Advanced Machine Learning - Lab 02

Maximilian Pfundstein (maxpf364)

2019-10-18

Contents

1 Building a Hidden Markov Model **1**

2 Simulate 100 Time Steps **2**

3 Discarding Hidden States **2**

4 Filtered and Smoothed Probability Distributions and Most Probable Path **6**

5 Accuracy of the Filtered and Smoothed Probability Distributions **7**

6 Is Having more Observations Always Better? **9**

7 Probabilities of Time Step 101 **9**

8 Source Code **10**

The purpose of the lab is to put in practice some of the concepts covered in the lectures. To do so, 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 with equal $[i-2, i+2]$ probability.

1 Building a Hidden Markov Model

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

```
N = 10

# Defining States Z1, Z2, ..., ZN
states = paste(rep("Z", N), 1:N, sep = "")

# Defining Symbols S1, S2, ..., SN
symbols = paste(rep("S", N), 1:N, sep = "")

# Starting Probabilities
startProbs = rep(1/N, N)

# Transition Probabilities
transProbs = matrix(0, ncol = N, nrow = N)
# Staying in the current state with 0.5 probability is just the diagonal
diag(transProbs) = 0.5
# Moving to the next is also 0.5
diag(transProbs[,-1]) = 0.5
transProbs[10, 1] = 0.5
```

```r
# Emission Probabilities
emissionProbs = matrix(0, ncol = N, nrow = N)

# 0.2 For i-2 to i+2
for (i in 1:N) {
  for (j in c(3:-1)) {
    emissionProbs[((i-j)%%N)+1,i] = 0.2
  }
}

robot_hmm = initHMM(States = states,
                    Symbols = symbols,
                    startProbs = startProbs,
                    transProbs = transProbs,
                    emissionProbs = emissionProbs)
```

2 Simulate 100 Time Steps

Task: Simulate the HMM for 100 time steps.

```r
nSim = 100

simulatedStates = simHMM(robot_hmm, nSim)
```

3 Discarding Hidden States

Task: 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.

```r
# Custom implementations of the forward and backward algorithm

custom_forward = function(hmm, observations) {

  Z = matrix(NA, ncol=length(hmm$States), nrow=length(observations))

  Z[1,] = hmm$emissionProbs[, observations[1]] * hmm$startProbs

  for (t in 2:length(observations)) {
    Z[t, ] = hmm$emissionProbs[, observations[t]] * (Z[t-1,] %*% hmm$transProbs)
  }

  return(t(Z))
}

custom_backward = function(hmm, observations) {

  Z = matrix(NA, ncol=length(hmm$States), nrow=length(observations))

  Z[length(observations),] = 1
```

```
  for (t in ((length(observations)-1):0)) {
    for (state in 1:length(hmm$States)) {
     Z[t, state] = sum(Z[t+1,] * hmm$emissionProbs[,observations[t+1]] * transProbs[state,])
    }
  }

  return(t(Z))
}
```

```
# The library returns the probabilities logged, so we have to de-log
alpha = exp(forward(robot_hmm, simulatedStates$observation))
beta = exp(backward(robot_hmm, simulatedStates$observation))

# Filtered
# Normalizing, for each column x = x/sum(x), done with prop.table
filtered = prop.table(alpha, 2)

# Smoothed
# Can either be done manually or using the function posterior (== smoothed) in
# this package
# AUTOMATIC
smoothed_automatically = posterior(robot_hmm, simulatedStates$observation)
# MANUALLY (Instead of division prop.table would work as well)
smoothed_manually = alpha * beta / colSums(alpha * beta)

# Path
hmm_viterbi = viterbi(robot_hmm, simulatedStates$observation)

# Print Filtered
filtered[,(nSim-5):nSim]
```

```
##       index
## states        95        96         97         98         99       100
##    Z1  0.0000000 0.0000000 0.00000000 0.00000000 0.00000000 0.0000000
##    Z2  0.0000000 0.0000000 0.00000000 0.00000000 0.00000000 0.0000000
##    Z3  0.0000000 0.0000000 0.00000000 0.00000000 0.00000000 0.0000000
##    Z4  0.0000000 0.0000000 0.00000000 0.00000000 0.00000000 0.0000000
##    Z5  0.0000000 0.0000000 0.00000000 0.00000000 0.00000000 0.0000000
##    Z6  0.0500000 0.0250000 0.00000000 0.00000000 0.00000000 0.0000000
##    Z7  0.2785714 0.1642857 0.09584087 0.05237154 0.05430328 0.0000000
##    Z8  0.6714286 0.4750000 0.32368897 0.22924901 0.29200820 0.1779884
##    Z9  0.0000000 0.3357143 0.41048825 0.40118577 0.65368852 0.4860453
##    Z10 0.0000000 0.0000000 0.16998192 0.31719368 0.00000000 0.3359663
```

```
# Print Smoothed
smoothed_automatically[,(nSim-5):nSim]
```

```
##       index
## states         95         96        97         98         99       100
##    Z1  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z2  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z3  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z4  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z5  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z6  0.07372301 0.02580305 0.0000000 0.00000000 0.00000000 0.0000000
```

```
##    Z7  0.43127962 0.31490258 0.1953660 0.08372828 0.02790943 0.0000000
##    Z8  0.49499737 0.56029489 0.5655608 0.48867825 0.30015798 0.1779884
##    Z9  0.00000000 0.09899947 0.2390732 0.42759347 0.67193260 0.4860453
##    Z10 0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.3359663
```

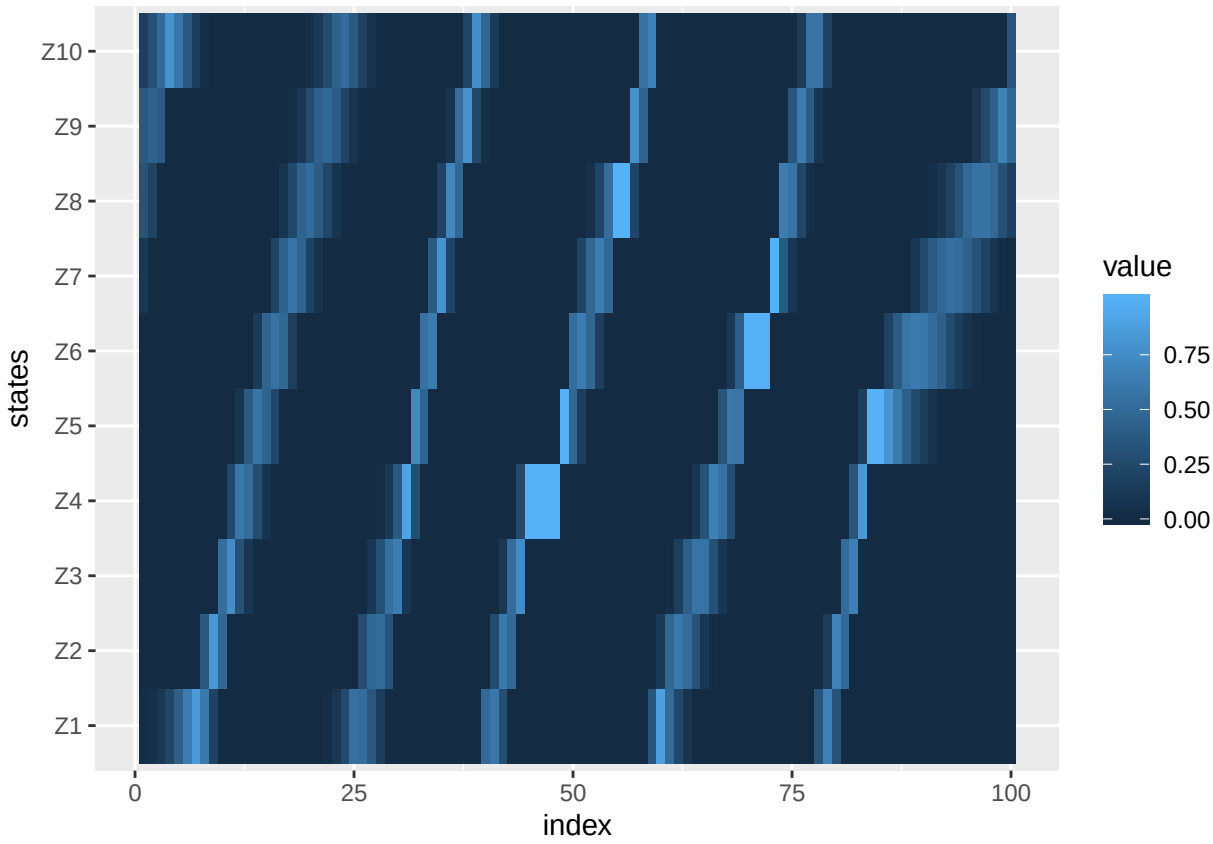
```
smoothed_manually[,(nSim-5):nSim]
```

```
##       index
## states         95         96        97         98         99       100
##    Z1  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z2  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z3  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z4  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z5  0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.0000000
##    Z6  0.07372301 0.02580305 0.0000000 0.00000000 0.00000000 0.0000000
##    Z7  0.43127962 0.31490258 0.1953660 0.08372828 0.02790943 0.0000000
##    Z8  0.49499737 0.56029489 0.5655608 0.48867825 0.30015798 0.1779884
##    Z9  0.00000000 0.09899947 0.2390732 0.42759347 0.67193260 0.4860453
##    Z10 0.00000000 0.00000000 0.0000000 0.00000000 0.00000000 0.3359663
```

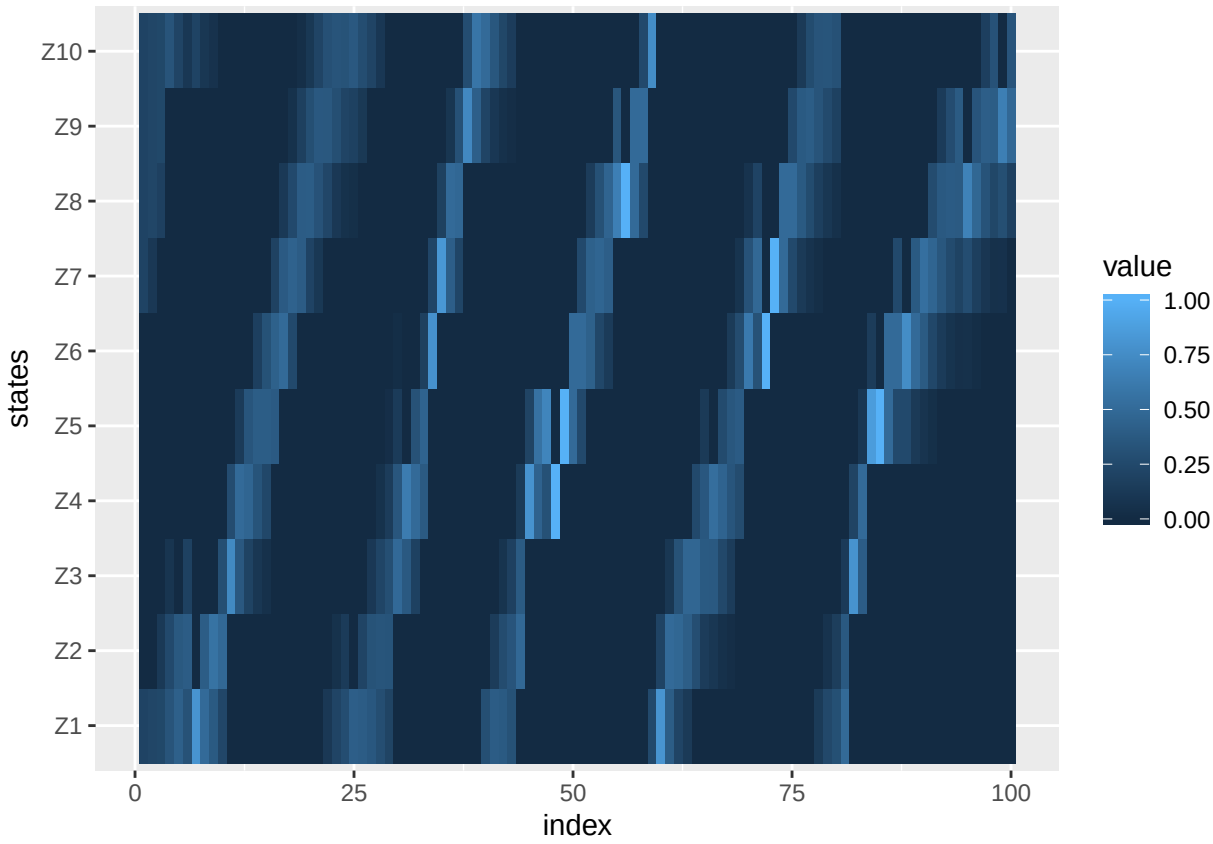
```
# Print Viterbi
hmm_viterbi
```

```
##   [1] "Z1"  "Z1"  "Z1"  "Z1"  "Z1"  "Z1"  "Z1"  "Z1"  "Z1"  "Z2"  "Z3"
##  [12] "Z3"  "Z3"  "Z3"  "Z4"  "Z5"  "Z6"  "Z7"  "Z8"  "Z9"  "Z10" "Z1"
##  [23] "Z1"  "Z1"  "Z1"  "Z1"  "Z1"  "Z1"  "Z2"  "Z3"  "Z3"  "Z4"  "Z5"
##  [34] "Z6"  "Z7"  "Z7"  "Z8"  "Z9"  "Z10" "Z1"  "Z1"  "Z1"  "Z2"  "Z3"
##  [45] "Z4"  "Z4"  "Z4"  "Z4"  "Z5"  "Z5"  "Z5"  "Z6"  "Z6"  "Z7"  "Z8"
##  [56] "Z8"  "Z9"  "Z10" "Z1"  "Z1"  "Z1"  "Z1"  "Z1"  "Z2"  "Z2"  "Z2"
##  [67] "Z3"  "Z4"  "Z5"  "Z6"  "Z6"  "Z6"  "Z7"  "Z7"  "Z8"  "Z9"  "Z10"
##  [78] "Z1"  "Z1"  "Z1"  "Z2"  "Z3"  "Z4"  "Z5"  "Z5"  "Z5"  "Z5"  "Z5"
##  [89] "Z5"  "Z5"  "Z5"  "Z6"  "Z6"  "Z6"  "Z6"  "Z6"  "Z7"  "Z7"  "Z7"
## [100] "Z8"
```

The visualization for the smoothed path (distributions) looks like this:



The visualization for the filtered path (distributions) looks like this:



4 Filtered and Smoothed Probability Distributions and Most Probable Path

Task: 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.

Hint: Note that the function `forward`in the `HMM` package returns probabilities in log scale. You may need to use the functions `exp` and `prop.table` in order to obtain a normalized probability distribution. You may also want to use the functions `apply` and `which.max` to find out the most probable states. Finally, recall that you can compare two vectors `A` and `B` elementwise as `A==B`, and that the function `table` will count the number of times that the different elements in a vector occur in the vector.

```
# Taking the max for each observation
# Smoothed
filtered_max = max.col(t(filtered), "first")
# Filtered
smoothed_max = max.col(t(smoothed_automatically), "first")

# ACC
# Smoothed
confusionMatrix_smoothed = table(simulatedStates$states,
                         paste(rep("Z", nSim), smoothed_max, sep=""))
acc_smoothed = sum(diag(confusionMatrix_smoothed))/sum(confusionMatrix_smoothed)

# Filtered
confusionMatrix_filtered = table(simulatedStates$states,
```

```
                        paste(rep("Z", nSim), filtered_max, sep=""))
acc_filtered = sum(diag(confusionMatrix_filtered))/sum(confusionMatrix_filtered)

# Viterbi
confusionMatrix_viterbi = table(simulatedStates$states, hmm_viterbi)
acc_viterbi = sum(diag(confusionMatrix_viterbi))/sum(confusionMatrix_viterbi)

acc_smoothed
```

```
## [1] 0.63
```

```
acc_filtered
```

```
## [1] 0.41
```

```
acc_viterbi
```

```
## [1] 0.49
```

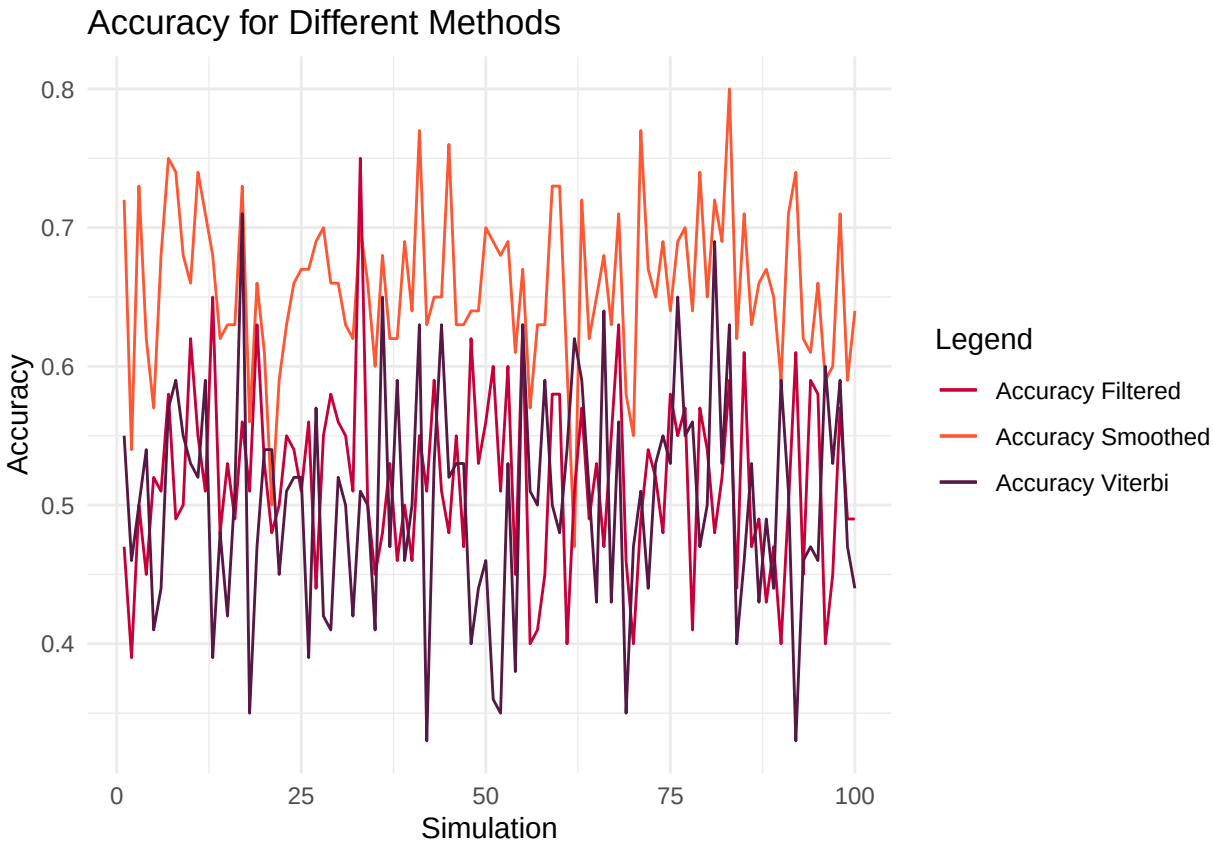
5 Accuracy of the Filtered and Smoothed Probability Distributions

Task: 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?

Answer: We will built a function that handles all of the above for us.

```
simulate_hmm = function(hmm, nSim = 100) {

  # Simulate
  simulatedStates = simHMM(robot_hmm, nSim)

  # Filtered
  alpha = exp(forward(hmm, simulatedStates$observation))
  filtered = prop.table(alpha, 2)
  filtered_max = max.col(t(filtered))

  # Smoothed
  smoothed = posterior(hmm, simulatedStates$observation)
  smoothed_max = max.col(t(smoothed))

  # Viterbi
  hmm_viterbi = viterbi(hmm, simulatedStates$observation)

  # Accuracies
  confusionMatrix_smoothed = table(simulatedStates$states,
                        paste(rep("Z", nSim), smoothed_max, sep=""))
  acc_smoothed = sum(diag(confusionMatrix_smoothed))/sum(confusionMatrix_smoothed)

  # Filtered
  confusionMatrix_filtered = table(simulatedStates$states,
                          paste(rep("Z", nSim), filtered_max, sep=""))
  acc_filtered = sum(diag(confusionMatrix_filtered))/sum(confusionMatrix_filtered)
```

```r
  # Viterbi
  confusionMatrix_viterbi = table(simulatedStates$states, hmm_viterbi)
  acc_viterbi = sum(diag(confusionMatrix_viterbi))/sum(confusionMatrix_viterbi)

  return(list(acc_smoothed = acc_smoothed,
              acc_filtered = acc_filtered,
              acc_viterbi = acc_viterbi))
}

df = t(sapply(1:100, function(x) {
  return(simulate_hmm(robot_hmm))
}))
```



The average accuracies are 0.6574 (Smoothed), 0.518 (Filtered) and 0.5029 (Viterbi).

Smoothing is in general better than Filtering, as Filtering is only allowed to use past data whereas Smoothing is also allowed to look at future observations.

The Viterbi algorithm has to fulfill another constraint which is that each step must make sense. This means that the robot is only allowed to move one field at a time. For some scenarios it might make sense to define this constraint, but when not needed it is better to just use the most probable field (which is what Smoothing is doing) and therefore has a higher overall accuracy. The "price" for the higher accuracy is that the logical path does not always make sense. One has to choose which metric is more important for the given problem.

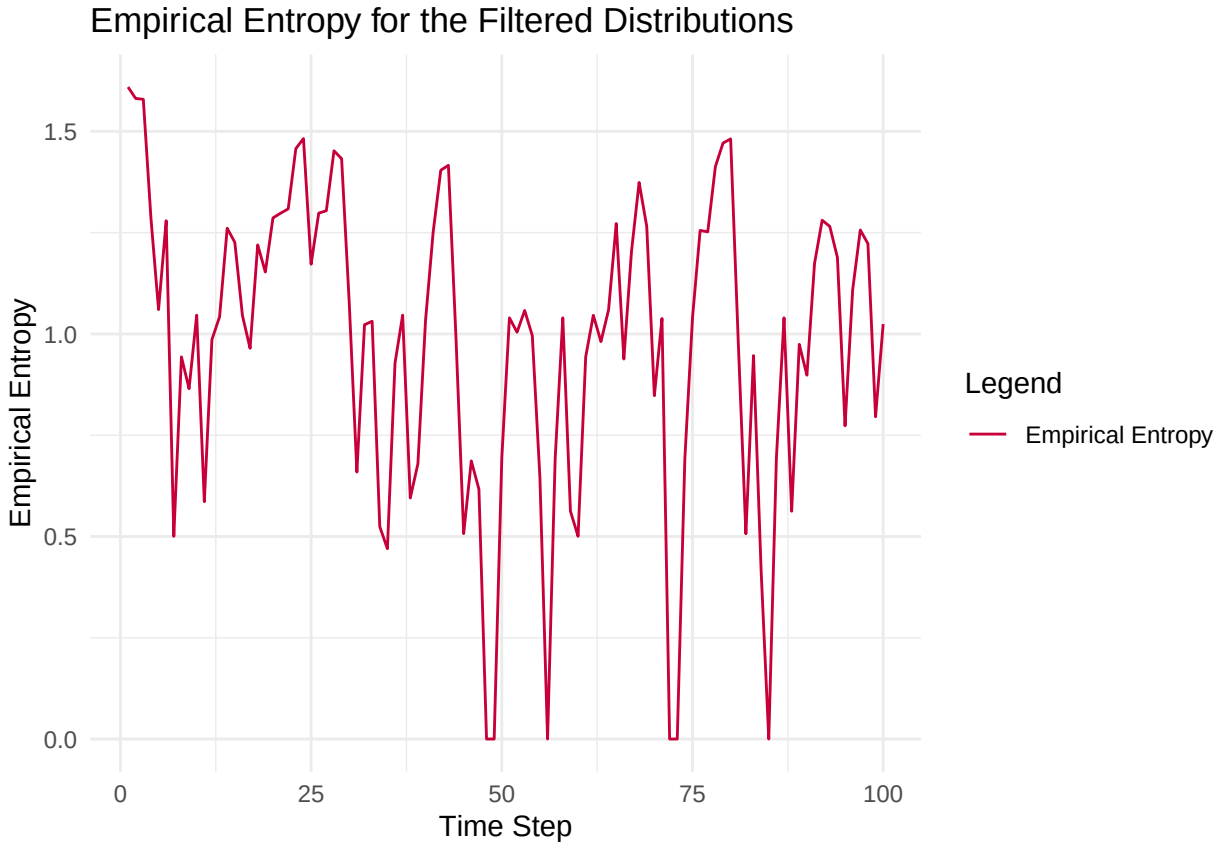
6 Is Having more Observations Always Better?

Task: 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`.

```
# Estimates the Shannon entropy H of the random variable Y from the
# corresponding observed counts y by plug-in of the empirical frequencies.
empirical_entropy = apply(filtered, 2, entropy.empirical)
```

Answer: We observe that the entropy does not change, even given the fact that each time step counts in for one more data point. Therefore it is not true that given more observations, we know better where the robot is.



7 Probabilities of Time Step 101

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

Answer: Taking our probabilites from out last state and the transition matrix, we obtain the next distribution for step 101 by a simple matrix multiplication, as we have a first over Markov process in the background.

```
post = transProbs %*% filtered[,100]
rownames(post) = names(filtered[,100])
post
```

```
##           [,1]
```

```
## Z1  0.00000000
## Z2  0.00000000
## Z3  0.00000000
## Z4  0.00000000
## Z5  0.00000000
## Z6  0.00000000
## Z7  0.08899421
## Z8  0.33201685
## Z9  0.41100579
## Z10 0.16798315
```

So the most probable state would be given by:

```
# Can sometimes result in two maximum values
post[which.max(post),]
```

```
##        Z9
## 0.4110058
```

8 Source Code

```
library(HMM)
library(ggplot2)
library(entropy)
library(reshape2)
knitr::opts_chunk$set(echo = TRUE)

N = 10

# Defining States Z1, Z2, ..., ZN
states = paste(rep("Z", N), 1:N, sep = "")

# Defining Symbols S1, S2, ..., SN
symbols = paste(rep("S", N), 1:N, sep = "")

# Starting Probabilities
startProbs = rep(1/N, N)

# Transition Probabilities
transProbs = matrix(0, ncol = N, nrow = N)
# Staying in the current state with 0.5 probability is just the diagonal
diag(transProbs) = 0.5
# Moving to the next is also 0.5
diag(transProbs[,-1]) = 0.5
transProbs[10, 1] = 0.5

# Emission Probabilities
emissionProbs = matrix(0, ncol = N, nrow = N)

# 0.2 For i-2 to i+2
for (i in 1:N) {
  for (j in c(3:-1)) {
    emissionProbs[((i-j)%%N)+1,i] = 0.2
```

```
  }
}

robot_hmm = initHMM(States = states,
                    Symbols = symbols,
                    startProbs = startProbs,
                    transProbs = transProbs,
                    emissionProbs = emissionProbs)


nSim = 100

simulatedStates = simHMM(robot_hmm, nSim)


# Custom implementations of the forward and backward algorithm

custom_forward = function(hmm, observations) {

  Z = matrix(NA, ncol=length(hmm$States), nrow=length(observations))

  Z[1,] = hmm$emissionProbs[, observations[1]] * hmm$startProbs

  for (t in 2:length(observations)) {
    Z[t, ] = hmm$emissionProbs[, observations[t]] * (Z[t-1,] %*% hmm$transProbs)
  }

  return(t(Z))
}

custom_backward = function(hmm, observations) {

  Z = matrix(NA, ncol=length(hmm$States), nrow=length(observations))

  Z[length(observations),] = 1

  for (t in ((length(observations)-1):0)) {
    for (state in 1:length(hmm$States)) {
     Z[t, state] = sum(Z[t+1,] * hmm$emissionProbs[,observations[t+1]] * transProbs[state,])
    }
  }

  return(t(Z))
}


# The library returns the probabilities logged, so we have to de-log
alpha = exp(forward(robot_hmm, simulatedStates$observation))
beta = exp(backward(robot_hmm, simulatedStates$observation))

# Filtered
# Normalizing, for each column x = x/sum(x), done with prop.table
filtered = prop.table(alpha, 2)
```

```r
# Smoothed
# Can either be done manually or using the function posterior (== smoothed) in
# this package
# AUTOMATIC
smoothed_automatically = posterior(robot_hmm, simulatedStates$observation)
# MANUALLY (Instead of division prop.table would work as well)
smoothed_manually = alpha * beta / colSums(alpha * beta)

# Path
hmm_viterbi = viterbi(robot_hmm, simulatedStates$observation)

# Print Filtered
filtered[,(nSim-5):nSim]

# Print Smoothed
smoothed_automatically[,(nSim-5):nSim]
smoothed_manually[,(nSim-5):nSim]

# Print Viterbi
hmm_viterbi


ggplot(data = melt(smoothed_automatically),
       aes(y=states, x=index, fill=value)) +
       geom_raster()


ggplot(data = melt(filtered),
       aes(y=states, x=index, fill=value)) +
       geom_raster()


# Taking the max for each observation
# Smoothed
filtered_max = max.col(t(filtered), "first")
# Filtered
smoothed_max = max.col(t(smoothed_automatically), "first")

# ACC
# Smoothed
confusionMatrix_smoothed = table(simulatedStates$states,
                         paste(rep("Z", nSim), smoothed_max, sep=""))
acc_smoothed = sum(diag(confusionMatrix_smoothed))/sum(confusionMatrix_smoothed)

# Filtered
confusionMatrix_filtered = table(simulatedStates$states,
                         paste(rep("Z", nSim), filtered_max, sep=""))
acc_filtered = sum(diag(confusionMatrix_filtered))/sum(confusionMatrix_filtered)

# Viterbi
confusionMatrix_viterbi = table(simulatedStates$states, hmm_viterbi)
acc_viterbi = sum(diag(confusionMatrix_viterbi))/sum(confusionMatrix_viterbi)
```

```
acc_smoothed
acc_filtered
acc_viterbi


simulate_hmm = function(hmm, nSim = 100) {

  # Simulate
  simulatedStates = simHMM(robot_hmm, nSim)

  # Filtered
  alpha = exp(forward(hmm, simulatedStates$observation))
  filtered = prop.table(alpha, 2)
  filtered_max = max.col(t(filtered))

  # Smoothed
  smoothed = posterior(hmm, simulatedStates$observation)
  smoothed_max = max.col(t(smoothed))

  # Viterbi
  hmm_viterbi = viterbi(hmm, simulatedStates$observation)

  # Accuracies
  confusionMatrix_smoothed = table(simulatedStates$states,
                         paste(rep("Z", nSim), smoothed_max, sep=""))
  acc_smoothed = sum(diag(confusionMatrix_smoothed))/sum(confusionMatrix_smoothed)

  # Filtered
  confusionMatrix_filtered = table(simulatedStates$states,
                           paste(rep("Z", nSim), filtered_max, sep=""))
  acc_filtered = sum(diag(confusionMatrix_filtered))/sum(confusionMatrix_filtered)

  # Viterbi
  confusionMatrix_viterbi = table(simulatedStates$states, hmm_viterbi)
  acc_viterbi = sum(diag(confusionMatrix_viterbi))/sum(confusionMatrix_viterbi)

  return(list(acc_smoothed = acc_smoothed,
              acc_filtered = acc_filtered,
              acc_viterbi = acc_viterbi))
}

df = t(sapply(1:100, function(x) {
  return(simulate_hmm(robot_hmm))
}))


df = data.frame(list(index = 1:100), df)
df$acc_smoothed = as.numeric(df$acc_smoothed)
df$acc_filtered = as.numeric(df$acc_filtered)
df$acc_viterbi = as.numeric(df$acc_viterbi)

ggplot(df) +
  geom_line(aes(x = index, y = acc_smoothed, colour = "Accuracy Smoothed")) +
```

```
  geom_line(aes(x = index, y = acc_filtered, colour = "Accuracy Filtered")) +
  geom_line(aes(x = index, y = acc_viterbi, colour = "Accuracy Viterbi")) +
  labs(title = "Accuracy for Different Methods",
       y = "Accuracy",
       x = "Simulation", color = "Legend") +
  scale_color_manual(values = c("#C70039", "#FF5733", "#581845")) +
  theme_minimal()


# Estimates the Shannon entropy H of the random variable Y from the
# corresponding observed counts y by plug-in of the empirical frequencies.
empirical_entropy = apply(filtered, 2, entropy.empirical)


df = data.frame(empirical_entropy)
df = data.frame(list(index = 1:100), df)

ggplot(df) +
  geom_line(aes(x = index, y = empirical_entropy, colour = "Empirical Entropy")) +
  labs(title = "Empirical Entropy for the Filtered Distributions",
       y = "Empirical Entropy",
       x = "Time Step", color = "Legend") +
  scale_color_manual(values = c("#C70039")) +
  theme_minimal()


post = transProbs %*% filtered[,100]
rownames(post) = names(filtered[,100])
post


# Can sometimes result in two maximum values
post[which.max(post),]
```