using Word2Vec on small datasets like user-input sentences?

Answer:
Word2Vec needs large corpora to learn meaningful relationships. On small datasets:

- Vocabulary is too limited.
- Vectors may not generalize.
- Similarity and analogy tasks are inaccurate due to insufficient context.

🔹 5. Explain cosine similarity and its role in Word2Vec's `` `most_similar()` `` method.

Answer:
Cosine similarity measures the angle between two vectors. In `most_similar()`, it finds words whose vector angles are closest to the given word:

$$\text{cosine\_similarity} = \frac{A \cdot B}{||A|| \cdot ||B||}$$

This helps identify words used in similar contexts, regardless of magnitude.

🔹 6. How would you improve this project to make it more robust for production use?

Answer:

- Add advanced preprocessing: stemming, stop-word removal, lemmatization.
- Train on larger corpora or use pretrained embeddings (e.g., GloVe, fastText).
- Visualize embeddings using t-SNE or PCA.
- Create a user interface for non-technical users.

🔹 7. Can Word2Vec be used for sentence-level tasks? If not, what would you use instead?

Answer:
Word2Vec only gives word-level embeddings. For sentence-level tasks:

- Average word vectors (simple).
- Use **Doc2Vec** (from gensim) for paragraph-level representation.
- Use models like **BERT**, **Sentence-BERT** for context-aware embeddings.

🔹 8. What is out-of-vocabulary (OOV) and how is it handled in traditional vs modern models?

Answer:
OOV refers to words not seen during training:

- In **Word2Vec**, OOV words can't be represented.
- In modern models like **fastText**, subword embeddings help represent OOV words.
- **BERT** uses WordPiece tokenization to break unknown words into known subwords.

🔹 9. What's the difference between TF-IDF and LSA (Latent Semantic Analysis)?

Answer:

- **TF-IDF** is a feature extraction method.

- **LSA** is a dimensionality reduction technique that applies SVD (Singular Value Decomposition) to a TF-IDF matrix to discover latent topics or concepts.

LSA can capture hidden structures and relationships between terms.

🔹 10. How can you evaluate the quality of Word2Vec embeddings?

Answer:

- **Intrinsic evaluation:** Check similarity/analogy accuracy using test sets like Google analogy dataset.
- **Extrinsic evaluation:** Use embeddings in downstream tasks (e.g., classification) and check performance.
- Visualize using t-SNE or PCA to inspect clustering behavior.

Would you like these compiled into a presentation or document for viva prep?