Data Structures and Algorithms I

Homework Assignment 3

Wachmann Elias

31. Oktober 2021

Inhaltsverzeichnis

1 Algotithmus in Worten & Pseudo-Code

Aufgabenstellung ist es, aus einer Liste mit beliebig vielen Produkt-Reviews eine abwärtssortierte Liste mit der Häufigkeit von k-hintereinander-stehenden Worten `c` und eine zugehörige Liste dieser Worte `y` zu generieren.

Der Algorithmus in Worten

Dem Algorithmus `count_vectorizer` wird eine list of lists namens `texts` und die Anzahl der aufeinanderfolgenden Wörter `k` übergeben. Zuerst werden für jede Review in `texts` immer k-hintereinander-stehende Wörter in einem String konkateniert und in ein Python `Dict` gespeichert, hierbei wird zuerst versucht den String als Key zu verwenden, um so den count um 1 zu erhöhen. Schlägt dies fehl, so wird ein neuer Key mit count = 1 angelegt. Nun werden die Keys und Values des Dictionaries in die beiden Listen `y` bzw. `c` entpackt. Nun wird `c` mittels `merge_sort` absteigend sortiert, dabei wird die Liste der Worte `y` gleich sortiert, sodass weiterhin jede Stelle in `c` die Anzahl der dazugehörigen Phrase in `y` gibt.

Pseudo-Code

Algorithm 1 `merge_`

```
1:  function MERGE__(words, weights, start, mid, end)
2:      // Input: words array of phrases with k consecutive words
3:      // weights for merging (corresponding count of phrases, merging in desc. order)
4:      // start start index for merge
5:      // mid mid index for merge
6:      // end end index for merge
7:      // Output: words array of unique phrases (locally sorted between start and end)
8:      // weights array of corresponding counts of phrases in words)
9:      for left ← start to mid do
10:         left_arr[left - start] ← (words[left], weights[left])
11:     for right ← mid + 1 to end do
12:         right_arr[right - mid] ← (words[right], weights[right])
13:     left_arr[mid + 1] ← (“ ”, 0)
14:     right_arr[end + 1] ← (“ ”, 0)
15:     left ← 0; right ← 0
16:     for counter ← start to end do
17:         if left_arr[left][1] ≥ right_arr[right][1] then
18:             words[counter] ← left_arr[left][0]
19:             weights[counter] ← left_arr[left][1]
20:             left ← left + 1
21:         else
22:             words[counter] ← right_arr[right][0]
23:             weights[counter] ← right_arr[right][1]
24:             right ← right + 1
25:     return (words, weights)
```

Algorithm 2 `merge_sort`

```
1:  function MERGE_SORT(words, weights, start, end)
2:      // Input: words array of phrases with k consecutive words
3:      // weights for sort (corresponding count of phrases, sorting in desc. order)
4:      // start start index for merge_sort
5:      // end end index for merge_sort
6:      // Output: words array of unique phrases sorted in desc. order
7:      // weights array of corresponding counts of phrases in words)
8:      if start < end then
9:          mid ← round down (start + end)/2
10:         MERGE_SORT(words,weights,start,mid)
11:         MERGE_SORT(words,weights,mid + 1,end)
12:         return (words, weights)
13:     else
14:         return MERGE_(words,weights,start,mid,end)
```

Algorithm 3 `count_vectorizer`

```
1:  function COUNT_VECTORIZER(texts, k=1)
2:      // Input: texts array of arrays with each containing the words from a review
3:      // k integer with the count of consecutive words in a phrase (defaults to 1)
4:      // Output: y array of unique phrases (with k consecutive words)
5:      // c array of occurrence count for corresponding index in y
6:      y ← [ ]
7:      c ← [ ]
8:      vals ← { }
9:      entries ← 0                                          ▷ empty Hashmap
10:     for text ← 0 to length of texts do
11:         counter ← 0
12:         len_ ← length of texts
13:         while counter ≤ (len_ - k) do
14:             for i ← 0 to k do
15:                 word ← word + texts[text][counter+i] + " "
16:             word ← strip right space from word
17:             try:
18:                 vals[word] ← vals[word] +1
19:             catch KeyNotFoundError:                  ▷ Create new Key in Hashmap
20:                 vals[word] ← 1
21:             counter ← counter +1
22:         entries ← entries + counter
```

```
23:     y ← convert keys of vals to list
24:     c ← convert values of vals to list
25:     arrlen_ ← length of y
26:     if arrlen_ = entries then
27:         return y,c
28:     return_vals ← MERGE_SORT(y,c,0,arrlen_−1)
29:     return return_vals[0], return_vals[1]          ▷ return_vals is a (y,c) tuple
```

2 Runtime-Analysis

Runtime is $\mathcal{O}(n \log(n))$

3 Correctness

Abbildungsverzeichnis

Tabellenverzeichnis