MapmyIndia – Assignment Round

Sarthak Gupta – 2019AAPS0290G

f20190290@goa.bits-pilani.ac.in || +91 9971565132

Code:

```
class DictionaryTrieNode:
    def __init__(self):
        self.children = {}
        self.is_word = False


class DictionaryTrie:
    def __init__(self):
        self.root = DictionaryTrieNode()

    def insert_word(self, word):
        # Inserts a word into the dictionary trie.
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = DictionaryTrieNode()
            node = node.children[char]
        node.is_word = True

    def search_word(self, word):
        # Searches for exact matches of a word in the dictionary trie.
        node = self._get_prefix_node(word)
        if not node:
            return []

        results = []
        self._dfs_traversal(node, word, '', results)
        return results

    def _get_prefix_node(self, prefix):
        # Traverses the trie to find the node representing the given prefix.
        node = self.root
        for char in prefix:
            if char not in node.children:
                return None
            node = node.children[char]
        return node

    def _dfs_traversal(self, node, remaining_word, current_word, results):
        # Performs a depth-first traversal to find all words matching the
prefix.
        if not remaining_word:
```

```
            if node.is_word:
                results.append(current_word)
            return

        char = remaining_word[0]
        if char in node.children:
            child_node = node.children[char]
            next_word = current_word + char
            self._dfs_traversal(child_node, remaining_word[1:], next_word,
results)

    def find_similar_words(self, word, max_length_diff=2, max_typos=2):
        # Finds similar words to the given word in the dictionary trie.
        all_words = self._get_all_words()
        similar_words = []
        for dict_word in all_words:
            if abs(len(dict_word) - len(word)) <= max_length_diff:
                distance = self._calculate_similarity(word, dict_word)
                if distance is not None and distance <= max_typos:
                    similarity_score = self._calculate_similarity_score(word,
dict_word, distance)
                    similar_words.append((dict_word, similarity_score))
        similar_words.sort(key=lambda x: x[1], reverse=True)
        return similar_words

    def _get_all_words(self):
        # Returns a list of all words in the dictionary trie.
        words = []
        self._dfs_collect_words(self.root, '', words)
        return words

    def _dfs_collect_words(self, node, current_word, words):
        # Performs a depth-first traversal to collect all words in the trie.
        if node.is_word:
            words.append(current_word)

        for char, child_node in node.children.items():
            next_word = current_word + char
            self._dfs_collect_words(child_node, next_word, words)

    @staticmethod
    def _calculate_similarity(word1, word2):
        # Calculates the Levenshtein distance between two words.
        len1 = len(word1)
        len2 = len(word2)
        dp = [[0] * (len2 + 1) for _ in range(len1 + 1)]

        for i in range(len1 + 1):
```

```
            dp[i][0] = i
        for j in range(len2 + 1):
            dp[0][j] = j

        for i in range(1, len1 + 1):
            for j in range(1, len2 + 1):
                cost = 0 if word1[i - 1] == word2[j - 1] else 1
                dp[i][j] = min(
                    dp[i - 1][j] + 1,       # Deletion
                    dp[i][j - 1] + 1,       # Insertion
                    dp[i - 1][j - 1] + cost  # Substitution
                )

                if i > 1 and j > 1 and word1[i - 1] == word2[j - 2] and
word1[i - 2] == word2[j - 1]:
                    dp[i][j] = min(dp[i][j], dp[i - 2][j - 2] + cost)  #
Transposition

        distance = dp[len1][len2]
        if distance is None:
            distance = float('inf')

        return distance

    @staticmethod
    def _calculate_similarity_score(word1, word2, distance):
        # Calculates the similarity score between two words based on their
distance.
        max_len = max(len(word1), len(word2))
        return (max_len - distance) / max_len


def build_dictionary():
    # Builds a dictionary trie from a data file.
    dictionary_trie = DictionaryTrie()
    with open("datafile.txt", "r") as file:
        for line in file:
            word = line.strip().lower()  # Remove leading/trailing whitespaces
and convert to lowercase
            dictionary_trie.insert_word(word)
    return dictionary_trie


def main():
    # Main function to search for words and find similar words in the
dictionary.
    dictionary = build_dictionary()
    search_term = input("Enter a word to search: ").lower()
```

```
    results = dictionary.search_word(search_term)
    similar_words = dictionary.find_similar_words(search_term)

    if results:
        print("Exact matches:")
        for word in results:
            print(word)

    if not results and similar_words:
        print("Showing similar words:")
        top_similar_words = sorted(similar_words, key=lambda x: x[1],
reverse=True)[:10]
        for word, similarity_score in top_similar_words:
            if similarity_score > 0.5:
                print(f"{word}")
                # print(f"{word} (Similarity Score: {similarity_score})") #
Uncomment to show similarity score

    if not results and not similar_words:
        print("No matches found.")


if __name__ == "__main__":
    main()
```

Explanation:

This code implements a spell checker using a dictionary trie data structure and the Levenshtein distance algorithm to measure word similarity. The spell checker suggests alternative words for a given input, accounting for variations, typos, and misspellings.

The dictionary trie is a tree-like structure that efficiently stores a large set of words. It allows for quick word retrieval and search operations, making it suitable for spell checking. The trie is built by inserting each word from a data file, ensuring efficient storage and retrieval.

To measure word similarity, we use the Levenshtein distance algorithm. It calculates the minimum number of edit operations (insertions, deletions, and substitutions) required to transform one word into another. This algorithm considers both the length difference and individual character differences between words.

The Levenshtein distance approach offers several benefits. Firstly, it's flexible, capturing various types of errors and typos that can occur in words. Secondly, it's effective, providing a reliable measure of similarity between strings. Lastly, it's widely used in natural language processing tasks, including spell checking, text correction, and fuzzy matching.

Here's how the spell checker works: It first searches for exact matches of the input word in the dictionary trie. If any exact matches are found, they're considered the correct spellings. If no exact matches exist, the spell checker employs the Levenshtein distance algorithm to find similar words in the dictionary. It considers words with a maximum length difference and a

maximum number of allowable typos. The algorithm calculates the Levenshtein distance between the input word and each word in the dictionary, and assigns a similarity score based on the distance. Words with a similarity score above a specified threshold are considered similar.

The similar words are then sorted based on their similarity scores in descending order, and the top matching words are provided as suggestions. If no exact matches or similar words are found, the spell checker notifies the user that no matches were found.

By combining the dictionary trie and the Levenshtein distance algorithm, this spell checker provides accurate suggestions for alternative words, even in the presence of variations, typos, or misspellings. The efficient data structure and reliable similarity metric ensure dependable and effective spell checking functionality.