Bigram-Based Sentence Generator - Detailed Documentation

1. Overview:

This script implements a **Bigram-Based Sentence Generator** using **Natural Language Processing (NLP)** techniques. It processes a dataset of text files, extracts **bigrams** (pairs of consecutive words), and generates sentences based on frequently occurring word pairs. The approach uses **Conditional Frequency Distributions** to determine the most probable next word for a given input.

2. Prerequisites and Setup:

2.1 Required Libraries

Ensure that the following Python libraries are installed:

```
pip install nltk
```

Additionally, download the required NLTK dataset:

```
import nltk
nltk.download('punkt')
```

2.2 Directory Structure

Ensure that your project follows this structure:

```
/your_project_directory
  /data        # Folder containing text files for training
    file1.txt
    file2.txt
  slm_book_qa.py  # Main script
```

The data folder must contain .txt files with meaningful text content for training the model.

3. Working Mechanism:

3.1 Text Preprocessing

- Reads all .txt files from the data directory.
- Removes all punctuation **except full stops (.)** to retain sentence boundaries.
- Converts the text to **lowercase** for uniformity.
- Tokenizes the text into words.

3.2 Bigram Extraction & Frequency Calculation

- Generates **bigrams** (pairs of consecutive words) from the tokenized words.
- Computes **frequency distribution** for each bigram.

- Uses a **heap structure** to retain only the **top 3 most frequent bigrams** for each word.

3.3 Sentence Generation

- Accepts a **starting word** as input.
- Predicts the next word using the **most frequent bigrams**.
- Randomly selects one of the top probable words to form a sequence.
- Continues the process until the specified number of words is reached.

4. Code Explanation:

4.1 Importing Required Libraries

```
import random
import nltk
from nltk import bigrams, FreqDist, ConditionalFreqDist
import os
import string
nltk.download('punkt')
```

- Imports random for random selection of words.
- Imports nltk for NLP processing.
- Uses bigrams and FreqDist to compute word pair frequencies.
- Downloads the punkt tokenizer if not already installed.

4.2 Defining the Input Data Directory:

```
input_data_dir = "data"
```

Specifies the folder containing training text files.

4.3 Removing Unnecessary Punctuation:

```
punctuation = string.punctuation.replace('.', '')
```

Removes all punctuation **except full stops** to retain sentence structure.

4.4 Reading and Processing Text Files:

```
def is_hidden(filepath):
    return os.path.basename(filepath).startswith('.')

text_data = ""
for filename in os.listdir(input_data_dir):
```

```
    filepath = os.path.join(input_data_dir, filename)
    if not is_hidden(filepath):
        with open(filepath) as infile:
            for line in infile:
                if line.strip():  # Ignore empty lines
                    for char in punctuation:
                        line = line.replace(char, '')
                    text_data += line
```

- Reads all text files from the data directory.
- Skips hidden files and empty lines.
- Removes unnecessary punctuation.
- Combines the cleaned text into a single string.

4.5 Tokenization and Bigram Frequency Calculation

```
words = nltk.word_tokenize(text_data.lower())
bi_grams = list(bigrams(words))
bi_gram_freq_dist = FreqDist(bi_grams)
```

- Tokenizes words into a list.
- Creates a **bigram list** from the tokenized words.
- Computes the **frequency distribution** of each bigram.

4.6 Filtering Top 3 Bigrams Per Word

```
import heapq

topk = 3
top_bigrams_per_first_word = {}
for (first_word, second_word), freq in bi_gram_freq_dist.items():
    if first_word not in top_bigrams_per_first_word:
        top_bigrams_per_first_word[first_word] = []
    heapq.heappush(top_bigrams_per_first_word[first_word], (freq, second_word))
    if len(top_bigrams_per_first_word[first_word]) > topk:
        heapq.heappop(top_bigrams_per_first_word[first_word])
```

- Uses a **heap queue** to store only the **top 3 most frequent** bigrams for each first word.

- Reduces memory usage and improves performance.

4.7 Converting Heap to a List

```
for first_word in top_bigrams_per_first_word:
    sorted_bigrams = sorted(top_bigrams_per_first_word[first_word], reverse=True)
    top_bigrams_list = []
    for freq, second_word in sorted_bigrams:
        top_bigrams_list.append(second_word)
    top_bigrams_per_first_word[first_word] = top_bigrams_list
```

- Converts the heap data structure into a sorted list for easy lookup.

4.8 Creating a Conditional Frequency Distribution

```
filtered_bi_grams = []
for first_word in top_bigrams_per_first_word:
    for second_word in top_bigrams_per_first_word[first_word]:
        filtered_bi_grams.append((first_word, second_word))


bi_gram_freq = ConditionalFreqDist(filtered_bi_grams)
```

- Converts the filtered bigrams into an NLTK ConditionalFreqDist object.

4.9 Sentence Generation Function

```
def generate_sentence(word, num_words):
    word = word.lower()
    for _ in range(num_words):
        print(word, end=' ')
        next_words = [item for item, freq in bi_gram_freq[word].items()]
        if len(next_words) > 0:
            word = random.choice(next_words)
        else:
            break
    print()


generate_sentence('Asia', 100)
```

- Accepts a **starting word** and a **sentence length**.

- Predicts the next word using the **top bigram pairs**.
- Generates a sentence of the given length.

5. Execution and Observations:

Run the script using:

python slm_book_qa.py

Modify the function call at the end of the script to generate different sentences:

generate_sentence('Asia', 10)

Observations & Possible Issues

Scenario	Observation	Solution
✅ Works Correctly	Generates coherent sentences	No fix needed
⚠️ Word Not Found	No sentence generated	Add more text data
⚠️ Short Sentences	Stops too soon	Increase dataset size
⚠️ Repetitive Output	Words repeat oddly	Implement weighted selection
⏳ Slow Execution	Large dataset takes time	Optimize processing

6. Future Enhancements:

- **Use Trigrams** for improved fluency.
- **Implement Weighted Selection** for more natural output.
- **Expand Training Data** for better sentence diversity.