Search Engine with Auto-Correct

This project requires you to build a functional search engine for a given text corpus. The search engine must include an auto-correct feature using concepts of tokenization, edit distance, and BPE. The search result should contain relevant paragraphs related to the search query. The final deliverable will be a single Python script. You must provide detailed instructions of how to successfully execute your code.

---

1. Data and Resources

- **Corpus**: https://shibamoulilahiri.github.io/gutenberg_dataset.html This source has 3000+ books in text format. Use a random sample of 100 books. If your system works well for 100 books, then scale up to 3000.

---

2. Core Components and Functionality

Your Python script should perform the following tasks sequentially.

Phase 1: Indexing the Corpus

1. **Break into Paragraphs**: Process the corpus and divide each book into multiple paragraphs. These paragraphs will be your unit of search. Assign ID to each paragraph and keep record of the book where it belongs.
2. **Read and Tokenize**: Read the entire corpus and tokenize it into a list of tokens using BPE algorithm.
3. **Build Inverted Index**: Construct an inverted index. The keys of this index should be the unique tokens from the corpus. The value for each key should be a list of paragraph IDs where that token appears. You can also store the position of the token within the paragraph for a more advanced implementation.
4. **Vocabulary**: Maintain a separate list or set of all unique tokens. This will be your vocabulary for the auto-correct feature.

Phase 2: User Interaction and Search

The script should enter a loop that prompts the user for a search query.

1. **Input**: Prompt the user with a message like `Enter search query:`.
2. **Process Query**:
    - Tokenize the user's input.
    - For each token, check if it exists in your **vocabulary**.

Phase 3: Auto-Correct and Out-of-Vocabulary (OOV) Handling

1. **Auto-correct**: If a query token is **not** in the vocabulary, use **edit distance** (e.g., Levenshtein distance) to find a potential correction.
    - Calculate the edit distance between the misspelled token and every token in your vocabulary.
    - Find the vocabulary word with the **minimum edit distance**.
    - If this minimum distance is below a certain threshold (e.g., 2), suggest the corrected word to the user. Print a message like `Did you mean: [corrected_word]?`. The search should then be performed using the corrected word.
2. **BPE for OOV**: If the query token is not in the vocabulary and the edit distance suggestion is not applicable (e.g., all distances are above the threshold), assume the word is a new or compound word. Simply break the OOV word into known sub-word units. For example, if "running" is an OOV word, you might find "run" and "ing" are in your vocabulary.
    - Search for documents containing these sub-word units.

Phase 4: Output

- **Search Results**: For each query, print the top-10 paragraph IDs, content snipper, and book title that matches the query terms.
- **Rank**: You should rank the resultant paragraphs based on some similarity function (e.g., cosine similarity, euclidean distance). The similarity function should measure the similarity between the query and the paragraphs.
- **No Match**: If no paragraph is found, print a message like `No paragraph found for this query.`
- **Loop**: The program should continue to prompt for new queries until the user enters an exit command (e.g., "exit" or "quit").

---

3. Evaluation Criteria

Your project will be graded on the following criteria:

- **Correctness (30%)**:
    - Does the inverted index work correctly?
    - Is your tokenization algorithm accurate?
    - Does the edit distance calculation and auto-correct feature work as described?
    - Is the BPE-like sub-word search functional?
    - Is the similarity measurement function working correctly?
- **Code Quality (30%)**:
    - Is the code well-structured, readable, and commented?
    - Is it contained within a single script?
    - Are your data structures (inverted index) efficient and correctly implemented?
- **Efficiency and Edge Cases (20%)**:

    - Does the search perform reasonably fast on the given corpus?
    - How does the program handle edge cases like empty queries, queries with multiple words, and OOV words with no close matches?
    - Is the logic for handling the auto-correct threshold sensible?
- **Analysis (20%)**:
    - Provide 5 sample queries and their search results as per your system. Explain why you think these results were retrieved? Did auto-correct help? Were there errors?
    - Explain what happens to your search results if you don't do BPE based tokenization and do whitespace based tokenization instead.
    - What happens to your results if you change the edit-distance threshold? Explain with queries and results as per your system.

---

3. Deliverables

You must deliver the following-

1. A python script
2. A detailed instruction of how to correctly execute your code
3. Your analysis writeup.