Programming with Data

Coursework 2: Problem solving and scalability analysis

Guillermo Fremd Kanovich

16th March 2019

Birkbeck College, University of London

Academic Declaration

I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software.

- ***Phase 1: MyHealthcare device: Vital signs simulator***

Funcion myHealthcare(n) creates a dataset of n rows, and 8 columns, using randon.randit and random.uniform to create the respective integer and float records. The first column, timestamp, is the only one that is not crated randomly.

- ***Phase 2: a) Find abnormal values for pulse***

I created two different functions presenting abnormal pulse values:

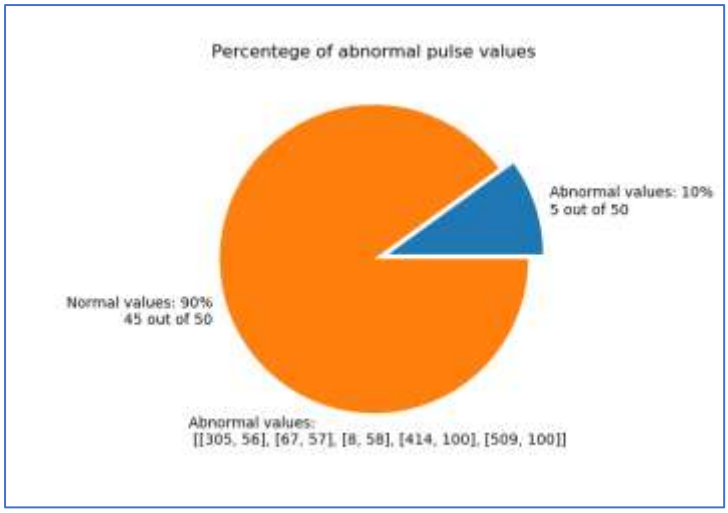
1. Function AbnormalPulseAnalytics_basic (table) uses simply for loop that goes over each value of the sample, and if it is an abnormal value (meaning, above 99 or below 60), adds it into a new list of abnormal values. This is a linear search algorithm. Thus, its cost is O(n)
2. In turn, function AbnormalPulseAnalytics2(table) sorts the values using a merge-sort algorithm I created for lists of lists (mergesortLoL), which sorts the data set using a merge-sort algorithms, not necessarily according to the values of the first column, but according to the column one choses. After sorting, the AbnormalPulseAnalytics2 function does two binary searches: one two identify the abnormally high valuesbinarySearchExtremeHigh, and one to identify the abnormally low values (binarySearchExtremelow). Then, I merge the results of both functions to see have all the abnormal values together. It is relevant to note that, given that the "basic" binary search only identifies one value matching the searched one, I developed two additional functions, named subsetleft and subsetright, which return subsets with all records with pulse values less than 60 or more than 100 respectively. Given that it is a binary search, its computational cost is O(Log n). More details about the specific of the sorting and search algorithms have been included as comments in the python code.

- ***Phase 2: b) Present a frequency histogram of pulse rates***

Function frequencyAnalytics (table) uses a linear search (using a For loop), and therefore its cost is O(n)

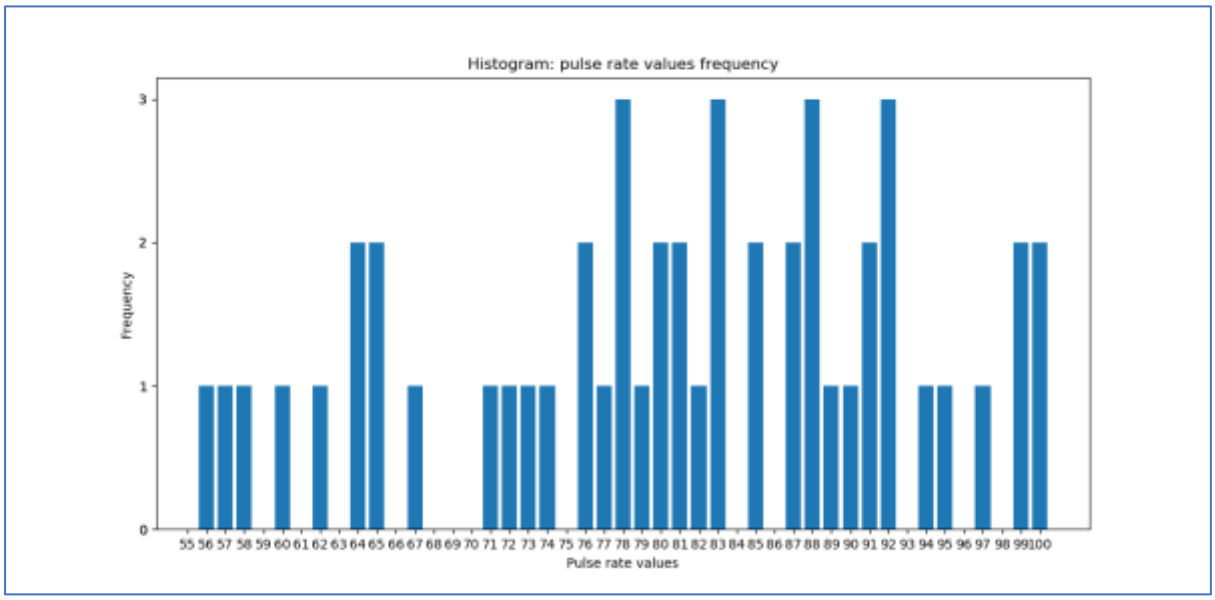
- ***Phase 2: c) Plot the results for 2a and 2b and briefly discuss your observations.***

I used matplotlib to plot my results.
For the plotting of the abnormal pulse values (plot_AbnormalPulseAnalytics), I plotted a pie chart, showing the percentage of abnormal values on the total, and listing all the identified abnormal values, with its respective timestamp:



As shown in the chart, 5 out of the 50 records of the sample present abnormal values for Pulse (10% of the total

For the plotting of the histogram (plot_frequencyAnalysis) I used a bar chart. Given that the data was generated randomly and not with a Normal distribution, the histogram is not "bell-shaped", but, on the contrary, the values appear to be more or less evenly distributed among the range (55 to 100)



- **Phase 2: d) What is the computational cost (as a function of n) of your solution?**

The computational cost of the solutions is commented above.

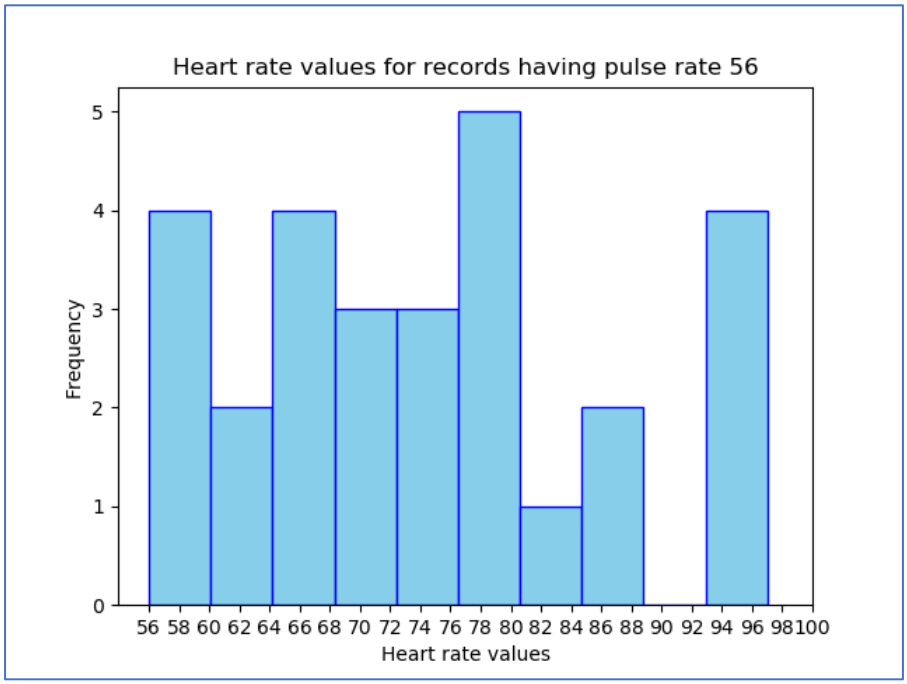
- ***Phase 3: Search for heart rates using the HealthAnalyzer***

In this case, I also created two different solutions:

1. healthAnalyzer_basic (table,value, column) simply does a linear search in the list of list "table", in column "column", and append all values that are equal to "value" in a new list. This is a linear search algorithm and, its cost is O(n)

2. healthAnalyzer2(LoL,value,column), in turn, first uses a the binary search algorithm for lists of lists commented before (mergesortLoL), and the performs a binary search in the relevant column to search for one records having such value. Once it identifies such value, it makes use of another algorithm I created, named LookLeftRight, which searches to the left and to the right of such records all the "neighbour" records that have the same value. While the use of a binary search is intended to enable the function to have an O(Log n) computational cost, in the worst case scenario, when all the values equals to the searches value, the function “LookLeftRight” are almost identifcal to a linear search, and thefore the cost of the function is O(n).

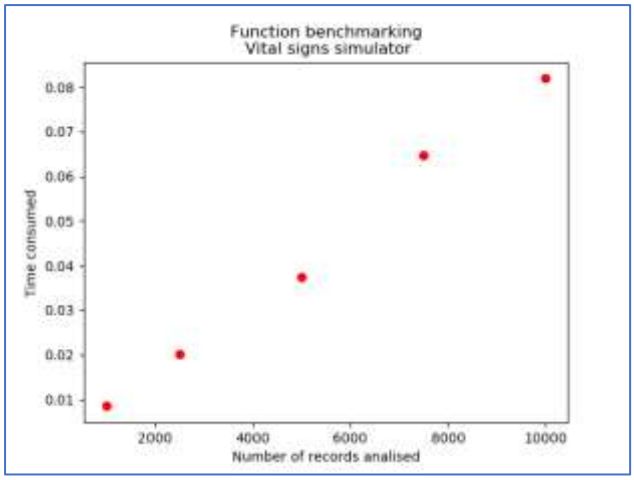
I created an histogram plotting the frequency of the obtained values:



Given that the data was generated randomly and not with a Normal distribution, the histogram is not “bell-shaped”.

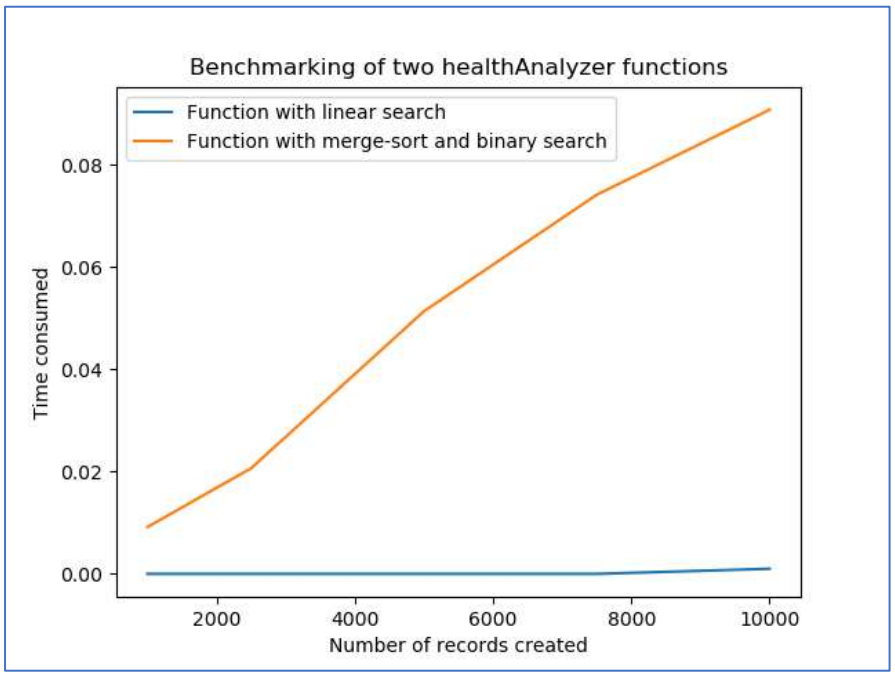
- ***Phase 4: Testing scalability of your algorithm (20 marks). Benchmark the MyHealthData application simulating n = 1000, 2500, 5000, 7500 and 10000 records from phase 1***

While function benchmarking(myHealthcare) does the benchmarking calculation of the datacreation function, it is then plotted with the value function plotbench(timeskeeping). The result is presented below:



The plot obtained suggest that time needed for data generation grows linearly with the number of records generated.

In addition, I also benchmarks the two solutions I created for the healthAnalyzer function (the one using a linear search method, and the one using a merge-sort and binary search), which were discussed above in Phase 3. While function bench_healthAnalyzer does the benchmarking calculation of the both healthAnalyzer functions, its results are then plotted with the value function plot_bench_healthAnalyzer(timeskeeping). The result is presented below:



Contrary to my expectations, the solution making use of a linear search is much faster than the one making a merge-sort and binary search. Not only the linear-search solution is significantly faster for any number of records, but also the time required for the binary search increases at a rate significantly higher. This suggests that, in this case, the more complex solution, requiring several steps (which were intended to and reduce the computational cost from O(n) to O(log n) was not effective, possibly precisely due to the large amount of steps that the sort-merging and the binary searches for list of lists created comprise.