Final_Report

Arnav Bhutani

12/6/2020

The initial purpose of this project was to create a non-package implementation of the viterbi algorithm, and then parallelize it on an Kubernetes network that I had previously set up. This would involve actually researching the Viterbi algorithm to figure out how it works, write the code and then finding some way to parallelize the core workload that drove it. The problem is discussed in more detail in the project_walkthrough.pdf.

Unfortunately, I was unable to use the algorithm on Kubernetes, as I ran out of time, but I was able to implement the Viterbi algorithm. I won't go into the specifics of what I learned or implemented here, but if you are curious, feel free to visit the project_walkthrough.pdf document. In the end, I was able to construct an implementation of the Viterbi Algorithm that was just as accurate as the HMM package's viterbi algorithm with a slightly faster runtime, and a much narrower HMM scope. My package will only process HMMs that have three visible states and two hidden states. I ran both my and the HMM package's viterbi algorithm over 100 simulated HMMs with different lengths to produce the plot figure 1. In the figure, figure 1, you may notice that the accuracy between the two packages is remarkably close. The minute differences suggest that the HMM package uses a similar function to the one that I use to implement the Viterbi algorithm, as many of the results very by less than a percentage point. The runtime of the algorithms created can be viewed in plot two. It is interesting to see that the c implementation of the algorithm is slower than both my and the HMM packages' implementation. This could be because I was required to use three for loops in the c implementation that I wrote, opposed to the two loops that are used in the pure R implementation. This speaks to how the flexibility of matrix multiplication in R. Similarly it is interesting to see that my implementation is faster than the HMM package's implementation. This could be due to the narrow scope of my algorithm, and that only implementing my algorithm for 2 hidden steps and 3 visible steps allowed me to 'hard-code' efficiency into the function.

In the future I would love to extend this package to encompass more HMM's by making the viterbi algorithm definition far more dynamic than it is today. I could do this by dynamically assigning hidden state names based on a variable that is passed in, basing the BaumWelch algorithm's convergence on an actual difference in the maximization statistic, rather than iterations. Lastly, I would love to parallelize and run this algorithm on the cloud. This would require me to implement some task scheduling for the more time intensive processes in the algorithm so that it can scale with cluster size.

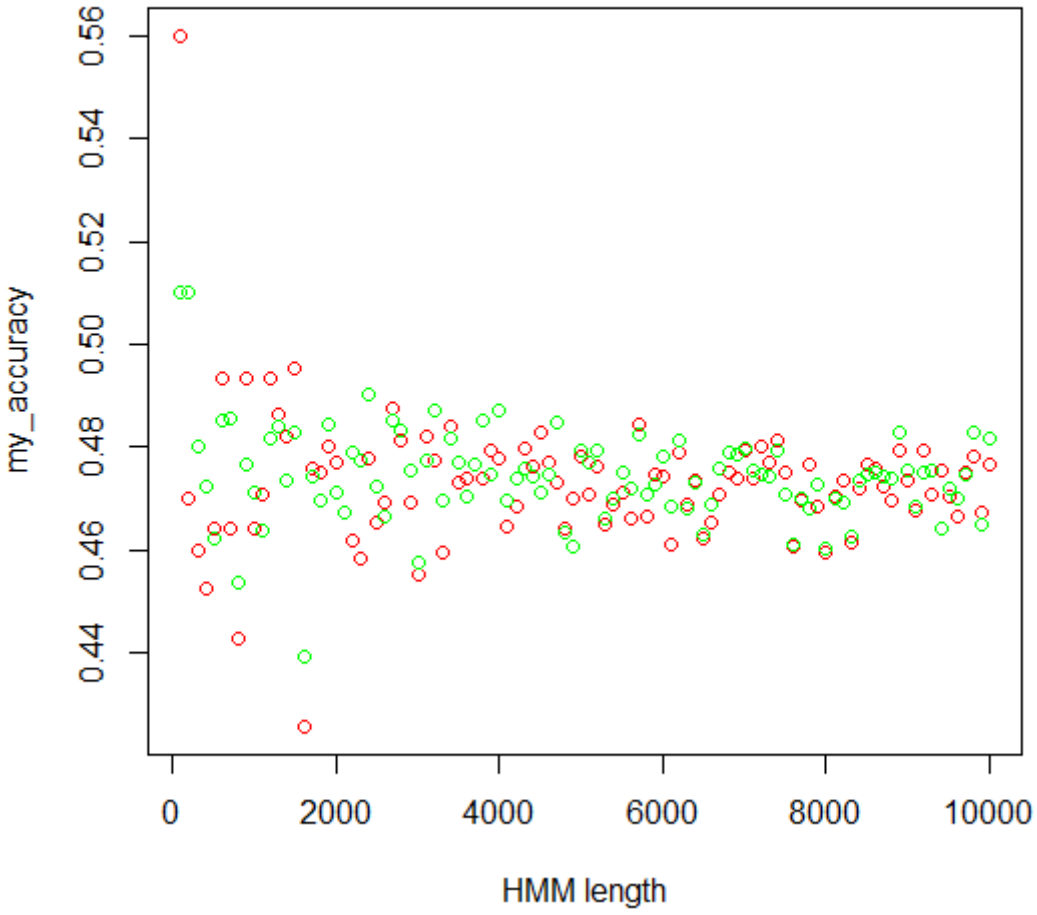

Figure 1: Accuracy of implementation (red) vs. HMM package (green)

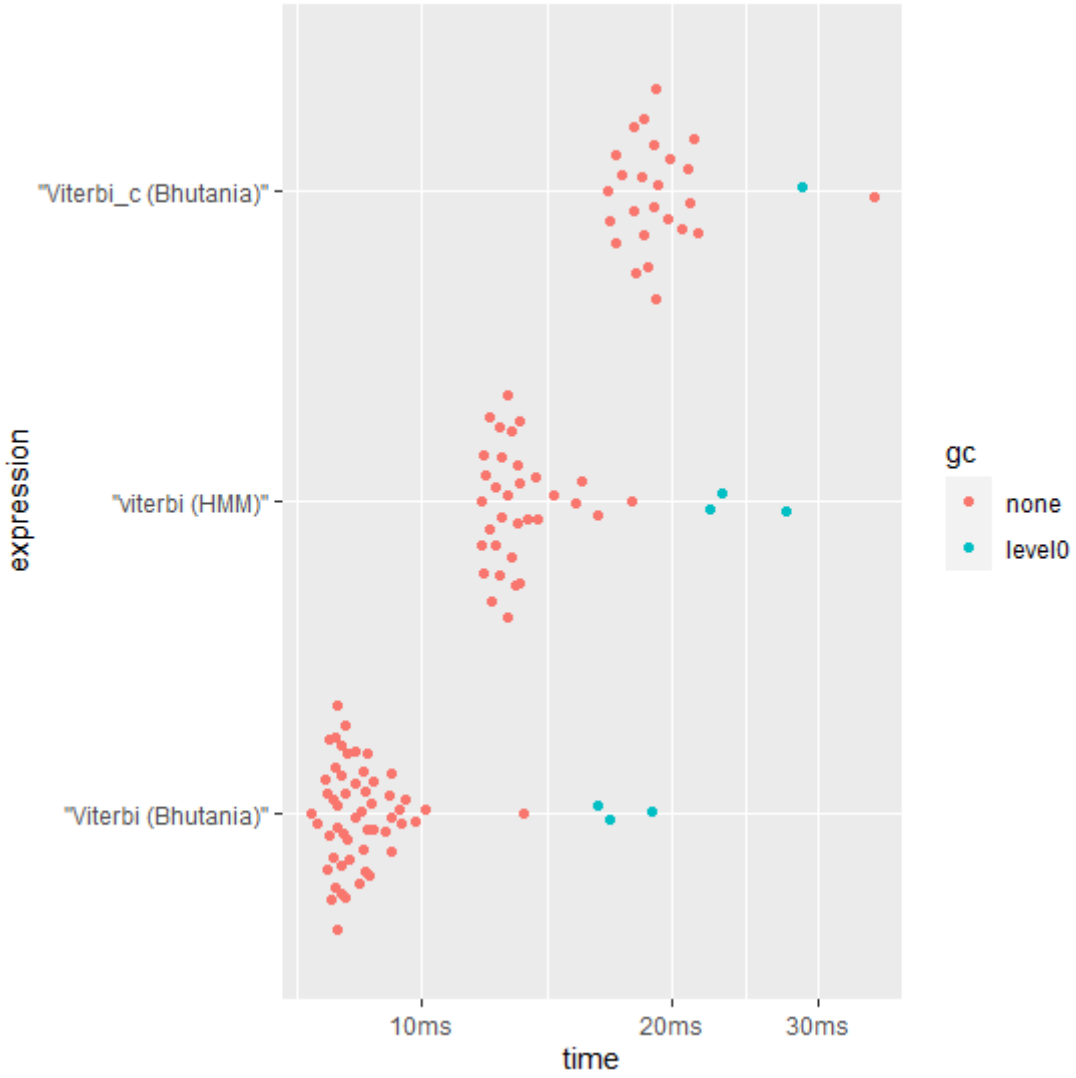

Figure 2: Timing of implementations vs. HMM package