Advanced Machine Learning - lab02

Lennart Schilling (lensc874)

2019-09-29

Contents

Lab02: Hidden Markov Models

You are asked to model the behavior of a robot that walks around a ring. The ring is divided into 10 sectors. At any given time point, the robot is in one of the sectors and decides with equal probability to stay in that sector or move to the next sector. You do not have direct observation of the robot. However, the robot is equipped with a tracking device that you can access. The device is not very accurate though: If the robot is in the sector i, then the device will report that the robot is in the sectors [i-2, i+2] with equal probability.

Assignment 1. Building a HMM.

Build a hidden Markov model (HMM) for the scenario described above.

```
# Loading library.
library(HMM)

# Initializing possible states / hidden variables.
states_val = c(1:10)

# Initializing possible emissions / observable variables.
emissions_val = c(1:10)

# Initializing starting probabilities of the states.
states_startProbs = rep(0.1, 10)

# Initalizing transition matrix.
transition_matrix = matrix(data = c(0.5, 0.5, 0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0.5, 0.5, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0.5, 0.5, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0.5, 0.5, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0.5, 0.5, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0.5, 0.5, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0.5, 0.5, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0.5, 0.5, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0.5, 0.5,
                                    0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0.5),
                           nrow = 10,
                           byrow = T,
                           dimnames = list(paste0("z", states_val),
                                           paste0("z", states_val)))
knitr::kable(transition_matrix, caption = "Transition matrix", row.names = TRUE)
```

Table 1: Transition matrix

	z1	z2	z3	z4	z5	z6	z7	z8	z9	z10
z1	0.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
z2	0.0	0.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
z3	0.0	0.0	0.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0
z4	0.0	0.0	0.0	0.5	0.5	0.0	0.0	0.0	0.0	0.0
z5	0.0	0.0	0.0	0.0	0.5	0.5	0.0	0.0	0.0	0.0
z6	0.0	0.0	0.0	0.0	0.0	0.5	0.5	0.0	0.0	0.0
z7	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.5	0.0	0.0
z8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.5	0.0
z9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.5
z10	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5

```
# Initializing emission matrix.
emission_matrix = matrix(data = c(0.2, 0.2, 0.2, 0, 0, 0, 0, 0, 0.2, 0.2,
                                  0.2, 0.2, 0.2, 0.2, 0, 0, 0, 0, 0, 0.2,
                                  0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0, 0, 0,
                                  0, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0, 0,
                                  0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0,
                                  0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0,
                                  0, 0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0,
                                  0, 0, 0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2,
                                  0.2, 0, 0, 0, 0, 0, 0.2, 0.2, 0.2, 0.2,
                                  0.2, 0.2, 0, 0, 0, 0, 0, 0.2, 0.2, 0.2),
                         nrow = 10,
                         byrow = T,
                         dimnames = list(paste0("z", states_val),
                                         paste0("x", states_val)))
knitr::kable(emission_matrix, caption = "Emission matrix", row.names = TRUE)
```

Table 2: Emission matrix

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10
z1	0.2	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.2	0.2
z2	0.2	0.2	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.2
z3	0.2	0.2	0.2	0.2	0.2	0.0	0.0	0.0	0.0	0.0
z4	0.0	0.2	0.2	0.2	0.2	0.2	0.0	0.0	0.0	0.0
z5	0.0	0.0	0.2	0.2	0.2	0.2	0.2	0.0	0.0	0.0
z6	0.0	0.0	0.0	0.2	0.2	0.2	0.2	0.2	0.0	0.0
z7	0.0	0.0	0.0	0.0	0.2	0.2	0.2	0.2	0.2	0.0
z8	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.2	0.2	0.2
z9	0.2	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.2	0.2
z10	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.2

The transition matrix tells you the probability of transitioning from its row's state to its column's state. This means that the column represents the target state and the row represents the start state.

Using these implemented information, we can finally inizialize the HMM.

```
# Initializing HMM.
HMM = initHMM(States = states_val,
              Symbols = emissions_val,
              startProb = states_startProbs,
              transProbs = transition_matrix,
              emissionProbs = emission_matrix)
```

Assignment 2. Simulating the HMM.

Simulate the HMM for 100 time steps.

```
hmm_sim = simHMM(hmm = HMM, length = 100)
```

Assignment 3. Computing filtered/smoothed probability distributions and most probable path.

Discard the hidden states from the sample obtained above. Use the remaining observations to compute the filtered and smoothed probability distributions for each of the 100 time points. Compute also the most probable path.

The *forward-backward algorithm* can be used to calculate the probability of all possible states at time t. The algorithm returns for each time t and every state a specific α- and $\beta-$value which will be used afterwards to compute the filtered and smoothed probability distributons. Even if it finds the most likely state for any point in time, it cannot be used to find the most likely sequence of states. This is rather done by the *Viterbi algorithm*.

In this assignment, we will work on both cases: First, we will compute the filtered and smoothed probability distributions for each of the 100 time points using the forward-backward algorithm. Furthermore, we will compute the most probable path with help of the Viterbi algorithm.

For practice, we implemented the algorithms manually. Since the actual goal was to work with the `HMM`-package, we will first work on the assignemnt with the own implementation and then redo it with the `HMM`-package.

3.1 Manual implementation of relevant functions.

$\alpha(Z^t)$

```
manual_get_alphas = function(states, emission_matrix, transition_matrix, obs, alpha_prev) {

  # Initializing alphas.
  alphas = c()

  # Computing alpha for every possible state.
  for (state in states) {

    # Extracting probability of observed variable at time t given state at time t.
    prob_obs_t = emission_matrix[state, obs]

    # Summing up probability of observed variable at time t-1 given state z at time t-1
    # over all states at time t-1 (alpha of each state z at time t-1)
    # * transition of each state z to state.
    prob_obs_t_minus_1 = 0
    for (z in states) {
      prob_obs_t_minus_1 = prob_obs_t_minus_1 + alpha_prev[z] * transition_matrix[z, state]
    }

    # Storing product for state in alpha.
    alphas = c(alphas, prob_obs_t * prob_obs_t_minus_1)
  }
  # Returning alphas.
  return(alphas)
}
```

$\beta(Z^t)$

```
manual_get_betas = function(states, emission_matrix, transition_matrix, obs_next, beta_next) {

  # Initializing beta.
  betas = c()

  # Computing beta for every possible state.
  for (state in states) {
    betas[state] <- sum(
      sapply(states, function(z) {
        beta_next[z] * emission_matrix[z, obs_next] * transition_matrix[state, z]
      })
    )
  }

  # Returning betas.
  return(betas)
}
```

Forward-Backward Algorithm

```
manual_fb_algorithm = function(states, states_startProbs, transition_matrix,
                               emission_matrix, obs) {

  # Initializing t, alphas and betas.
  t_max = length(obs)
  alphas = matrix(NA, nrow = t_max, ncol = length(states))
  colnames(alphas) = paste0("z", states_val)
  rownames(alphas) = paste0("t", c(1:t_max))
  betas = matrix(NA, nrow = t_max, ncol = length(states))
  colnames(betas) = paste0("z", states)
  rownames(betas) = paste0("t", c(1:t_max))

  # Performing forward-part.
    # Initializing alpha for t = 0 (or 1 respectively).
    alphas[1, ] = emission_matrix[, obs[1]] * states_startProbs
    # Computing alpha for all other other t.
    for (t in 2:t_max) {
      alphas[t, ] = manual_get_alphas(states = states,
                                      emission_matrix = emission_matrix,
                                      transition_matrix = transition_matrix,
                                      obs = obs[t],
                                      alpha_prev = alphas[t-1, ])
    }

  # Performing backward-part.
    # Initializing beta for t = t_max.
    betas[t_max, ] = 1
    # Computing beta for all oher t.
    for (t in (t_max-1):1) {
      betas[t, ] = manual_get_betas(states = states,
                                    emission_matrix = emission_matrix,
                                    transition_matrix = transition_matrix,
                                    obs_next = obs[t+1],
```

```
                                    beta_next = betas[t+1, ])
    }

  # Returning results.
  return(list(alphas = alphas, betas = betas))
}
```

Filtering

```
manual_filtering = function(alphas) {

  # Initializing filering results.
  results = matrix(NA, nrow = nrow(alphas), ncol = ncol(alphas))

  # Filering for every time t.
  for (t in 1:nrow(alphas)) {
    results[t, ] = alphas[t, ] / sum(alphas[t, ])
  }

  # Returning results.
  return(results)
}
```

Smoothing

```
manual_smoothing = function(alphas, betas) {

  # Initializing smoothing results.
  results = matrix(NA, nrow = nrow(alphas), ncol = ncol(alphas))

  # Filering for every time t.
  for (t in 1:nrow(alphas)) {
    results[t, ] = (alphas[t, ] * betas[t, ]) / (sum(alphas[t, ] * betas[t, ]))
  }

  # Returning results.
  return(results)
}
```

Viterbi algorithm

```
# Not done manually yet.
```

3.2 Performing filtering, smoothing. Computing most probable path.

Usage of manual implementations.

```
# Computing alphas and betas.
manual_alphas_betas = manual_fb_algorithm(states = states_val,
                                          states_startProbs = states_startProbs,
                                          transition_matrix = transition_matrix,
                                          emission_matrix = emission_matrix,
                                          obs = hmm_sim$observation)

# Computing normalized filtered probability distributions.
```

```r
manual_filtering_results = manual_filtering(manual_alphas_betas$alphas)

# Computing normalized smoothed probability distributions.
manual_smoothing_results = manual_smoothing(manual_alphas_betas$alphas,
                                            manual_alphas_betas$betas)

# Computing most probable path.
# ...
```

Usage of `HMM` package.

Usage of the `forward()` or `backward`-function leads to probabilities that are given on a logarithmic scale. The requires to wrap the `exp()`-function around it. Furthermore, the probabilties are not normalized yet. For every time step t, the probabilities of all states must sum up to 1. To achieve that, we will use an own-implemented function `normalize()`.

```r
# Implementing function for normalization.
normalize = function(c) {
  return(c/sum(c))
}

# Computing filtered probability distributions.
  # Computing forward probabilities.
  forwards_unnormalized = exp(forward(hmm = HMM,
                                      observation = hmm_sim$observation))
  # Normalizing.
  forwards_normalized = apply(X = forwards_unnormalized,
                              MARGIN = 2,
                              FUN = normalize)

  # Renaming and labeling.
  HMM_filtering_results = forwards_normalized
  rownames(HMM_filtering_results) = paste0("z", states_val)
  colnames(HMM_filtering_results) = paste0("t", c(1:length(hmm_sim$observation)))

# Computing smoothed probability distributions.
  # Computing backward probabilities.
  backwards_unnormalized = exp(backward(hmm = HMM,
                                        observation = hmm_sim$observation))
  # Multiplying backward and forward probabilities.
  forwards_backwards_unnormalized = forwards_unnormalized * backwards_unnormalized
  # Normalizing.
  forwards_backwards_normalized = apply(X = forwards_backwards_unnormalized,
                                        MARGIN = 2,
                                        FUN = normalize)
  # Renaming and labeling.
  HMM_smoothing_results = forwards_backwards_normalized
  rownames(HMM_smoothing_results) = paste0("z", states_val)
  colnames(HMM_smoothing_results) = paste0("t", c(1:length(hmm_sim$observation)))

# Computing most probable path.
HMM_viterbi_results = viterbi(hmm = HMM,
                              observation = hmm_sim$observation)
```

Assignment 4. Computing accuracies. Plotting results.

4.1 Computing accuracies.

Compute the accuracy of the filtered and smoothed probability distributions, and of the most probable path. That is, compute the percentage of the true hidden states that are guessed by each method.

Implementing functions to obtain accuracies.

Again, we will compare the accuracies between the different methods (filtering, smoothing, most probable path) for both the manual implementation and also the `HMM`-version.

To do so, for each time point t, we will compare the true state of the simulation to the state with the highest probability according to the probability matrices returned by the algorithms from *assignment 3*. Since this implies that we first need to identify the state with the highest probability for each t, we will implement the function `get_most_probable_states()`. Using the most probable states obtained by the algorithms, we will compare them to the true simulated states to obtain the accuracy by the own-implemented function `get_accuracy()`.

```
# Implementing function to obtain most probable state for each t.
get_most_probable_states = function(matrix_probs_normalized, MARGIN) {
  most_probable_states = apply(X = matrix_probs_normalized,
                               MARGIN = MARGIN,
                               FUN = which.max)
  return(most_probable_states)
}

# Implementing function to obtain accuracy.
get_accuracy = function(most_probable_states, true_states) {
  # Computing confusion matrix.
  confusion_matrix = table(most_probable_states, true_states)
  # Computing accuracy.
  accuracy = sum(diag(confusion_matrix))/sum(confusion_matrix)
  # Returning accuracy.
  return (accuracy)
}
```

Computing accuracies of manual implementations.

```
# Obtaining most probable state (mps) for each t.
  # Filtering.
  manual_filtering_mps = get_most_probable_states(manual_filtering_results, 1)
  # Smoothing.
  manual_smoothing_mps = get_most_probable_states(manual_smoothing_results, 1)
  # Viterbi.
  # ...

# Obtaining accuracy.
  # Filtering.
  manual_filtering_accuracy = get_accuracy(manual_filtering_mps, hmm_sim$states)
  # Smoothing.
  manual_smoothing_accuracy = get_accuracy(manual_smoothing_mps, hmm_sim$states)
  # Viterbi.
  # ...
```

Computing accuracies of `HMM`-implementations.

```
# Obtaining most probable state (mps) for each t.
  # Filtering.
  HMM_filtering_mps = get_most_probable_states(HMM_filtering_results, 2)
  # Smoothing.
  HMM_smoothing_mps = get_most_probable_states(HMM_smoothing_results, 2)

# Obtaining accuracy.
  # Filtering.
  HMM_filtering_accuracy = get_accuracy(HMM_filtering_mps, hmm_sim$states)
  # Smoothing.
  HMM_smoothing_accuracy = get_accuracy(HMM_smoothing_mps, hmm_sim$states)
  # Viterbi.
  HMM_viterby_accuracy = get_accuracy(HMM_viterbi_results, hmm_sim$states)
```

Comparison.

```
knitr::kable(data.frame(Method = c("Manual filtering", "Manual smoothing",
                                   "HMM filtering", "HMM smoothing", "HMM viterbi"),
                        Accuracy = c(manual_filtering_accuracy, manual_smoothing_accuracy,
                                     HMM_filtering_accuracy, HMM_smoothing_accuracy,
                                     HMM_viterby_accuracy)),
             caption = "Comparison of the accuracies of the applied methods")
```

Table 3: Comparison of the accuracies of the applied methods

Method	Accuracy
Manual filtering	0.56
Manual smoothing	0.68
HMM filtering	0.56
HMM smoothing	0.68
HMM viterbi	0.43

4.2 Visual analysis of the accuracies.

Even if it is not specifically asked, we will implement two plots (one plot for each implementation) to analyze visually how the algorithms have performed. We will do it by comparing the most probable states guessed by the methods to the true states of the simulation.

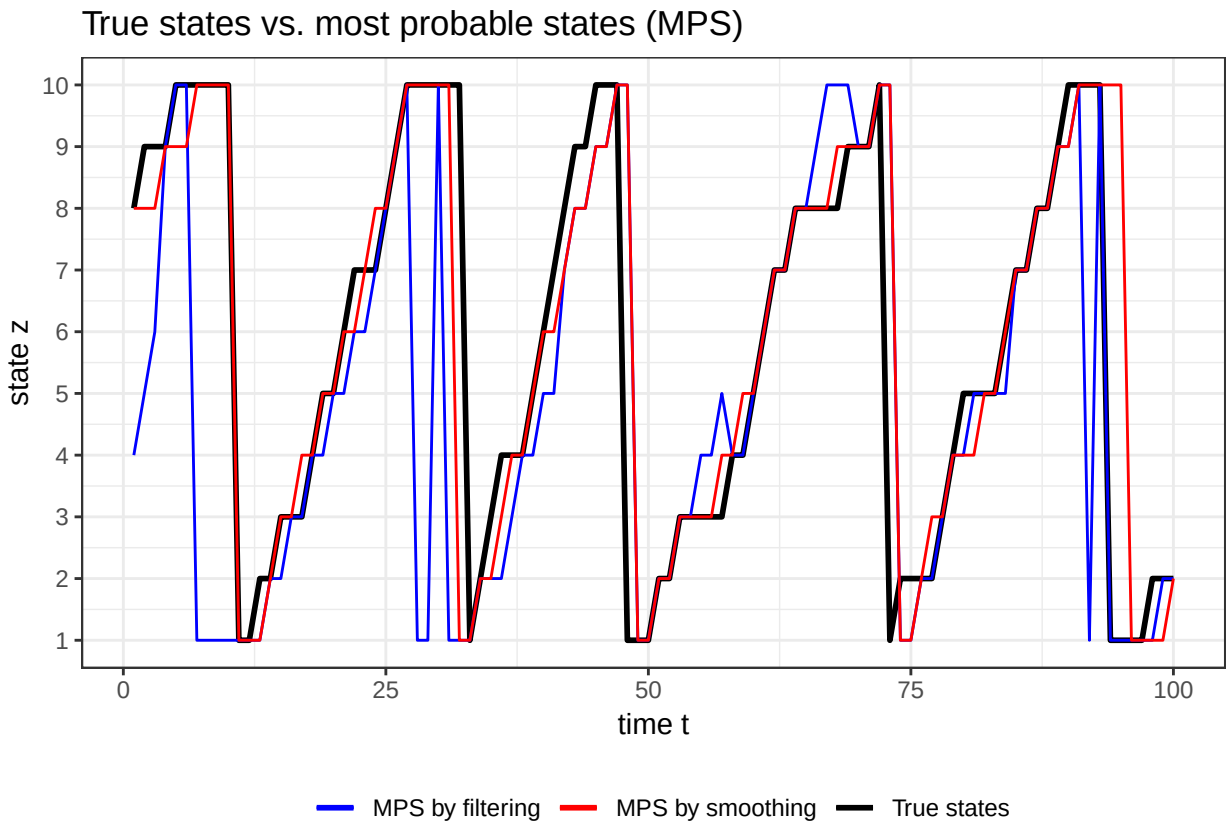
Implementing function to create plot.

```
library(ggplot2)

analyze_accuracy_visually = function(sim, filtering_mps, smoothing_mps, viterbi = NA) {

  p = ggplot() +

  # Adding true states of simulation.
  geom_line(aes(x = 1:length(sim$observation),
                y = sim$states,
                color = "True states"),
            size = 1) +

  # Adding most probable state by filtering.
```

```
  geom_line(aes(x = 1:length(sim$states),
                y = filtering_mps,
                color = "MPS by filtering")) +

  # Adding most probable state by smoothing.
  geom_line(aes(x = 1:length(sim$states),
                y = smoothing_mps,
                color = "MPS by smoothing")) +

  # Setting up plot layout.
  scale_y_continuous(breaks = c(1:length(sim$states))) +
  theme_bw() +
  labs(title = "True states vs. most probable states (MPS)",
       x = "time t",
       y = "state z",
       colour = NULL) +
  theme(legend.position="bottom")

  # Optionally adding most probable state by viterbi.
  if (!is.na(viterbi)) {
    p = p +
      geom_line(aes(x = 1:length(sim$states),
                y = viterbi,
                color = "MPS by viterbi")) +
      scale_color_manual(values = c("True states" = "black",
                                    "MPS by filtering" = "blue",
                                    "MPS by smoothing" = "red",
                                    "MPS by viterbi" = "grey"))
    plot(p)
  } else {
    p = p +
      scale_color_manual(values = c("True states" = "black",
                                    "MPS by filtering" = "blue",
                                    "MPS by smoothing" = "red"))
    plot(p)
  }
}
```

Plotting results for manual implementation.

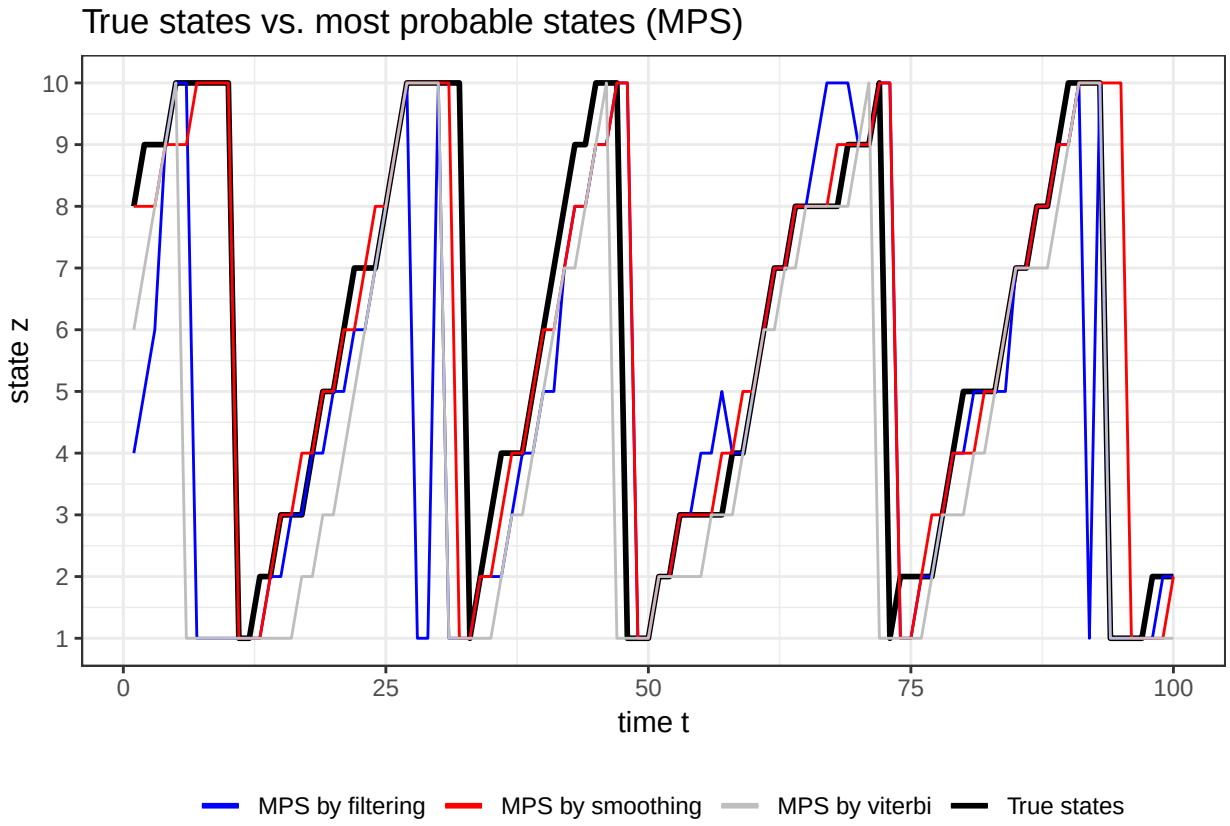
```
analyze_accuracy_visually(sim = hmm_sim,
                          filtering_mps = manual_filtering_mps,
                          smoothing_mps = manual_smoothing_mps)
```



Plotting results for `HMM` implementation.

```
analyze_accuracy_visually(sim = hmm_sim,
                          filtering_mps = HMM_filtering_mps,
                          smoothing_mps = HMM_smoothing_mps,
                          viterbi = HMM_viterbi_results)
```

```
Warning in if (!is.na(viterbi)) {: Bedingung hat Länge > 1 und nur das
erste Element wird benutzt
```


True states vs. most probable states (MPS)
state z
time t
0
25
50
75
100
1
2
3
4
5
6
7
8
9
10
MPS by filtering
MPS by smoothing
MPS by viterbi
True states

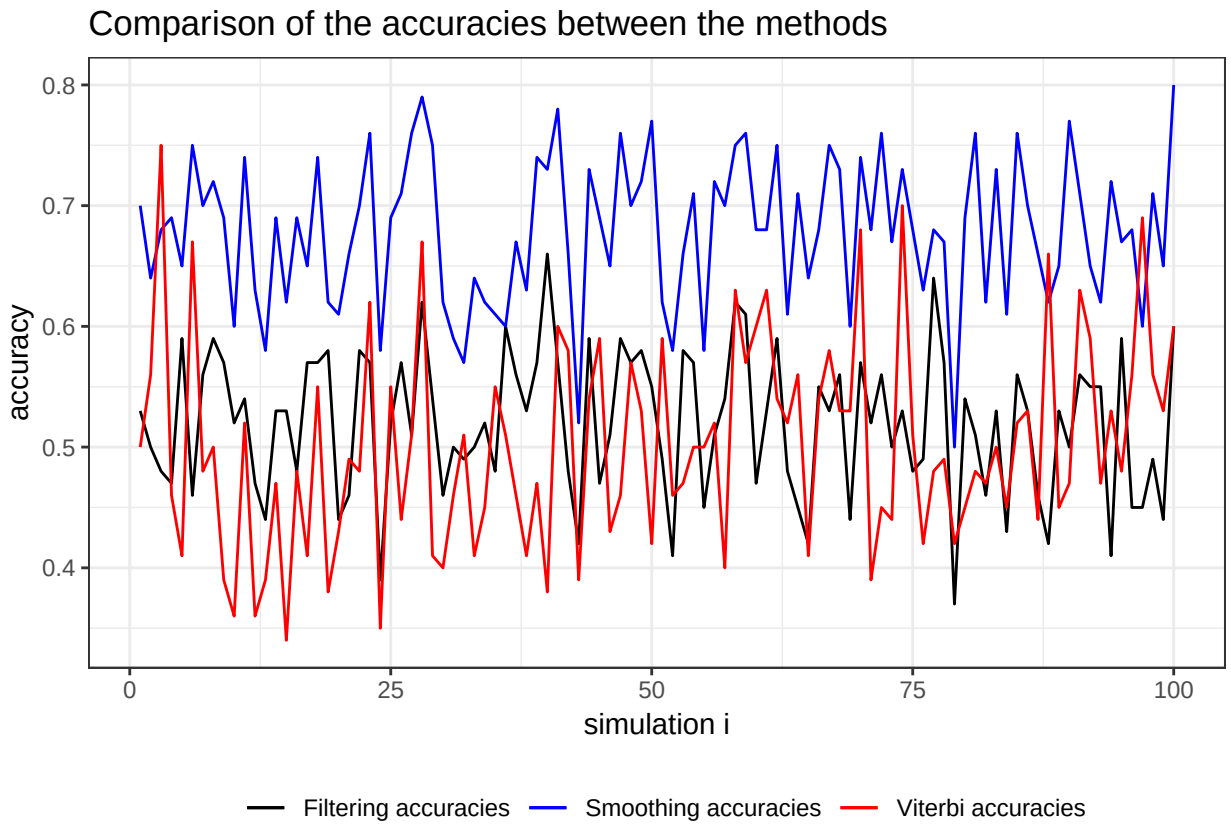
Assignment 5. Repeating process with different simulations.

Repeat the previous exercise with different simulated samples. In general, the smoothed distributions should be more accurate than the filtered distributions. Why ? In general, the smoothed distributions should be more accurate than the most probable paths, too. Why ?

Since we have showed in assignment 3 already how to implement the process manually, we will only use the given functions from the HMM-package within this assignment. We will generate one hundret simulations and repeat the whole process for each simulation. Within every process, the accuracy for filtering, smoothing and the viterby algorithm are obtained and stored. As a result, these accuracies can be compared.

```
# Creating object to store accuarcies.
filtering_accuracies = c()
smoothing_accuracies = c()
viterbi_accuracies = c()

# Computing accuracies for 100 differenct simulations.
for (i in 1:100) {

  # Simulating.
  hmm_sim = simHMM(hmm = HMM, length = 100)

  # Computing filtered probability distributions.
    # Computing forward probabilities.
    forwards_unnormalized = exp(forward(hmm = HMM,
                                        observation = hmm_sim$observation))
    # Normalizing.
    forwards_normalized = apply(X = forwards_unnormalized,
                                MARGIN = 2,
                                FUN = normalize)
    # Renaming.
    HMM_filtering_results = forwards_normalized


  # Computing smoothed probability distributions.
    # Computing backward probabilities.
    backwards_unnormalized = exp(backward(hmm = HMM,
                                          observation = hmm_sim$observation))
    # Multiplying backward and forward probabilities.
    forwards_backwards_unnormalized = forwards_unnormalized * backwards_unnormalized
    # Normalizing.
    forwards_backwards_normalized = apply(X = forwards_backwards_unnormalized,
                                          MARGIN = 2,
                                          FUN = normalize)
    # Renaming.
    HMM_smoothing_results = forwards_backwards_normalized

  # Computing most probable path.
    HMM_viterbi_results = viterbi(hmm = HMM,
                                  observation = hmm_sim$observation)

  # Obtaining most probable state (mps) for each t.
    # Filtering.
    HMM_filtering_mps = get_most_probable_states(HMM_filtering_results, 2)
    # Smoothing.
```

```
    HMM_smoothing_mps = get_most_probable_states(HMM_smoothing_results, 2)

  # Obtaining and storing accuracies.
    # Filtering.
    filtering_accuracies[i] = get_accuracy(HMM_filtering_mps, hmm_sim$states)
    # Smoothing.
    smoothing_accuracies[i] = get_accuracy(HMM_smoothing_mps, hmm_sim$states)
    # Viterbi.
    viterbi_accuracies[i] = get_accuracy(HMM_viterbi_results, hmm_sim$states)
}

# Plotting results.
ggplot() +
  # Adding accuracies by filtering.
  geom_line(aes(x = 1:i,
                y = filtering_accuracies,
                color = "Filtering accuracies")) +
  # Adding accuracies by smoothing.
  geom_line(aes(x = 1:i,
                y = smoothing_accuracies,
                color = "Smoothing accuracies")) +
  # Adding accuracies by viterbi.
  geom_line(aes(x = 1:i,
                y = viterbi_accuracies,
                color = "Viterbi accuracies")) +
  # Setting up plot layout.
  scale_color_manual(values = c("Filtering accuracies" = "black",
                                "Smoothing accuracies" = "blue",
                                "Viterbi accuracies" = "red")) +
  theme_bw() +
  labs(title = "Comparison of the accuracies between the methods",
       x = "simulation i",
       y = "accuracy",
       colour = NULL) +
  theme(legend.position="bottom")
```

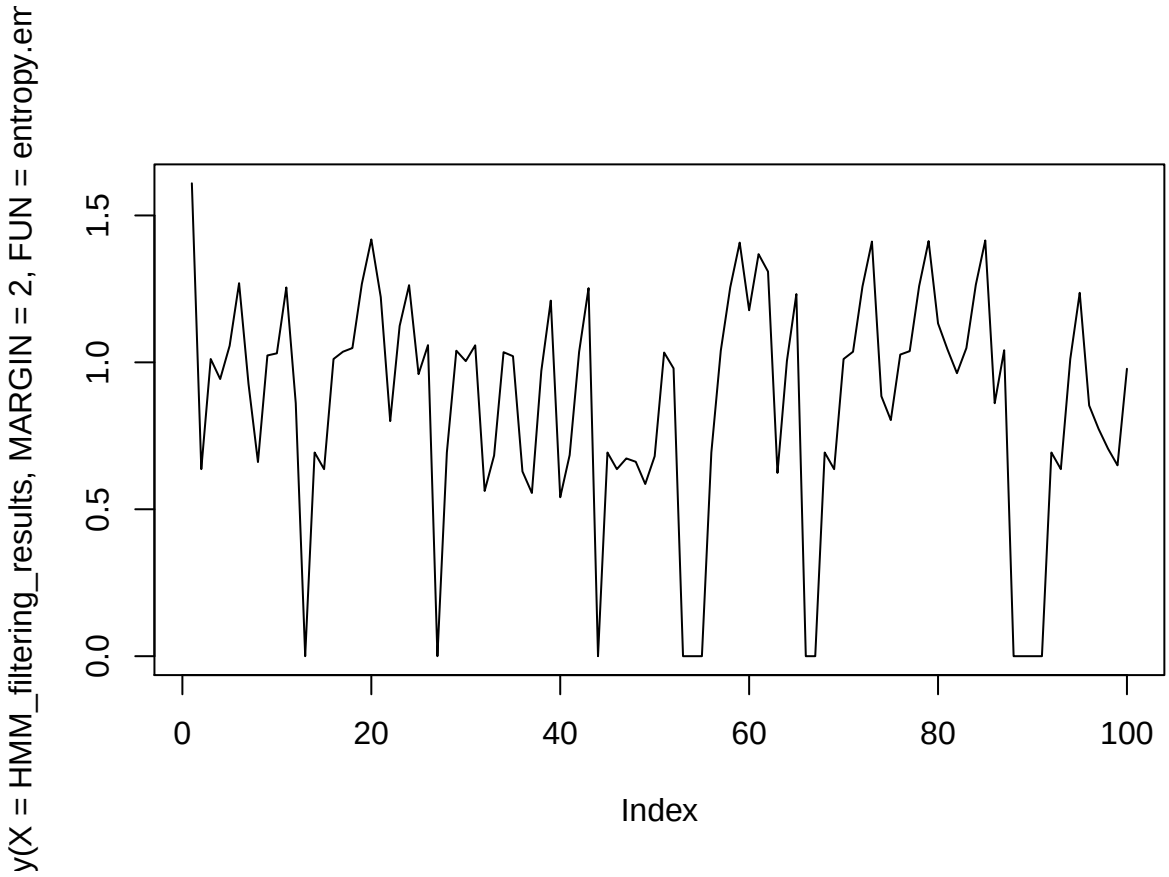


Generally, the smoothed probabilites show higher accuracy. Because, when looking at the formulas onecan see that the smoothed probabilites include observations 0:T, whilst filtered only uses 0:t. Meaning, thesmoothed probabilites have more information to base predictions on. Most probable path (Viterbi) actuallygenerally shows the lowest accuracy, we argue this could be due to the fact that this algorithm takes intothe constraint that it has to provide a feasible path. But for instance, filtered or smoothed could come upwith "jumps" in the path that are not possible, e.g. from state 1 at time=1 to state 5 at time =2.

Assignment 6. Analysing entropy.

Is it true that the more observations you have the better you know where the robot is ?
Hint: You may want to compute the entropy of the filtered distributions with the function entropy.empirical of the package entropy.

```
library(entropy)
plot(apply(X = HMM_filtering_results,
           MARGIN = 2,
           FUN = entropy.empirical),
     type = "l")
```



No, looking at the entropy plot, we can see that there are many fluctuations in the level of entropy (certaintyin the predictions) across all time steps. This means that the algorithm e.g. at step 5 can be as uncertainas at step 95, whilst step 95 has more information to work with. But this increase in information does notlead to an increase in prediction certainty

Assignment 7. Predicting hidden states for next time step.

Consider any of the samples above of length 100. Compute the probabilities of the hidden states for the time step 101.

We achieve the "forecast" by usage of the obtained matrices containing either the filtered or smoothed probability distributions for each of the 100 time points. Since we do not have any observations for time point 101, using the transition probabilites is our only possibiliy. Multiplying the state probabilities of the last time point (time point 100) by the transition matrix leads to the new state probabilites for time point 101.

```
# Computing probabilities for hidden states for next time step 101.
knitr::kable(transition_matrix %*% as.matrix(HMM_filtering_results[, 100]),
             caption = "Probabilities of the hidden states for the time step 101")
```

Table 4: Probabilities of the hidden states for the time step 101

z1	0.0000000
z2	0.1901408
z3	0.4366197
z4	0.3098592
z5	0.0633803
z6	0.0000000
z7	0.0000000
z8	0.0000000
z9	0.0000000
z10	0.0000000

```
knitr::kable(transition_matrix %*% as.matrix(HMM_smoothing_results[, 100]),
             caption = "Probabilities of the hidden states for the time step 101")
```

Table 5: Probabilities of the hidden states for the time step 101

z1	0.0000000
z2	0.1901408
z3	0.4366197
z4	0.3098592
z5	0.0633803
z6	0.0000000
z7	0.0000000
z8	0.0000000
z9	0.0000000
z10	0.0000000

```
# transition_matrix %*% manual_filtering_results[100, ]
```

Using the filtered probabilites we estimate where the robot would be at time = 100. From the estimatedprobabilites we want to predict the next step. To do so, we use the transition matrix. As expected usingeither the filtered or smoothed probabilites will return the same predictions because at time = 100, t=T.