project_walkthrough

Arnav Bhutani

12/6/2020

```
library(here)
```

```
## here() starts at C:/Users/arnav/Documents/ST441/bhutania-project-viterbi
```

```
devtools::load_all()
```

```
## Loading BhutaniaViterbi
```

I have created a package with four different functions: forward backward BaumWelch viterbi

These functions may be viewed in the /R/ directory from the base project path. For ease, I have also shown them here.

```
BhutaniaViterbi::forward
```

```
## function(example, switching, visible, init) {
##   #Initialization
##   example_length = length(example)
##   hidden_states = nrow(switching)
##   example_p = matrix(0, example_length, hidden_states)
##
##   #The first state is determined using a
##   #initial chance * prob(visible_state| hidden state)
##   example_p[1, ] = init*visible[ , example[1]]
##
##   #calculate the probability of the next result being one of the
##   #hidden states based on the probabilty of switching hidden states,
##   #and the probability of those hidden states producing each visible
##   #state.
##   for(t in 2:example_length)
##   {
##     tmp = example_p[(t-1), ] %*% switching
##     example_p[t, ] = tmp * visible[ ,example[t]]
##   }
##   return(example_p)
## }
## <environment: namespace:BhutaniaViterbi>
```

```
BhutaniViterbi::backward
```

```
## function(example, switching, visible, init)
## {
##   example_length = length(example)
##   hidden_states = nrow(switching)
##   # The last state is given, so it has a p() of 1.
##   example_p = matrix(1, example_length, hidden_states)
##
##   #calculate the probabiity of the previous hidden state based on the
##   #example visible state.
##   for(t in (example_length-1):1)
##   {
##     tmp = as.matrix(example_p[t+1, ] * visible[, example[t+1]])
##     example_p[t, ] = t(switching %*% tmp)
##   }
##   return (example_p)
## }
## <environment: namespace:BhutaniViterbi>
```

```
BhutaniViterbi::BaumWelch
```

```
## function(example, switching, visible, initial_distribution, n.iter = 100){
##   for(i in 1:n.iter){
##     #initialization
##     time_step = length(example)
##     hidden_s = nrow(switching)
##     visible_s = ncol(visible)
##
##     #get the probabilities of each timestep forward and backwards
##     #based on the current switching and visible probabilities
##     alpha = BhutaniViterbi::forward(example, switching, visible, initial_distribution)
##     beta = BhutaniViterbi::backward(example, switching, visible)
##
##     #array that contains the probability of switching from one state
##     #to another for each time step.
##     xi = array(0, dim=c(hidden_s, hidden_s, time_step-1))
##
##     #For each timestep, calculate:
##     #denominator: the probability of a visible state at time step
##     #             t from all hidden states at time step t-1.
##     #numerator: the probability of the example visible state at
##     #           time step t, being a result of a switch from
##     #           each state at time step t-1.
##     #xi[s,,t]:  The probability of switching from hidden state s
##     #           to another hidden state at time step t
##     for(t in 1:time_step-1){
##       denominator = ((alpha[t,] %*% switching) * visible[,example[t+1]]) %*% matrix(beta[t+1,])
##       for(s in 1:hidden_s){
##         numerator = alpha[t,s] * switching[s,] * visible[,example[t+1]] * beta[t+1,]
##         xi[s,,t]=numerator/as.vector(denominator)
##       }
##     }
```

```
##
##     #The probability of switching from hidden state s to another
##     #hidden state for all time steps
##     xi.all.t = rowSums(xi, dims = 2)
##     #The probability of switching from hidden state s
##     switching = xi.all.t/rowSums(xi.all.t)
##
##     #Find the probability of being in a hidden state at time
##     #state t
##     gamma = apply(xi, c(1, 3), sum)
##
##     #use gamma to find the probability of a hidden state
##     #producing a visible state l
##     gamma = cbind(gamma, colSums(xi[, , time_step-1]))
##     for(l in 1:visible_s){
##       visible[, l] = rowSums(gamma[, which(example==l)])
##     }
##     visible = visible/rowSums(visible)
##
##   }
##   return(list(switching = switching, visible = visible, initial_distribution = initial_distribution))
## }
## <environment: namespace:BhutaniaViterbi>
```

```
BhutaniaViterbi::Viterbi
```

```
## function(example, switching, visible,initial_distribution) {
##   #initialization
##   time_step = length(example)
##   hidden_s = nrow(switching)
##   prev = matrix(0, time_step-1, hidden_s)
##   omega = matrix(0, hidden_s, time_step)
##
##   #Log likelyhood to ease computation
##   omega[, 1] = log(initial_distribution * visible[, example[1]])
##
##   #Find the most likely hidden step for time step t.
##   for(t in 2:time_step){
##     for(s in 1:hidden_s) {
##       #Find the probability of the next switch
##       probs = omega[, t - 1] + log(switching[, s]) + log(visible[s, example[t]])
##       #Find the largest probability of the switch
##       prev[t - 1, s] = which.max(probs)
##       #Find the probability of the most probable switch
##       omega[s, t] = max(probs)
##     }
##   }
##
##   S = rep(0, time_step)
##   #Most probable last hidden state
##   last_state=which.max(omega[,ncol(omega)])
##   S[1]=last_state
##
##   #Find most probable route back to front
```

```
##   j=2
##   for(i in (time_step-1):1){
##     S[j]=prev[i,last_state]
##     last_state=prev[i,last_state]
##     j=j+1
##   }
##
##   #Set hidden states
##   S[which(S==1)]='A'
##   S[which(S==2)]='B'
##
##   #reverse the list
##   S=rev(S)
##
##   #return the list
##   return(S)
## }
## <environment: namespace:BhutaniaViterbi>
```

I sourced these algorithms from this set of articles about HMM's, expectation maximization, and the Viterbi algorithm. I have used their code as reference, and have attempted to make improvements to the Viterbi algorithm described in these articles using the techniques that we have learned in class.

Before diving into the algorithms, I would like to explain what the Viterbi algorithm is, and by extension what an HMM is. A Hidden Markov Model (HMM) is a two state model, where there are visible states, and hidden states. Visible states are states that can be observed, while hidden states are a related variable that can affect the outcome of the visible state, but are not directly observable themselves. These variables can be repeatedly sampled over a time (or similar consistently repetitive) series, creating what we call a Hidden Markov Model (HMM). A common example of a Hidden Markov Model is in speech recognition, where sound waves are used as the 'visible' state, while the characters and words that make up the sound byte are considered to be the 'hidden' state. To recognize speech, researchers use the Viterbi algorithm to find the most likely set of words or characters that make up the observed sound waves.

The Viterbi algorithm is a powerful tool for analysis and allows one quickly to break down chunks of data into related meaningful observations. The viterbi algorithm does this by finding the most probable path of 'hidden' variables that result in the observed path of 'visible' variables, given the probability of switching from a hidden state to another hidden state over any given time series, and the probability of a given hidden state producing each visible output. The viterbi algorithm then uses the recursive algorithm:

$p_h(v,t) = p_h(v) * max_k(p_k(j,t-1) * p_{kh})$

Where h is a hidden state, v is a given visible state, t is the time, k is the most probable previous hidden state, j is k's corresponding visible state, p_{kh} represents the vector of probabilities of switching from state k to state h.

This algorithm can then be converted to look like:

$p_h(v,t+1) = max_h(p_h(v,t) * p_{hk} * p_{kj})$

Where p_{hk} is the probability of h transitioning to k, and $p_{kj} is the probability of k producing visible output j. Thus should we be given the first or last step of the equation, we can find the most probable sequence. This exactly what the following loop sampled from viterbi.R does:

```
# Find the most likely hidden step for time step t.
for (t in 2:time_step) {
    for (s in 1:hidden_s) {
        # Find the probability of the next switch
```

```
        probs = omega[, t - 1] + log(switching[, s]) + log(visible[s, example[t]])
        # Find the largest probability of the switch
        prev[t - 1, s] = which.max(probs)
        # Find the probability of the most probable switch
        omega[s, t] = max(probs)
    }
}
```

The rest of the algorithm, just converts the results of this loop into a readable result.

I will not go into quite as much focus on the other three functions, as they were not the focus of my research, but I will go over their objectives.

The BaumWelch algorithm attempts to run an expectation maximization on the parameter:

$$\frac{p_h(t)*p_{hk}*p_{kj}*p_k(t+1)}{\sum_{h=1}^{m}\sum_{k=1}^{m} p_h(t)*p_{hk}*p_{kj}*p_k(t+1)}$$

such that p_{hk} and p_{jk} are incrementally modified to maximize the probabilities of each hidden state producing the visible state over the entire time sequence. For more information, visit: expectation maximization.

The forward and backward algorithms are reversals of each other. They estimate the probabilities of each hidden variable at a time step based on a given a switching matrix and a visible matrix. The forward algorithm starts at the first time step, while the backward algorithm starts at the last time step. For more information visit: forward and backward algorithms.

The algorithms performed admirably with the viterbi algorithm matching the accuracy of the HMM package, a R package that can construct HMMs and path trace using the Viterbi Algorithm. An acuracy graph one run can be seen in figure 1 below:

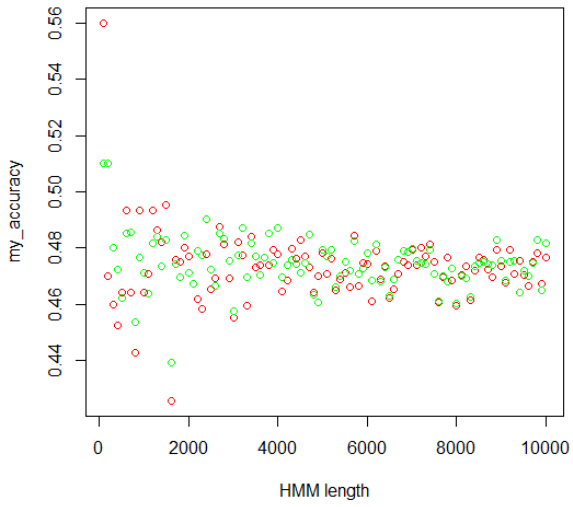

Figure 1: Accuracy of implementation (red) vs. HMM package (green)

This plot shows the results of comparing each viterbi algorithm against a HMM constructed by the simHMM function from the HMM package, with parameters trained from the Baum-Welch algorithm on the first example dataset. Each of the 100 HMMs tested were constructed with lengths ranging from 100 to 10000, with increments of 100. The accuracy was determined by counting each hidden state that was different from the constructed HMM, and then dividing by the length of the constructed HMM. Both algorithms hovered

around 46-48% accuracy, which isn't exactly impressive. If I had more time, I would like to look into why these accuracies are so low.

The timing for my viterbi implementation is also superior as seen in figure 2 below:

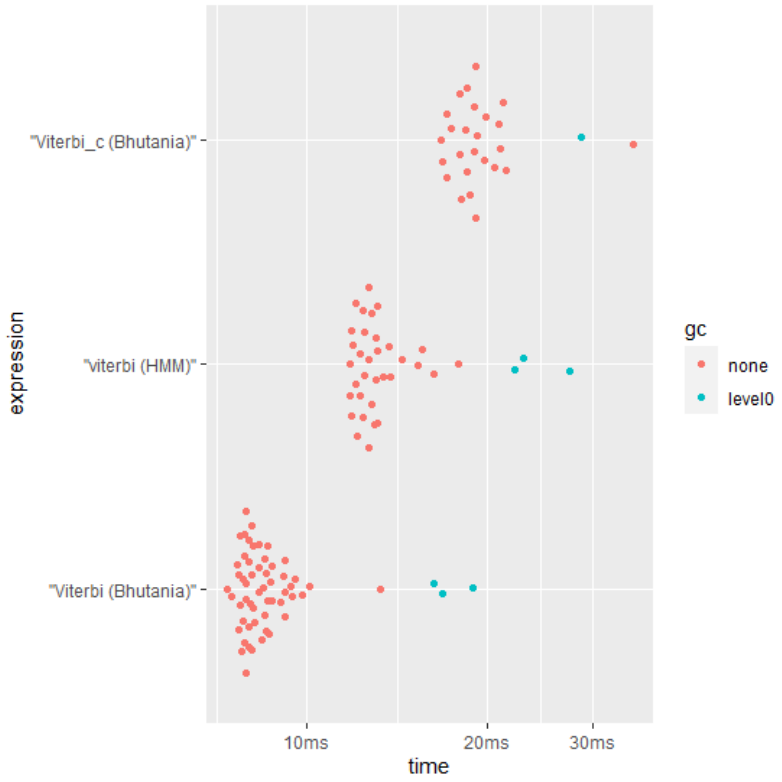

Figure 2: Timing of implementations vs. HMM package

These timings were determined by running three different bench::mark timings on the original dataset. The timings were then plotted with the beeswarm plot method. It is interesting to note that the C implementation that I constructed was actually slower than the pure R implementation. The superior speed of my pure R package in comparison to the HMM package could be because my algorithm only has to deal with the possibility of an HMM with two hidden states and three visible states, thus allowing me to initialize and loop through my variables when the function is constructed by the compiler, as opposed to a potential dynamic approach that the HMM package might use which would require some sort of runtime or linker setup, which would take more time.

Now lets look specifically at the C implementation of the Viterbi algorithm. The only real changed part is where this section of the R code:

```
# Find the most likely hidden step for time step t.
for (t in 2:time_step) {
    for (s in 1:hidden_s) {
        # Find the probability of the next switch
        probs = omega[, t - 1] + log(switching[, s]) + log(visible[s, example[t]])
        # Find the largest probability of the switch
        prev[t - 1, s] = which.max(probs)
        # Find the probability of the most probable switch
        omega[s, t] = max(probs)
    }
}
```

is replaced with:

```
cppFunction("
    using namespace Rcpp;

    // [[Rcpp::export]]
    void calculate_P(
    int hidden_s, int time_s, NumericMatrix &prev_m,
    NumericMatrix &prob_m, NumericMatrix switching,
    NumericMatrix visible, NumericMatrix example
    ){
      for(int i = 1; i < time_s; i++)
      {
        for(int j = 0; j < hidden_s; j++)
        {
          NumericVector probs = {};
          for(int k = 0; k < hidden_s; k++)
          {
            probs.insert(k, (prob_m(k, (i-1)) + log(switching(k, j)) +
                             log(visible(j, example[i]-1))));
          }
          prev_m(i-1, j) = which_max(probs);
          prev_m(i-1, j)++;
          prob_m(j, i) = max(probs);
        }
      }
    }")

calculate_P(hidden_s, time_step, prev, omega, as.matrix(switching), as.matrix(visible),
    as.matrix(example))
```

The two portions of code give the same result, however the C++ code requires an extra for loop loop, while the R code uses matrix addition which may account for the extra time taken during this operation.

In conclusion, the package that I have written evaluates 2, 3 transition, emission HMM's faster than the standard HMM package, and is just as accurate. While I believe there are ways to improve the package, such as through broadening the scope of HMM's evaluated, and potentially writing smaller loops in C++ to squeeze out more performance from the functions, I believe the result I have achieved is satisfactory for the scope of the project. Unfortunately I was unable to adapt this package to run on a distributed Kubernetes cluster, but I plan to do so in the future with a python implementation.