TDDE15 - Lab 2

Erik Jareman

2022-09-14

```
library(HMM)
library(entropy)
set.seed(1)
```

Task 1

If the robot is in S1, it can stay or move to S2 with equal probability, giving the first transition matrix row: {.5, .5, 0, 0, 0, 0, 0, 0, 0, 0}. The same goes for the second row with the difference being that it starts in position 2: {0, .5, .5, 0, 0, ...}...

For the emission matrix, we set the current state and the four closest states to 0.2 as the probabilities are equal (divided on five states).

We execute the above mentioned steps for each row to create the transition- and emission matrices, and use these to build the HMM.

```
transition = matrix(0, 10, 10)
emission = matrix(0, 10, 10)
for (i in 0:9)
{
  # Set transition values to .5 for S(current) and S(current+1)
  transition[i+1, i:(i+1)%%10 +1] = 0.5

  # Set emission values to .2 for S(current) and the four closest columns
  emission[i+1, (i-2):(i+2)%%10 +1] = 0.2
}

transition
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]  0.5  0.5  0.0  0.0  0.0  0.0  0.0  0.0  0.0   0.0
##  [2,]  0.0  0.5  0.5  0.0  0.0  0.0  0.0  0.0  0.0   0.0
##  [3,]  0.0  0.0  0.5  0.5  0.0  0.0  0.0  0.0  0.0   0.0
##  [4,]  0.0  0.0  0.0  0.5  0.5  0.0  0.0  0.0  0.0   0.0
##  [5,]  0.0  0.0  0.0  0.0  0.5  0.5  0.0  0.0  0.0   0.0
##  [6,]  0.0  0.0  0.0  0.0  0.0  0.5  0.5  0.0  0.0   0.0
##  [7,]  0.0  0.0  0.0  0.0  0.0  0.0  0.5  0.5  0.0   0.0
##  [8,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.5  0.5   0.0
##  [9,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.5   0.5
## [10,]  0.5  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0   0.5
```

```
emission
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]  0.2  0.2  0.2  0.0  0.0  0.0  0.0  0.0  0.2   0.2
##  [2,]  0.2  0.2  0.2  0.2  0.0  0.0  0.0  0.0  0.0   0.2
##  [3,]  0.2  0.2  0.2  0.2  0.2  0.0  0.0  0.0  0.0   0.0
##  [4,]  0.0  0.2  0.2  0.2  0.2  0.2  0.0  0.0  0.0   0.0
##  [5,]  0.0  0.0  0.2  0.2  0.2  0.2  0.2  0.0  0.0   0.0
##  [6,]  0.0  0.0  0.0  0.2  0.2  0.2  0.2  0.2  0.0   0.0
##  [7,]  0.0  0.0  0.0  0.0  0.2  0.2  0.2  0.2  0.2   0.0
##  [8,]  0.0  0.0  0.0  0.0  0.0  0.2  0.2  0.2  0.2   0.2
##  [9,]  0.2  0.0  0.0  0.0  0.0  0.0  0.2  0.2  0.2   0.2
## [10,]  0.2  0.2  0.0  0.0  0.0  0.0  0.0  0.2  0.2   0.2
```

```
# Build the HMM using transition and emission matrices created
HMM = initHMM(States = rep(1:10),
              Symbols = rep(1:10),
              transProbs = transition,
              emissionProbs = emission)
```

Task 2

Simulate 100 time steps for the HMM.

```
sim = simHMM(HMM, 100)
sim
```

```
## $states
##   [1]  4  4  5  5  6  7  8  9  9 10  1  2  2  3  4  4  5  5  6  7  8  8  9 10  1
##  [26]  2  3  3  4  4  4  5  6  6  7  8  9 10 10  1  1  1  1  1  1  1  2  3  4  5
##  [51]  6  7  7  7  7  7  7  8  9 10 10  1  2  3  4  4  4  5  6  7  7  8  8  8  8
##  [76]  9  9 10 10 10  1  1  2  3  4  4  5  6  6  6  6  6  7  8  9  9 10  1  1  1
##
## $observation
##   [1]  2  3  3  6  4  7  8  1  8  2  3 10  4  1  4  4  4  7  5  9  6 10  1 10 10
##  [26]  1  1  4  3  5  5  7  8  4  6 10  1  2  9  1 10 10  2  2 10  1  2  5  4  6
##  [51]  4  5  7  5  5  8  5  8  7  8  8  3  1  5  6  5  4  3  4  7  9  6  6  7  7
##  [76]  8 10  1  1  9  2  9  3  1  6  4  4  4  6  5  4  7  8  6  7  1 10  3  2 10
```

Task 3

Use the forward-backward algorithm to compute the filtered- and smoothed probability distributions to get the robot position at time t. Find the most likely path using the Viterbi algorithm.

```
# Discard hidden states from obtained sample
simObservation = sim$observation
simStates = sim$states

# Calculate alpha and beta to be used in filtering and smoothing (+log convert)
alpha = exp(forward(HMM, simObservation))
beta = exp(backward(HMM, simObservation))
```

```r
# Calculate filtering- and smoothing probability distributions
filteringPD = matrix(NA, 10, 100)
smoothingPD = matrix(NA, 10, 100)
for (i in 1:10)
{
  # Filtering, p(z(t)|x(0:t))
  filteringPD[i, ] = alpha[i, ] / sum(alpha[i, ])

  # Smoothing, p(z(t)|x(0:T))
  smoothingPD[i, ] = alpha[i, ]*beta[i, ] / sum(alpha[i, ]*beta[i, ])
}

# Normalize pd (not really needed since we use which.max later (?) )
filteringPD = prop.table(x=filteringPD, margin=2)
smoothingPD = prop.table(x=smoothingPD, margin=2)

# Predict state at time t as the state (row) with the highest PD (column) value
filteringPred = apply(X=filteringPD, MARGIN=2, FUN=which.max)
smoothingPred = apply(X=smoothingPD, MARGIN=2, FUN=which.max)

# Compute the most probable path using the Viterbi algorithm
probPath = viterbi(HMM, simObservation)
```

Task 4

Compute the accuracy of the filtered pd, the smoothed pd, and the most probable path.

```r
# Calculate accuracy compared to actual observations
filteringAcc = table(filteringPred==simStates)[2]/length(simStates)
smoothingAcc = table(smoothingPred==simStates)[2]/length(simStates)
probPathAcc = table(probPath==simStates)[2]/length(simStates)
```

filteringAcc: 0.47
smoothingAcc: 0.71
probPathAcc: 0.51

Task 5

Calculate the accuracy for the smoothed distribution, the filtered distribution, and the most probable path with different simulated samples.

```r
sim2 = simHMM(HMM, 100)

alpha2 = exp(forward(HMM, sim2$observation))
beta2 = exp(backward(HMM, sim2$observation))

filteringPD2 = matrix(NA, 10, 100)
smoothingPD2 = matrix(NA, 10, 100)
for (i in 1:10)
{
  # Filtering p(z(t)|x(0:t))
```

```
  filteringPD2[i, ] = alpha2[i, ] / sum(alpha2[i, ])

  # Smoothing p(z(t)|x(0:T))
  smoothingPD2[i, ] = alpha2[i, ]*beta2[i, ] / sum(alpha2[i, ]*beta2[i, ])
}

filteringPD2 = prop.table(x=filteringPD2, margin=2)
smoothingPD2 = prop.table(x=smoothingPD2, margin=2)

filteringPred2 = apply(X=filteringPD2, MARGIN=2, FUN=which.max)
smoothingPred2 = apply(X=smoothingPD2, MARGIN=2, FUN=which.max)

probPath2 = viterbi(HMM, sim2$observation)

# Calculate accuracy compared to actual observations
filteringAcc2 = table(filteringPred2==sim2$states)[2]/length(sim2$states)
smoothingAcc2 = table(smoothingPred2==sim2$states)[2]/length(sim2$states)
probPathAcc2 = table(probPath2==sim2$states)[2]/length(sim2$states)
```

Accuracy for the different methods and both our data sets:

filteringAcc: 0.47
filteringAcc2: 0.43

smoothingAcc: 0.71
smoothingAcc2: 0.66

probPathAcc: 0.51
probPathAcc2: 0.51

The smoothed distribution is more accurate than the filtered distribution for both of our generated data sets. The reason for this is that the smoothed distribution utilizes future observations as well as historical observations, whilst the filtered distribution only utilizes the latter. The smoothed distribution has more data to base its probability distribution on.

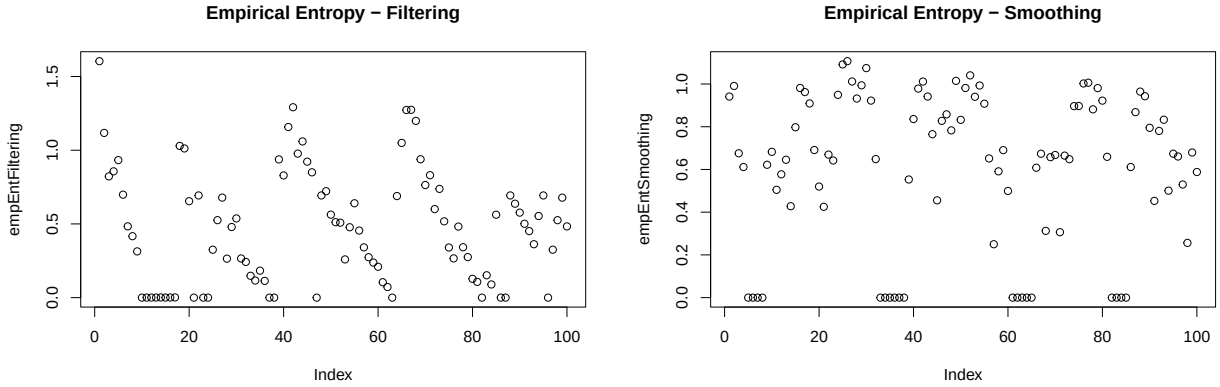
The smoothed distribution is more accurate than the most probable path for both of out data sets. The reason for this is that the most accurate path has to follow a path that is actually possible, meaning that the robot can not skip states, it can only either stay or move forward for each predicted state. Due to this dependency, the most probable path approach can end up in situations where it would make sense to make a prediction that is two steps forward, but the dependency only allows for predictions one step forward.

Task 6

Calculate the empirical entropy for the filtering- and smoothing probability distributions.

```
# Apply empirical.entropy on each pd (column)
empEntFiltering = apply(filteringPD2, 2, FUN=entropy.empirical)
empEntSmoothing = apply(smoothingPD, 2, FUN=entropy.empirical)
```

```
plot(empEntFiltering, main="Empirical Entropy - Filtering")
plot(empEntSmoothing, main="Empirical Entropy - Smoothing")
```



The graphs do not indicate that more observations will result in a more accurate model. The entropy for each prediction seems to fall somewhere between 0 and 1.5, and it does not seem like the entropy gets lower later in time.

Task 7

Compute the probabilities of the hidden states for the time step 101.

```
# Multiply the state probabilities at t=100 with the action probabilities for
# the given states
transProbs101 = filteringPD[, 100]*transition

# Summarize the probabilities for each state/action probability over state only
PD101 = colSums(transProbs101)
```

```
table100and101
```

```
##         t=100     t=101
## S1  0.3169601 0.1584801
## S2  0.6830399 0.5000000
## S3  0.0000000 0.3415199
## S4  0.0000000 0.0000000
## S5  0.0000000 0.0000000
## S6  0.0000000 0.0000000
## S7  0.0000000 0.0000000
## S8  0.0000000 0.0000000
## S9  0.0000000 0.0000000
## S10 0.0000000 0.0000000
```