Computing the Pearson Correlation Coefficient of a Matrix and a Vector

Elvin Marc M. Bautista

ABSTRACT

This paper presents the execution time of a C programming language function that computes the Pearson Correlation Coefficient of a Matrix and a Vector. I examined the connection between the algorithm's time complexity and actual run time. I also presented ways on how to improve the performance of the algorithm without utilizing extra computer processors.

Keywords

Pearson Correlation Coefficient, Matrix, Vectors, C Programming Language, Time Complexity

1. INTRODUCTION

The Pearson Correlation Coefficient is used to measure the relationship between two variables. This is often denoted as r and can have values ranging from -1 to 1, where 0 indicates no relationship while as the value approaches 1, it indicates a higher correlation between the variables. The variables have negative correlation as r approaches -1. The formula for calculating the Pearson Correlation Coefficient is given by

$$r = \frac{n\sum(xy) - (\sum x)(\sum y)}{\sqrt{(n\sum x^2 - (\sum x)^2)(n\sum y^2 - (\sum y)^2)}}$$

where r represents the Pearson Correlation Coefficient, x and y are data points and, n is the number of data points. The algorithm will compute the Pearson Correlation Coefficient of two given vectors. The vectors x will come from the columns of the given matrix. Finally, the result of each computation will be stored in a separate vector.

This research aims to identify the time complexity of the algorithm that will compute for the Pearson Correlation Coefficient of a vector from an $n \times n$ matrix with an $n \times 1$ vector. Additionally, the research will identify the maximum feasible value of n for which the algorithm can compute. Finally, the research will explore methods to improve the algorithm's time complexity without requiring additional computer processors or cores.

2. METHODOLOGY

2.1. Pearson Correlation Coefficient Calculation Algorithm

The algorithm was implemented in C Programming Language. Below is a pseudocode of the algorithm.

```
func pearson_cor(mat: matrix, vecY: array, vecX: array, size: integer)
    for i from 0 to size - 1 do
        sumX, sumY, sumX2, sumY2, sumXY := 0
        for j from 0 to size - 1 do
            sumX += mat[j][i]
            sumY += vecY[j]
            sumX2 += mat[j][i] * mat[j][i]
            sumY2 += vecY[j] * vecY[j]
            sumXY += mat[j][i] * vecY[j]
        end for
        numerator := size * sumXY - sumX * sumY
        denominator := sqrt((size * sumX2 - sumX^2) *(size * sumY2 - sumY^2))
        vecX[i] := numerator / denominator
    end for
end
```

Listing 1: Pearson Correlation Coefficient Algorithm

There are two nested loops, the outer loop is used for going through all the columns of the matrix. The selected column in this loop is the vector that will be compared to the given vector Y. The inner loop is used to calculate the summations of the items in the vector Y and the currently selected vector by the outer loop. After computing the summations needed, the algorithm assigns a numerator and a denominator. Finally, the Pearson Correlation Coefficient will be computed by dividing the numerator by the denominator, the resulting value of this division will be stored in the given vector X.

The size of the matrix and the vector Y are not predetermined. The program asks for a user input and dynamically allocates a matrix and a vector with a size$n$. The values for the dynamically allocated matrix and vector are randomized using the C library function rand(). To ensure that all the values will be non-zero, the values are only assigned after applying modulo 50 and incrementing it by 1. Therefore the values inside the matrix and the vector are from 1-50. Recording the execution time of the calculation of Pearson Correlation Coefficient of a matrix and a vector is crucial in this research, and to take note of the time elapsed by each run, the C Library Function clock() is called and the value is stored in t, the current time before calling the function for calculating the coefficient. After the execution of the function, the value of t is updated by clock()$-t$. Finally to convert it into seconds, t is divided by CLOCKS_PER_SECONDS.

2.2. System Specifications

The experiments were conducted on a Laptop Computer with the following specifications

- Processor: Intel(R) Core(TM) i5-10300H @2.50GHz, 2496Mhz, 4 Cores, 8 Logical Processors
- Memory: 16 GB DDR4 RAM, 2933MHz
- Storage: 256 GB SSD; 500 GB SSD
- Operating System: Windows 10
- Compiler: GCC version 9.4.0

3. RESULTS AND DISCUSSION

The program was run three times for each given n. The recording of running time of the function to compute the Pearson Correlation Coefficient started when n was set to 100. The n used for each run of the program is increases, up to the n that the program can handle. The algorithm for calculating the Pearson Correlation Coefficient of a matrix and a vector has a time complexity of $O(n^2)$. As seen on the pseudocode for the calculation of Pearson Correlation Coefficient of a matrix and a vector, where there are two nested loops that run up to the size, which is essentially the given n[2].
The following tables show the elapsed time of the function and the average of each run given a value of n.

n	Time Elapsed		
	Run 1	Run 2	Run 3
100	0.000166	0.000184	0.000188
200	0.000485	0.000425	0.000522
300	0.000927	0.00092	0.000923
400	0.001737	0.001585	0.001642
500	0.002366	0.002382	0.00245
600	0.003655	0.003465	0.003944
700	0.005084	0.005311	0.005282
800	0.006705	0.006538	0.007225
900	0.008926	0.008388	0.008685
1000	0.010415	0.010829	0.011728
2000	0.062075	0.061926	0.06146
4000	0.256263	0.257	0.255554
6000	0.58125	0.586632	0.590124
8000	1.102023	1.108879	1.12298
16000	6.636051	6.486363	6.578398
20000	15.213005	15.038196	15.805107

Table 1: Program Run Times

The table shows the run times for each execution of the program, recorded in seconds. The only recorded values of n are from 100 up to 20,000. It can be observed that the three different instances of execution of the same program with the same n does not have the same running time. It is especially noticeable at $n = 20,000$, where the execution time of the 2nd run of the program and the execution time of the 3rd run of the program differs by approximately 0.8 seconds. Such differences can be pointed to the CPU scheduling. The program was able to handle $n = 50,000$ and even $n = 100,000$. However, the program failed to run to completion when the given $n = 10,000,000$. The process was killed during the randomization of items in the matrix. The memory allocation of a $10,000,000 \times 10,000,000$ matrix consumed a lot of memory and was killed by the kernel. After the process was killed at $n = 10,000,000$, the command `dmesg` is executed and it can be seen at the diagnostics that the OOM killer killed the `a.out` process. The `a.out` process is the process that runs the computation of the Pearson Correlation Coefficient. This diagnostic message informs that the system has ran out of memory or is critically low on memory. This happens when a process requires more memory than the available physical memory. The OOM killer is then tasked to terminate the process that takes up a lot of the memory to free that memory in order for the system to continue running [3]. In this case, the process `a.out` was killed. Since the termination of the process was due to a lack of memory, a solution for running the program at $n > 10,000,000$ is to run the program in a different system that has more memory than the once used in the experiment. Additionally, better memory allocation and usage may be required in order to run the program at $n > 10,000,000$ in the same machine.

n	Average (Seconds)	Complexity
100	0.0001793333333	10000
200	0.0004773333333	40000
300	0.0009233333333	90000
400	0.001654666667	160000
500	0.002399333333	250000
600	0.003688	360000
700	0.005225666667	490000
800	0.006822666667	640000
900	0.008666333333	810000
1000	0.01099066667	1000000
2000	0.06182033333	4000000
4000	0.2562723333	16000000
6000	0.586002	36000000
8000	1.111294	64000000
16000	6.566937333	256000000
20000	15.35210267	400000000

Table 2: Average Program Run Time and Complexity

The table shows the average runtime of the three runs, it also includes the complexity where n is squared, since it was established that the complexity of the algorithm is $O(n^2)$. Plotting the average run time and the complexity of the program against n will yield these graphs.

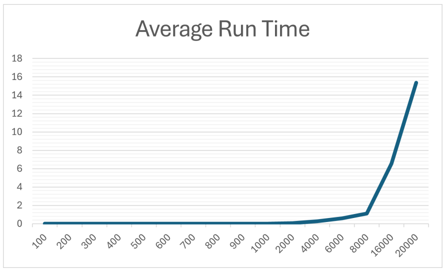

Figure 1: Average Run Time Graph

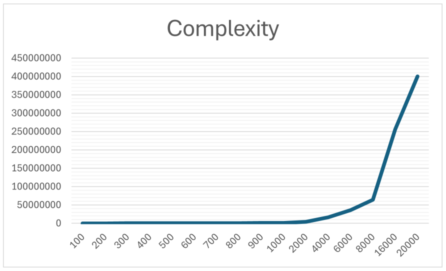

Figure 2: Complexity Graph

The x-axis of both of the graphs represents the given n. The y-axis of the Figure 1 is the average run time of the program given n, while the y-axis of the Figure 2 is n^2, illustrating the worst case running time of the algorithm. Upon inspection of the graphs, it can be observed that the Average Run Time and the Complexity of the program given n are related. The trend and form of the line are similar even though the y-axis of the graphs have significantly different values. This demonstrates the complexity of the program to be $O(n^2)$ since the average running time of the program exhibits a similar trend to the complexity when converted into time.

In conclusion, considering the $O(n^2)$ complexity of the algorithm, it becomes evident that there are opportunities for developers to enhance its average running time. One avenue for improvement involves leveraging multiple processors or cores. Alternatively, optimizing the algorithm by moving certain calculations outside the nested loop, such as the summation of items in vector Y, could significantly enhance its efficiency. By performing these calculations once in a separate loop, rather than within the inner loop where they are redundantly computed n times, considerable performance gains can be achieved. Additionally, beyond mere code adjustments, exploring alternative programming languages that offer superior performance compared to C could further reduce the average run time. These strategies collectively offer promising avenues for optimizing the algorithm's efficiency and improving its overall performance without necessitating additional hardware resources.

References

[1] S. Boslaugh and P. A. Watters, *Statistics in a Nutshell*. Sebastopol, CA, USA: O'Reilly Media, 2008. [Online]. Available: `http://ci.nii.ac.jp/ncid/BB11078099`

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[3] "Linux Out of Memory Killer - Knowledge Base," *Neo4j Graph Data Platform*. [Online]. Available: `https://neo4j.com/developer/kb/linux-out-of-memory-killer/`