1090. Largest Values From Labels

Medium Greedy Array Hash Table Counting Sorting

Leetcode Link

Problem Description

In this problem, there is a set of n items, each with an associated value and label. Two arrays `values` and `labels` represent these, where `values[i]` is the value and `labels[i]` is the label for the i-th item. We are given two additional integers, `numWanted` which represents the maximum number of items to choose, and `useLimit` which represents the maximum number of items with the same label we are allowed to choose.

The goal is to find a subset s of these items that maximizes the total value (or "score") while adhering to the constraints:

1. The size of s is less than or equal to `numWanted`.
2. No more than `useLimit` items in s can have the same label.

The desired output is the maximum score that can be achieved following these rules.

Intuition

The intuitive approach to solving this problem is to maximize the value while respecting the constraints given. To start, since we want the maximum value, we should consider the items with the highest values first. Hence, we sort the items by value in descending order. Once sorted, a simple strategy can be employed:

- Iterate over the list of items in order of descending value.
- Keep a count of how many items with each label have been added to our subset s.
- Add the value of the current item to our score if the count of items with the same label is less than the `useLimit`.
- Increment the count of items chosen.
- If the number of items chosen reaches `numWanted`, we've found the best possible subset, and we can stop and return the score.

This approach ensures that we are always adding the item with the highest available value while respecting the `useLimit` condition for labels and not exceeding the desired subset size `numWanted`.

Solution Approach

The solution approach relies on a greedy algorithm that selects items with higher values first, while ensuring that it does not select more than `useLimit` items with the same label. To accomplish this effectively, the following steps are taken:

- **Sorting**: We need to sort the items by their values in descending order. To do so, we use Python's built-in `sorted` function, pairing each value with its corresponding label using `zip(values, labels)`. The `reverse=True` parameter is used to sort the pairs in descending order of value.
- **Counting**: To keep track of the number of items with the same label selected, we use a `Counter` from Python's `collections` module. This data structure allows easy increments and tracking of counts corresponding to each label.
- **Selection Loop**: Iterate through the sorted pairs of `(value, label)`. For each pair, check if the current label's count is less than the `useLimit` using the previously initialized `Counter`.
- If the label's count is within the limit, we proceed to include the item's value in our current subset score `ans` and increment the count for that label in the `Counter`.
- We also keep a tally of the total number of items selected in `num`. When `num` reaches `numWanted`, we have selected the maximum allowed number of items, and we break out of the loop since we cannot add any more items to the subset.
- Finally, after the loop (either once we've reached `numWanted` or gone through all items), we return `ans` as the maximum score obtained from the selected subset of items.

The combination of sorting, counting, and a selection loop makes this greedy algorithm efficient and ensures that at each step, the best choice is made towards achieving the optimal solution (maximizing the subset score under the given constraints).

Mathematically, if we denote x as the value and l as the label of the current item being considered, and cnt as the `Counter` for labels, the pseudo-code can be described as follows:

```
sort items by value in descending order
initialize ans to 0, num to 0, and cnt to Counter()
for each (x, l) in sorted items:
    if cnt[l] < useLimit:
        cnt[l] += 1
        num += 1
        ans += x
        if num == numWanted:
            break
return ans
```

This implementation ensures that our greedy algorithm selects items in a way that maximizes the score while conforming to the constraints of `numWanted` and `useLimit`.

Example Walkthrough

Consider the following small example to illustrate the solution approach:

Suppose we have the following input:

- `values` = [5, 4, 3, 2, 1]
- `labels` = [1, 1, 2, 2, 3]
- `numWanted` = 3
- `useLimit` = 1

Following the solution approach, we would execute these steps:

1. **Sort the items by value in descending order:**

   We pair each value with its corresponding label and sort:

   Before sorting:

   - values: [5, 4, 3, 2, 1]
   - labels: [1, 1, 2, 2, 3]

   After sorting:

   - sorted_pairs: [(5, 1), (4, 1), (3, 2), (2, 2), (1, 3)]
2. **Initialize the counter and other variables:**

   We start with an empty `Counter` to track the labels, `ans` as 0 (our cumulative value), and `num` as 0 (the number of items in our subset):

   - `ans` (max score): 0
   - `num` (number of items selected): 0
   - `cnt` (the `Counter` for labels): {}
3. **Selection Loop:**

   We iterate over the sorted pairs and apply the constraints:

   a) (5, 1): Count for label 1 is 0, which is less than `useLimit`. We select this item.

   - `ans` becomes 5 (0 + 5)
   - `num` becomes 1 (0 + 1)
   - `cnt` becomes {1: 1}

   b) (4, 1): Count for label 1 is currently 1, equal to `useLimit`. We cannot select this item.

   c) (3, 2): Count for label 2 is 0, which is less than `useLimit`. We select this item.

   - `ans` becomes 8 (5 + 3)
   - `num` becomes 2 (1 + 1)
   - `cnt` becomes {1: 1, 2: 1}

   d) (2, 2): Count for label 2 is currently 1, equal to `useLimit`. We cannot select this item.

   e) (1, 3): Count for label 3 is 0, which is less than `useLimit`. We select this item.

   - `ans` becomes 9 (8 + 1)
   - `num` becomes 3 (2 + 1) --> This reaches `numWanted`, we stop selecting items.
   - `cnt` becomes {1: 1, 2: 1, 3: 1}
4. **Return the maximum score:**

   Since we have selected the maximum number of items (`numWanted`), we return `ans` which is 9. This is the maximum score we can achieve by selecting a subset of 3 items following the given constraints.

   The selected subset is represented by the pairs (5, 1), (3, 2), and (1, 3), leading to the maximum value of 9.

Applying this approach has allowed us to maximize our score under the constraints of `numWanted` and `useLimit`.

Python Solution

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

class Solution:
    def largestValsFromLabels(self, values: List[int], labels: List[int], num_wanted: int, use_limit: int) -> int:
        # Initialize the answer variable to store the sum of the largest values
        # Initialize a counter variable to keep track of how many values are included
        total_value = 0
        chosen_count = 0

        # Create a counter to keep track of how many times each label has been used
        label_count = Counter()

        # Sort the value-label pairs in decreasing order of values to pick the largest values first
        for value, label in sorted(zip(values, labels), reverse=True):
            # Check if the current label has been used less than the use_limit
            if label_count[label] < use_limit:
                # If so, increment the count for the label
                label_count[label] += 1
                # Increment the count of chosen values
                chosen_count += 1
                # Add the current value to the total_value
                total_value += value
                # If we have reached the num_wanted, no need to consider further values
                if chosen_count == num_wanted:
                    break

        # Return the final sum of the chosen values
        return total_value
```

Java Solution

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

class Solution {
    public int largestValsFromLabels(int[] values, int[] labels, int numWanted, int useLimit) {
        int itemCount = values.length; // Total number of items
        int[][] valueLabelPairs = new int[itemCount][2]; // To store value-label pairs

        // Populate the value-label pairs for sorting
        for (int i = 0; i < itemCount; ++i) {
            valueLabelPairs[i] = new int[] {values[i], labels[i]};
        }

        // Sort the pairs in descending order based on the values
        Arrays.sort(valueLabelPairs, (pair1, pair2) -> pair2[0] - pair1[0]);

        Map<Integer, Integer> labelUsageCount = new HashMap<>(); // Map to keep track of label usage

        int totalValue = 0; // Sum of values selected
        int itemsSelected = 0; // Number of items selected

        // Iterate over sorted value-label pairs
        for (int i = 0; i < itemCount && itemsSelected < numWanted; ++i) {
            int currentValue = valueLabelPairs[i][0]; // Current item's value
            int currentLabel = valueLabelPairs[i][1]; // Current item's label

            // Check if we can use more items with this label
            if (labelUsageCount.getOrDefault(currentLabel, 0) < useLimit) {
                // If yes, select the current item
                labelUsageCount.merge(currentLabel, 1, Integer::sum); // Increment label usage
                itemsSelected++; // Increment number of selected items
                totalValue += currentValue; // Add value to total
            }
        }

        return totalValue; // Return the sum of the selected values
    }
}
```

C++ Solution

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

class Solution {
public:
    int largestValsFromLabels(vector<int>& values, vector<int>& labels, int numWanted, int useLimit) {
        int itemCount = values.size(); // Get the count from the given values.
        vector<pair<int, int>> valueLabelPairs(itemCount);

        // Pairing values with labels and negating values for reverse sort.
        for (int i = 0; i < itemCount; ++i) {
            valueLabelPairs[i] = {-values[i], labels[i]};
        }

        // Sort pairs by value in descending order.
        sort(valueLabelPairs.begin(), valueLabelPairs.end());

        unordered_map<int, int> labelCounts; // To track the number of times a label has been used.
        int totalValue = 0; // Total value of selected items.
        int selectedItems = 0; // Number of items selected.

        // Iterate over the sorted pairs.
        for (int i = 0; i < itemCount && selectedItems < numWanted; ++i) {
            int value = -valueLabelPairs[i].first; // Reverse negation to get the original value.
            int label = valueLabelPairs[i].second;

            // Check if we can use more items with this label.
            if (labelCounts[label] < useLimit) {
                labelCounts[label]++; // Increment the use count of this label.
                selectedItems++; // One more item is selected.
                totalValue += value; // Increase the total value by this item's value.
            }
        }

        return totalValue; // Return the total value of the selected items.
    }
};
```

Typescript Solution

```typescript
function largestValsFromLabels(
    values: number[],
    labels: number[],
    numWanted: number,
    useLimit: number,
): number {
    // The number of items in the values array
    const itemCount = values.length;

    // Combine each value with its corresponding label into a pair
    const valueLabelPairs = new Array(itemCount);
    for (let i = 0; i < itemCount; ++i) {
        valueLabelPairs[i] = [values[i], labels[i]];
    }

    // Sort the pairs descending by value
    valueLabelPairs.sort((a, b) => b[0] - a[0]);

    // Initialize a map to keep count of used labels
    const labelCount: Map<number, number> = new Map();

    // Initialize the accumulator for the sum of the largest values
    let totalValue = 0;

    // Loop through the sorted pairs, and pick the largest unused values
    // respecting the useLimit for labels we also respect the numWanted limit.
    for (let i = 0, chosenItems = 0; i < itemCount && chosenItems < numWanted; ++i) {
        const [value, label] = valueLabelPairs[i];
        // Retrieve the current count for this label, defaulting to 0 if not present
        const currentCount = labelCount.get(label) || 0;

        // If we haven't reached the useLimit for this label, choose this value
        if (currentCount < useLimit) {
            labelCount.set(label, currentCount + 1); // Increment the count for this label
            chosenItems++; // Increment the count of chosen items
            totalValue += value; // Add the value to the total
        }
    }

    // Return the sum of the largest values chosen
    return totalValue;
}
```

Time and Space Complexity

The given Python code snippet is a function that selects the largest values from a list `values`, each associated with a label from the list `labels`, with two constraints: a maximum number of items selected (`numWanted`) and a maximum number of times each label can be used (`useLimit`). The function returns the sum of the largest values selected according to these constraints.

Time Complexity

The time complexity of the code can be broken down as follows:

1. Sorting the combined list of values and labels: `sorted(zip(values, labels), reverse=True)` takes $O(N \log N)$ time, where N is the length of the values (and labels) list.
2. The `for` loop iterates over the sorted list, which has N elements. In the worst case, it will iterate over all N elements once.

Combining these operations, the overall time complexity of the code is dominated by the sorting step, resulting in $O(N \log N)$.

Space Complexity

The space complexity can be analyzed as follows:

1. The additional space used by the sorted list of zipped values and labels is $O(N)$, where N is the length of the original lists.
2. The `Counter` object `cnt` which keeps track of the number of times each label has been used will, in the worst case, have as many elements as there are distinct labels. In the worst case, this can also be $O(N)$ if every value has a unique label.

Therefore, the overall space complexity is $O(N)$ due to the space required to store the sorted list and the counter.