884. Uncommon Words from Two Sentences

Easy Hash Table String Leetcode Link

Problem Description

The given problem involves finding words that are unique to each of two separate sentences. In more detail, a 'sentence' is defined as a string of words, with each word being separated by a single space and consisting only of lowercase letters. A word is deemed 'uncommon' if it satisfies two requirements: firstly, it must appear exactly once within either sentence; secondly, it must not appear in the other sentence at all.

To solve this problem, we are tasked with comparing two distinct sentences, identified as `s1` and `s2`. The goal is to curate a list containing all such 'uncommon' words. The solution does not require the words to be in any particular sequence, implying that the words can be listed in any order. The core challenge lies in devising an efficient method to distinguish which words appear only once in either sentence and do not show up in the other.

Intuition

To arrive at the solution for identifying uncommon words from two sentences, consider a straightforward approach: counting the occurrences of each word across both sentences. By combining the counts, we can determine if a word is uncommon by checking if its total count is exactly one, signifying it appears only once and isn't shared between the two sentences.

The Python `Counter` class from the `collections` module simplifies this task. It allows us to count the frequency of elements within an iterable, such as a list of words. Therefore, the first step is to split each sentence into a list of words using the `split()` method, which naturally separates the sentence according to spaces. Applying `Counter` to these lists provides a dictionary-like object where keys are the words and values are their respective counts.

The next step is to combine these counts. The `+` operator merges the two `Counter` objects in a way that adds up the counts for common words between `s1` and `s2`. This merged counter now holds the total frequency of every word in both sentences.

The final step is straightforward: iterate over the items in the combined counter and select the words (`s`) where the associated count (`v`) is exactly one. These words are the 'uncommon' words which need to be returned. Using list comprehension makes this step concise and efficient, resulting in a one-liner solution that fetches the required list of uncommon words.

In essence, the solution leverages the power of Python's standard library to perform the frequency analysis and then filters the results to match the specific criterion laid out in the problem statement.

Solution Approach

The solution uses the `Counter` data structure from Python's `collections` module to implement the approach efficiently. Here's how the implementation breaks down:

1. **Split the sentences into words**: The first step is to split `s1` and `s2` into individual words based on spaces. This is done using Python's built-in `split()` method:

   ```
   words_s1 = s1.split()
   words_s2 = s2.split()
   ```

   After this step, `words_s1` and `words_s2` are lists that contain all the words from `s1` and `s2`, respectively.

2. **Count the word occurrences**: Next, we create two `Counter` objects for these lists:

   ```
   counter_s1 = Counter(words_s1)
   counter_s2 = Counter(words_s2)
   ```

   Here, `counter_s1` and `counter_s2` act like dictionaries where each word is a key, and its count in the corresponding sentence is the value.

3. **Combine the counters**: By adding these two `Counter` objects using the `+` operation, we obtain a single counter that contains the sum of word counts from both sentences:

   ```
   combined_counter = counter_s1 + counter_s2
   ```

   In `combined_counter`, any word with a total count greater than 1 indicates that it is either repeated within the same sentence or present in both sentences.

4. **Filter out the uncommon words**: Finally, we need to gather only those words that appear exactly once – which implies they're uncommon:

   ```
   uncommon_words = [word for word, count in combined_counter.items() if count == 1]
   ```

   This list comprehension iterates over the items of `combined_counter`. For each word, it checks if the count is 1 (using `if count == 1`), and if so, the word is added to the list `uncommon_words`.

The full implementation of the function `uncommonFromSentences` as a method inside the `Solution` class is as follows:

```
class Solution:
    def uncommonFromSentences(self, s1: str, s2: str) -> List[str]:
        # Step 1 and 2: count word occurrences
        cnt = Counter(s1.split()) + Counter(s2.split())
        # Step 3 and 4: combine counts and filter uncommon words
        return [word for word, count in cnt.items() if count == 1]
```

In conclusion, by utilizing the `Counter` data structure to perform frequency analysis and array comprehensions for filtering, the solution efficiently identifies all uncommon words with minimal code and avoids the need for handcrafting frequency calculations or manual comparisons between word lists.

Example Walkthrough

Let's consider two sentences as examples:

`s1`: `"apple banana"` `s2`: `"banana orange apple"`

Following the solution approach:

1. **Split the sentences into words**: For `s1`: `words_s1 = ['apple', 'banana']` For `s2`: `words_s2 = ['banana', 'orange', 'apple']`

   Both sentences are split into lists of individual words.

2. **Count the word occurrences**: Here's what the `Counter` objects might look like: `counter_s1 = Counter({'apple': 1, 'banana': 1})` `counter_s2 = Counter({'banana': 1, 'orange': 1, 'apple': 1})`

   Each word is now associated with its occurrence count in the sentences.

3. **Combine the counters**: When combined, the counters reflect the total occurrence of each word: `combined_counter = Counter({'apple': 2, 'banana': 2, 'orange': 1})`

   This shows 'apple' and 'banana' each have a total count of 2, while 'orange' has a count of 1.

4. **Filter out the uncommon words**: We want the words which have a count of exactly 1: `uncommon_words = ['orange']`

   Only 'orange' fulfills the criteria of being uncommon (appearing exactly once overall and not in both sentences).

Thus, with our example, the function `uncommonFromSentences(s1, s2)` would return `['orange']` as the list of uncommon words.

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def uncommonFromSentences(self, sentence1: str, sentence2: str) -> List[str]:
        # Combine the word counts from both sentences
        combined_counts = Counter(sentence1.split()) + Counter(sentence2.split())

        # Find and return the list of words that appear only once
        return [word for word, count in combined_counts.items() if count == 1]

```

Java Solution

```java
class Solution {
    public String[] uncommonFromSentences(String s1, String s2) {
        // Create a Hash Map to store word counts
        Map<String, Integer> wordCounts = new HashMap<>();

        // Split the first string by spaces and count the occurrences of each word
        for (String word : s1.split(" ")) {
            wordCounts.put(word, wordCounts.getOrDefault(word, 0) + 1);
        }

        // Split the second string by spaces and count the occurrences of each word
        for (String word : s2.split(" ")) {
            wordCounts.put(word, wordCounts.getOrDefault(word, 0) + 1);
        }

        // List to hold the words that occur exactly once
        List<String> uniqueWords = new ArrayList<>();

        // Iterate through the entry set of wordCounts
        for (Map.Entry<String, Integer> entry : wordCounts.entrySet()) {
            // If a word count is exactly 1, it's uncommon, add to the list
            if (entry.getValue() == 1) {
                uniqueWords.add(entry.getKey());
            }
        }
        // Return the unique words as an array of strings
        return uniqueWords.toArray(new String[0]);
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
#include <sstream>
using namespace std;

class Solution {
public:
    vector<string> uncommonFromSentences(string A, string B) {
        // Map to store the count of each word across both sentences
        unordered_map<string, int> wordCount;

        // Lambda function to parse each word in a sentence and update the word count
        auto addWordsToCount = [&](const string& sentence) {
            stringstream stream(sentence);
            string word;
            while (stream >> word) {
                ++wordCount[word]; // Increment the word count for each word found
            }
        };

        // Parse both sentences A and B to count the words
        addWordsToCount(A);
        addWordsToCount(B);

        // Vector to store the result - the uncommon words from both sentences
        vector<string> result;

        // Iterate through the word count map
        for (const auto& entry : wordCount) {
            // If the word count is 1, that means it's an uncommon word
            if (entry.second == 1) {
                // Add it to the result list
                result.emplace_back(entry.first);
            }
        }

        return result; // Return the list of uncommon words
    }
};

```

Typescript Solution

```typescript
// This function takes two sentences as input and returns an array of words that appear
// exactly once in either of the two sentences
function uncommonFromSentences(sentence1: string, sentence2: string): string[] {
    // Create a map to keep track of word counts across both sentences
    const wordCounts: Map<string, number> = new Map();

    // Split both sentences into words and combine them into a single array
    // Then iterate over the array to count the occurrences of each word
    for (const word of [...sentence1.split(' '), ...sentence2.split(' ')]) {
        wordCounts.set(word, (wordCounts.get(word) || 0) + 1);
    }

    // Array to store the uncommon words (words that appear exactly once)
    const uncommonWords: string[] = [];

    // Iterate over the wordCounts map to find words with a count of 1
    // These are the words that are unique to either sentence
    for (const [word, count] of wordCounts.entries()) {
        if (count === 1) {
            uncommonWords.push(word);
        }
    }

    // Return the array of uncommon words
    return uncommonWords;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code mainly involves three steps:

1. The splitting of strings `s1` and `s2`: This operation has a time complexity of `O(N + M)`, where `N` is the length of `s1` and `M` is the length of `s2`. The `.split()` method goes through each character in the strings.

2. The creation of two `Counter` objects from the split results of `s1` and `s2`: The `Counter` object creation from a list of words has a time complexity of `O(K1 + K2)`, where `K1` is the number of words in `s1` and `K2` is the number of words in `s2`. It counts the frequency of each word.

3. Adding two `Counter` objects and filtering for uncommon words: The addition of two `Counter` objects is `O(U)`, where `U` is the number of unique words across both `s1` and `s2`. The list comprehension that follows iterates through the combined `Counter` object, which also has a complexity of `O(U)`.

Thus, the overall time complexity of the code is `O(N + M + U)`, where `U <= K1 + K2` since `U` is the count of unique words in both `s1` and `s2`.

Space Complexity

The space complexity is determined by:

1. The lists created from splitting `s1` and `s2`: This depends on the number of words in `s1` and `s2`, which is `O(K1 + K2)`.

2. The `Counter` objects for `s1` and `s2`: This also depends on the number of unique words, which would be `O(U)`.

3. The final list of uncommon words: In the worst-case scenario, all words are uncommon, which would also result in `O(U)` space complexity.

Since `U <= K1 + K2`, the overall space complexity can be described as `O(K1 + K2)`.