Data Structures and Algorithms I

Homework Assignment 3

Wachmann Elias

4. November 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/Phrasen `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 MERGE_(words,weights,start,mid,end)
13:    else
14:        return (words, weights)
```

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 ← { }                                          ▷ empty Hashmap
9:     entries ← 0
10:    for text ← 0 to length of texts do
11:        counter ← 0
12:        len_ ← length of texts
13:        while counter ≤ (len_ - k) do
14:            if k != 1 then
15:                word ← concatenate texts[text] from counter to counter+1 separate
   with spaces
16:            else
17:                word ← texts[text][counter]
18:            try:
19:                vals[word] ← vals[word] +1
20:            catch KeyNotFoundError:                     ▷ Create new key in Hashmap
21:                vals[word] ← 1
22:            counter ← counter +1
23:        entries ← entries + counter
```

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

2 Laufzeitanalyse

Aus der Vorlesung ist bereits bekannt, dass *Mergesort* eine asymptotische Laufzeit von $T(n) = \mathcal{O}(n \log(n))$ und einen Speicherverbrauch von $S(n) = \mathcal{O}(n)$ aufweist. `merge_sort` und `merge_` sortieren dabei die beiden Listen `words` und `weights` was zu mehreren Zuweisungen und zum Doppeltem Speicherverbrauch - im Vergleich zum *Mergesort* mit einer Liste - führt. Da jedoch Konstanten nichts an der Ordnung der Laufzeit und des Speicherverbrauchs ändern bleibt dieser ordnungsmäßig gleich.

Die gesamte Laufzeit von `count_vectorizer` ergibt sich somit wie folgt: Laufzeit bis Zeile 9 ist $\mathcal{O}(1)$, da es sich nur um Initialisierung von Variablen handelt. Die folgende For-Schleife iteriert über die Listen mit den Wörtern aus den einzelnen Reviews und hat damit $T(m) = \mathcal{O}(m)$ wobei m die Anzahl an Reviews angibt. Nun werden $len\_ - k$ ($len\_$ ... Wörter in der Review, k Anzahl der aufeinanderfolgenden Wörter pro Phrase) Phrasen erstellt und in ein Dictionary gespeichert. $len\_ - k$ wird im folgendem als l bezeichnet und beschreibt die Länge der zu sortierenden Listen. Diese While-Schleife braucht somit $\mathcal{O}(l)$ * $\mathcal{O}(n)$ (vom konkatenieren der Wörter n ist Länge der ausgegebenen Phrase) Zeit. Nach der For-Schleife werden die keys und values aus dem Dictionary in die Listen `y` und `c` gespeichert, dies benötigt lineare Zeit. Die weiteren Zeilen benötigen konstante Zeit - außer der *Mergesort* welcher $T(n) = \mathcal{O}(n \log(n))$ benötigt - und fallen somit nicht weiter ins Gewicht.

Somit ergibt sich die gesamte Zeitkomplexität zu: $T(l,m,n) = \mathcal{O}(m) * (\mathcal{O}(l) * \mathcal{O}(n)) + \mathcal{O}(l \log(l))$ Diese ergibt sich weiter zu $T(l) = \mathcal{O}(l) + \mathcal{O}(l \log(l)) = \mathcal{O}(l \log(l))$, da für jeden gegebenen Aufruf, m, eine Konstanten - je nach Anzahl der Reviews - und n ebenfalls konstant, indem man die durchschnittliche Länge der Phrasen für n verwendet, ist.

3 Korrektheit des Algorithmus

Der Algorithmus durchläuft am Anfang alle Reviews und speichert sie in ein Dictionary, dies garantiert bereits, dass sich keine Duplikate unter den Phrasen von k-aufeinanderfolgenden Worten finden. Weiter kann man dadurch auch gleich die Auftrittshäufigkeit jeder Phrase in den Reviews bestimmen. Das Dictionary enthält somit alle einzigartigen Phrasen mit ihrer Häufigkeit und es werden daraus die beiden noch unsortierten Listen `y` und `c` generiert. Kommt jede Phrase genau einmal vor, so muss die Liste zwangsläufig sortiert sein

(alle Listeneinträge in `c` sind schließlich 1) und der Algorithmus bricht vor dem sortieren ab. Sind die Listen nicht schon zufälligerweise sortiert, so werden diese mit `merge_sort` sortiert. Hier ruft sich `merge_sort` rekursiv auf und teilt die Liste dabei in eine linke und rechte Subliste bis schließlich einzelne Elemente erreicht werden, welche zurückgegeben werden. `merge_` garantiert nun die richtige Sortierung dieser. Die linke und rechte Liste werden dabei verglichen, wobei das Element mit dem größeren weight immer zuerst gestellt wird. Das einfügen eines Tupels mit einer leeren Phrase und einem weight von 0 garantiert, dass nachdem eine der beiden Listen durchlaufen ist, immer ein Element aus der noch nicht vollständig durchlaufenen Liste zur Ausgabe hinzugefügt wird. Für den Fall, dass die Elemente in der linken und rechten Liste gleiches weight besitzen wird das linke bevorzugt, wodurch die ursprüngliche Reihenfolge innerhalb der Reviews (für gleichen count/weight) erhalten bleiben.

Der Algorithmus beendet richtig, falls `merge_sort` mit zwei gleich langen Listen aufgerufen wird. Dies ist in `count_vectorizer` klarerweise der Fall, da die Listen aus dem Dictionary erstellt werden und jeder key genau ein value besitzt. Zudem wird `merge_sort` immer mit Startindex 0 und Endindex `arrlen_-1` aufgerufen, wodurch in `merge_` nie auf einen Index außerhalb der Listen zugegriffen wird. Dadurch ist garantiert, dass der Algorithmus ohne Fehler beendet. Durch die im obigen Absatz beschriebene Vorgehensweise (in `merge_`), wird ebenso garantiert, dass die Sortierreihenfolge korrekt ist.