1772. Sort Features by Popularity

Medium Array Hash Table String Sorting Leetcode Link

Problem Description

In the given problem, we have two lists: `features` and `responses`.

`features` is a list of single-word strings where each word represents a unique feature of a product.

`responses` is a list of strings where each string may contain several words expressing features liked by a user. These strings represent user feedback from a survey.

The goal is to sort the features array in order of decreasing popularity. Popularity of a feature is defined by the number of responses that mention the feature at least once. One thing to note is that even if a response mentions a feature multiple times, it still only counts once towards the feature's popularity.

Furthermore, if two features have the same popularity, our sorting should prioritize them based on the order in which they appear in the `features` list.

Finally, we need to return the features sorted by the rules described above.

Intuition

For the solution, we need to do the following:

1. Count how many times each feature is mentioned in the `responses` list. To ensure that each feature is counted only once per response, we can create a set of words from each response. This will automatically remove duplicates.

2. After we have the counts of each feature as per the responses, we need to sort the `features` list. The sorting is not regular alphabetical or numerical sorting; it is based on the popularity of the features.

Therefore, we need a custom sort that:

- Sorts features in descending order of their popularity.
- When features have the same popularity count, they should retain their original relative order as given in the input `features` array.

3. To implement this, we use a sorting function with a custom key. In Python, this can be achieved via the `sorted` function with a `lambda` function as the key. The `lambda` function returns the negative count of how many times a feature occurs, thus enabling sorting in descending order based on popularity.

By following this approach, we can efficiently sort the features according to the problem's requirements.

Solution Approach

To implement the solution, the code utilizes a couple of Python-specific techniques and data structures. Here's a step-by-step walkthrough of the solution:

1. Import the `Counter` class from the `collections` module. A `Counter` is a dictionary subclass designed for counting hashable objects. It's an unordered collection where elements are stored as dictionary keys and their counts as dictionary values.

2. Initialize the `Counter` by the variable name `cnt`. It's going to be used to store the number of times each feature appears in the responses.

3. Iterate through each response in the `responses` list using a `for` loop.

4. Inside the loop, split the current response `r` using `r.split()`, converting it into a list of words. Then convert this list into a set `ws` to ensure that each word is unique and duplicate mentions of a feature in a single response are eliminated.

5. Iterate through each word `s` in the set `ws`. If it's a word that represents a feature, it should be counted. Increment its count in our `Counter` by doing `cnt[s] += 1`.

6. Now, we need to sort the `features` list by the count of each feature while preserving the order of features with the same popularity. To do this, we use the `sorted` function with a custom `lambda` function as the sorting key: `lambda x: -cnt[x]`. This lambda function returns the count from the `Counter` for each element `x` in `features`, negated so that the sorting is in descending order.

7. The `sorted` function with the custom lambda function will sort the features by descending popularity, and in case of a tie, it will preserve the original order, as it's stable sort (it maintains the relative order of records with equal values).

Here is the final sorted list:

```
sorted_features = sorted(features, key=lambda x: -cnt[x])
```

This Python code gives us the desired output where the `features` are sorted according to the specified rules.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we have the following `features` list and `responses` list:

```
features = ["storage", "color", "battery", "screen"]
responses = [
    "I wish the battery life was longer",
    "Loving the new screen and high storage capacity",
    "The color is so vibrant and the screen is the perfect size",
    "Battery life is great, screen is great, storage just what I needed",
    "I wish the color options were more vibrant"
]
```

Now, let's apply our solution approach step by step:

1. **Count Feature Mentions**: First, we need to count how many times each feature is mentioned across all `responses`. According to the given approach, for each response, we convert it into a set of words so that duplicates within the same response are eliminated.

2. **Initialize Counter**: We initialize a `Counter` object called `cnt` to keep the counts.

```
from collections import Counter
cnt = Counter()
```

3. **Process Responses**: Let's loop through each `response` and update the `Counter` with words only if they are in our `features` list.

```
for r in responses:
   ws = set(r.split())
   for s in ws:
      if s in features:
         cnt[s] += 1
```

After processing all responses, `cnt` might look like this:

```
cnt = Counter({'battery': 2, 'screen': 3, 'storage': 2, 'color': 2})
```

Here, 'screen' was mentioned in 3 different responses, while 'battery', 'storage', and 'color' were mentioned in 2.

4. **Sort Features**: Next, we sort the `features` by descending popularity using the `sorted` function with a `lambda` key that uses the `Counter` counts.

```
sorted_features = sorted(features, key=lambda x: -cnt[x])
```

The `sorted_features` list will look like this:

```
['screen', 'battery', 'storage', 'color']
```

The feature 'screen' comes first because it has the highest count. 'battery', 'storage', and 'color' all have the same count. However, 'battery' appears before 'storage' and 'color' in the original list, so it comes next. Likewise, 'storage' comes before 'color', which is why 'color' is last.

5. **Result**: Finally, we have our features sorted by popularity while preserving the order in which they appear in the `features` list when they have the same popularity:

```
sorted_features = ["screen", "battery", "storage", "color"]
```

This sorted list aligns with our problem's requirement of sorting the features by their popularity, ensuring that the ones with the same frequency of mentions maintain the order from the original `features` list.

Python Solution

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

class Solution:
    def sortFeatures(self, features: List[str], responses: List[str]) -> List[str]:
        # Create a counter to keep track of the number of times each word appears in the responses
        counter = Counter()

        # Iterate through each response in the list of responses
        for response in responses:
            # Split the response into words and convert it to a set to remove duplicates
            unique_words = set(response.split())

            # Update the count for each unique word in the current response
            for word in unique_words:
                counter[word] += 1

        # Sort the list of features based on their counts in a descending order
        # Features that do not appear in the counter will have a default count of 0
        sorted_features = sorted(features, key=lambda feature: -counter[feature])

        # Return the list of features sorted by their popularity
        return sorted_features

# Example usage:
# sol = Solution()
# features = ["cooler", "lock", "touch"]
# responses = ["I love the cooler cooler", "lock touch cool", "locker like touch"]
# print(sol.sortFeatures(features, responses)) # Output: ["touch", "cooler", "lock"]
```

Java Solution

```java
class Solution {
    public String[] sortFeatures(String[] features, String[] responses) {
        // Create a map to count the occurrence of each word in the responses
        Map<String, Integer> featureCounts = new HashMap<>();
        for (String response : responses) {
            // Create a set to store unique words in the current response
            Set<String> uniqueWords = new HashSet<>();
            // Split the response into words
            for (String word : response.split(" ")) {
                uniqueWords.add(word);
            }
            // Increment the count for each unique word
            for (String word : uniqueWords) {
                featureCounts.put(word, featureCounts.getOrDefault(word, 0) + 1);
            }
        }

        int numFeatures = features.length;
        // Initialize an array of indexes corresponding to the features array
        Integer[] indexes = new Integer[numFeatures];
        for (int i = 0; i < numFeatures; ++i) {
            indexes[i] = i;
        }

        // Sort the indexes based on the feature occurrence counts and their original order
        Arrays.sort(indexes, (index1, index2) -> {
            int countDifference = featureCounts.getOrDefault(features[index2], 0) -
                                  featureCounts.getOrDefault(features[index1], 0);
            // If counts are equal, sort by the original order
            return countDifference == 0 ? index1 - index2 : countDifference;
        });

        // Create an array to hold the sorted features
        String[] sortedFeatures = new String[numFeatures];
        for (int i = 0; i < numFeatures; ++i) {
            sortedFeatures[i] = features[indexes[i]];
        }

        // Return the sorted array of features
        return sortedFeatures;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <sstream>
#include <algorithm>
#include <numeric>

class Solution {
public:
    std::vector<std::string> sortFeatures(std::vector<std::string>& features, std::vector<std::string>& responses) {
        // Map to store the count of each feature mentioned in the responses
        std::unordered_map<std::string, int> featureCounts;

        // Iterate through all the responses
        for (auto& response : responses) {
            std::stringstream ss(response);
            std::string word;
            std::unordered_set<std::string> uniqueWords;

            // Extract each word from the response and add to a set to ensure uniqueness
            while (ss >> word) {
                uniqueWords.insert(word);
            }

            // Increase the count for each unique word found in the responses that match a feature
            for (auto& word : uniqueWords) {
                featureCounts[word]++;
            }
        }

        int numFeatures = features.size();
        std::vector<int> indices(numFeatures);
        // Initialize the indices sequence
        std::iota(indices.begin(), indices.end(), 0);

        // Sort the indices based on the counts of the features
        std::sort(indices.begin(), indices.end(), [&](int index1, int index2) -> bool {
            int countDifference = featureCounts[features[index1]] - featureCounts[features[index2]];
            // If counts differ, sort in descending order; if counts are same, sort by index
            return countDifference > 0 || (countDifference == 0 && index1 < index2);
        });

        // Prepare the sorted list of features based on the sorted indices
        std::vector<std::string> sortedFeatures(numFeatures);
        for (int i = 0; i < numFeatures; ++i) {
            sortedFeatures[i] = features[indices[i]];
        }

        // Return the sorted list of features
        return sortedFeatures;
    }
};
```

Typescript Solution

```typescript
// Required imports or type definitions (if any) would be placed here

let featureCounts: { [feature: string]: number } = {};

// Function to sort features based on their frequency in the given responses
function sortFeatures(features: string[], responses: string[]): string[] {
    featureCounts = {}; // Resetting the feature counts for a fresh start

    // Iterate through all the responses to count feature occurrences
    responses.forEach(response => {
        let uniqueWords: Set<string> = new Set<string>(); // To ensure uniqueness of words in a response
        response.split(/\s+/).forEach(word => {
            uniqueWords.add(word); // Add each word to the set
        });

        // Update the count for each unique word that matches a feature
        uniqueWords.forEach(word => {
            if (features.includes(word)) {
                if (!featureCounts[word]) featureCounts[word] = 0;
                featureCounts[word]++;
            }
        });
    });

    // Initialize indices for sorting
    let indices: number[] = features.map((_, index) => index);

    // Sort indices based on feature occurrence count and their original order
    indices.sort((index1, index2) => {
        let count1 = featureCounts[features[index1]] || 0; // Fallback to 0 if not found
        let count2 = featureCounts[features[index2]] || 0;
        return count2 - count1 || index1 - index2; // Descending by count, then by original index
    });

    // Prepare the sorted list of features
    let sortedFeatures: string[] = indices.map(index => features[index]);

    // Return the list of sorted features
    return sortedFeatures;
}

// Example use:
// const sorted = sortFeatures(["feature1", "feature2"], ["I love feature2", "feature1 is great"]);
// console.log(sorted); // Logs feature names sorted by their frequency and original order
```

Time and Space Complexity

Time Complexity

The time complexity of the given code snippet is `O(N + U + FlogF)`, where:

- `N` is the total number of words across all responses.
- `U` is the number of unique words across all responses.
- `F` is the number of features.

Here's how the time complexity breaks down:

1. Splitting each response and adding words to the set has a complexity of `O(N)`, where `N` is the total number of words. This includes the cost of splitting the response into words and processing each word.
2. Counting occurrences of each unique word involves a constant-time operation per unique word, resulting in a complexity of `O(U)`.
3. Sorting the features list is `O(FlogF)` because we are sorting `F` features using a comparison sort where features are compared based on their counts.

Space Complexity

The space complexity of the given code snippet is `O(U + F)`, where:

- `U` is the number of unique words across all responses. This is because the counter holds counts for each unique word.
- `F` is the number of features, which accounts for the space to store the `features` list.

We have to store the counts for all unique words and the final sorted `features` list, contributing to the space complexity.