Lab 3 Write-up

Sirajus Salekin

Algorithms:

1. **pure_dictionary():**

This algorithm uses python dictionary feature. Using a 'for' loop, this first checks for every word if the key exists in the dictionary. If not,it assigns the value of 1. Else, it adds 1 to the existing value. It returns a dictionary that is mapped in {...,word: frequency,..,} order.

Then it makes use of python's sorted() function to sort the dictionary in descending order, and finally it slices the top ten value from the sorted list and use the top ten values for presentation

1. *Strength:* dictionary feature is a hash based algorithm and its pretty time efficient. The sorted() uses Timsort which has an average case of O(n log n) time complexity, so it would be pretty fast
2. *Weakness:* the large dictionary could take up a lot of spaces, and it's not very efficient over space.

2. **heap_heapq():**

This algorithm makes use of the mapped dictionary as we did in case of pure_dictionary() algorithm. However, the difference is this time instead of using python's generic sorted() algorithm, we map the dictionary to list and heapify it to a Heap data structure. Then we pop the top ten values.

1. *Strength:* Using a heap to get the most frequent 10 values. Heap is an efficient data structure in terms of finding the max and min values. We heapify the list over O(n) time, and popping 10 most frequent word takes about O(10*log(n)) time. In total, it's (O(n) + O(log(n)) time, which is supposedly better than Timsort.
2. We have overhead of converting the dictionary to a list in order to be able to work with python's heapq library. This may kill extra time, especially in the case of a large data file.

Analysis:

We thought initially that algorithm 1 is going to take longer than usual time, since it uses the generic tim algorithm. However, after doing timing different tests, it seems that pure_dictionary, timewise, is a more viable option

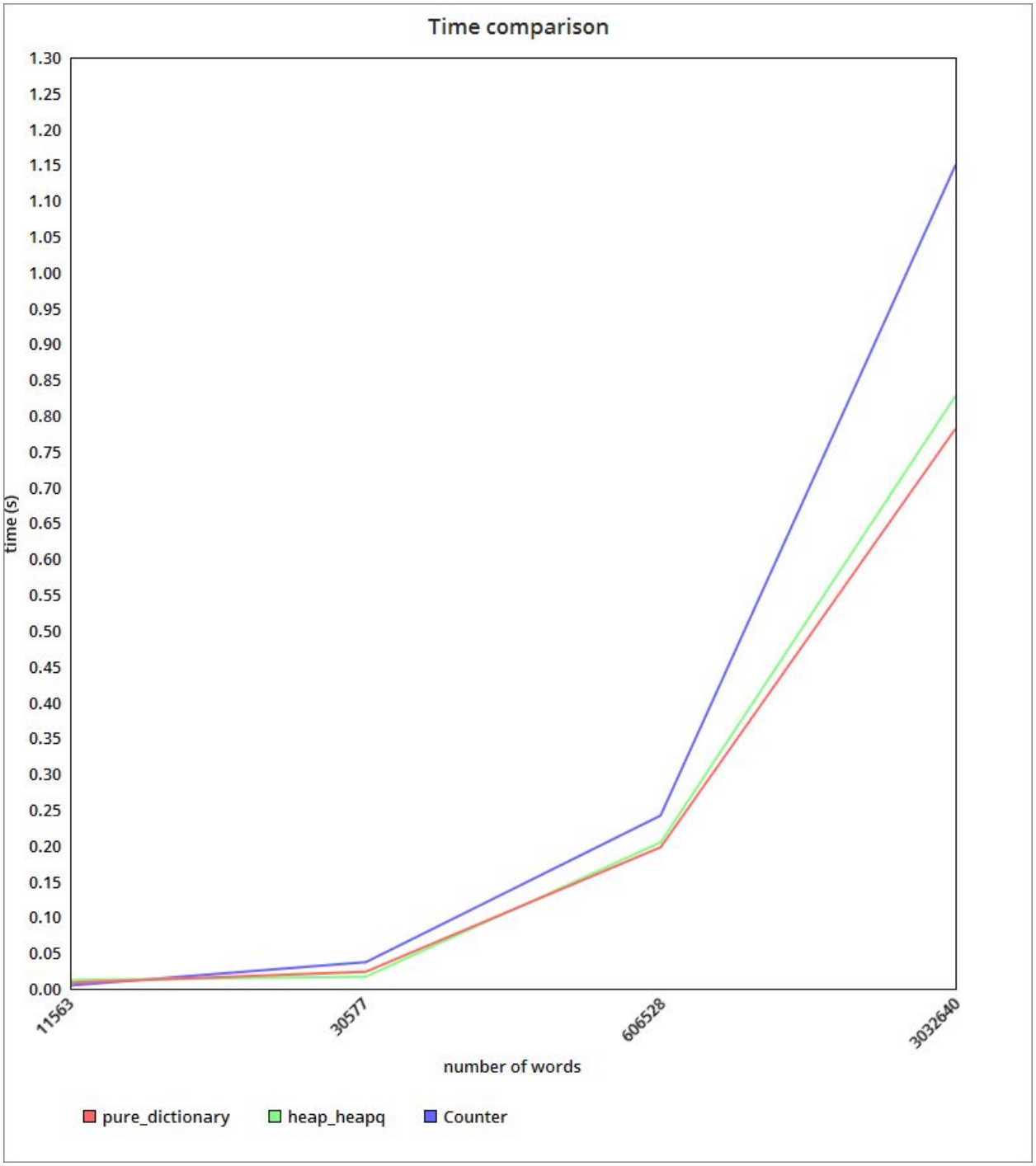

Fig: Time comparison of three algorithms

It's interesting to note that pure_dictionary starts to have advantage over heap_heapq as the the number of words grow larger. However, before about 300K heap_heapq has slight advantage over pure_dictionary. Although, they are very close in terms of performance. So I'd prefer using pure_dictionary implementation unless time is a huge issue in most cases. One important thing

to notice is Counter does better than the other two algorithm for an extremely smaller number of cases.

Testings:

1. We are keeping a record every single time the program is run, and we are able to give better insight into strength of each algorithm
2. We have some very big to some very small .txt files to see how the algorithms fare over space, and if those algorithms are good candidates for even larger files.

Stretch Goals:

1. *Adding another algorithm:* **collections_Counter()** uses the collections.Counter() function. It does all the work: it notes the frequency and produce a list that contains word and frequency of each word. It
2. *Graphical Histogram:* using matplotlib, we added a graphical interface that represents the same as ascii graph. You can resize the window to be able to see better. You need to close the window using the cross button at the left of it.
3. *File Stored at a Web URL*: You can type the url right after the program name. For example: python driver.py "http://abc.com/big.txt" and the program will fetch the file from there.
4. *Stemming*: This is a very primitive stemming function. We first went through the most common prefixes, and after that, most common suffixes. The aim was to reduce the words into a singular, simpler, mostly noun form as opposed to going to the "root" of the word. This is done considering the aim of the lab: finding the most frequent words. In order to get a better result in respect to the lab's goal while keeping only roots of the words, we need a more sophisticated algorithm, which is perhaps out of the scope of the lab.
   Usage of this feature is very easy: when you run the driver.py program, it will ask for enabling stemming. Type 'y' if you want it enabled.
5. *N-gram:* This script works the same way as the driver.py program:

   ```
   python ngramGenerator.py  words-file stop-words
   ```

   It will ask for the length of n-gram you want to generate, and then present the result in a histogram format.