LAB 2

2.1

Build a hidden Markov model (HMM) for the scenario described below. - The robot walks around a ring divided into 10 sectors. At any time, the robot is in one sector and can either stay or move to the next sector with equal probability. You don't have direct observations of the robot but can access a tracking device. The device inaccurately reports the robot's position in one of the sectors [i-2, i+2] with equal probability if the robot is in sector i. - Note: The `emissionProbs` matrix should be of size [number of states]x[number of symbols] (not as stated in the documentation).

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

states = 1:10
symbols = 1:10
start_probs = rep(1/length(states), length(states))

# Define the transition probabilities
trans_probs = matrix(0, nrow=length(states), ncol=length(states))
for (i in 1:(length(states)-1)) {
  trans_probs[i, i] = 0.5  # Stay in the same sector
  trans_probs[i, i+1] = 0.5  # Move to the next sector
}
# Last Sector transitions to Sector 1
trans_probs[length(states), length(states)] = 0.5
trans_probs[length(states), 1] = 0.5

# Define the emission probabilities
emission_probs = matrix(0, nrow=length(states), ncol=length(symbols))
for (i in 1:length(states)) {
  # Get the sectors in the range [i-2, i+2]
  sectors = c((i-2):(i+2)) %% length(symbols) # (-1%%10 transforms -1 to 9)
  sectors[sectors == 0] = length(symbols)  # (10%%10 transfroms 10 to 0, which is incorrect.)

  emission_probs[i, sectors] = 1/5  # Equal probability for the 5 neighboring sectors
}

hmm_model = initHMM(States=states, Symbols=symbols, startProbs=start_probs,
                    transProbs=trans_probs, emissionProbs=emission_probs)
print(hmm_model)
```

```
## $States
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $Symbols
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
##
## $startProbs
##   1   2   3   4   5   6   7   8   9  10
## 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
##
## $transProbs
##     to
## from   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
##
## $emissionProbs
##       symbols
## states   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
```

2.2

Simulate the HMM for 100 time steps.

```
simres = simHMM(hmm_model, 100)
table(simres$states, simres$observation)
```

```
##
##      1 2 3 4 5 6 7 8 9 10
##   1  2 4 3 0 0 0 0 0 1  5
##   2  2 1 1 1 0 0 0 0 0  1
##   3  3 0 0 1 2 0 0 0 0  0
##   4  0 1 2 5 3 2 0 0 0  0
##   5  0 0 2 2 0 2 2 0 0  0
##   6  0 0 0 6 2 1 1 1 0  0
##   7  0 0 0 0 4 1 3 2 2  0
##   8  0 0 0 0 0 4 2 2 0  2
##   9  4 0 0 0 0 0 2 2 0  1
##   10 2 2 0 0 0 0 0 2 2  2
```

2.3

Discard the hidden states from the sample obtained and use the remaining observations to compute the **filtered** and **smoothed** probability distributions for each of the 100 time points. Also compute the **most probable path**.

```
simobv = simres$observation
alpha = exp(forward(hmm_model, simobv))
beta = exp(backward(hmm_model, simobv))

## Filtered
filtered = matrix(0,10, length(simobv))
for (i in 1:length(simobv)) {
  filtered[,i] = alpha[,i] / sum(alpha[,i])
}

## Smoothed
smoothed = matrix(0,10, length(simobv))
for (i in 1:length(simobv)) {
  smoothed[,i] = (alpha[,i]*beta[,i]) / sum(alpha[,i]*beta[,i])
}

## Viterbi
viterbi = viterbi(hmm_model, simobv)
```

2.4

Compute the accuracy of the **filtered** and **smoothed** probability distributions, as well as of the **most probable path**. Compare with the true hidden states.

```
predict_filtered = apply(filtered, MARGIN = 2, FUN = which.max)
predict_smoothed = apply(smoothed, MARGIN = 2, FUN = which.max)

calculate_accuracy = function(predicted_values, true_values) {
  cm = table(predicted_values, true_values)
  correct_predictions = sum(diag(cm))
  total_predictions = sum(cm)
  accuracy = correct_predictions / total_predictions

  return(accuracy)
}

cat("Filtered: ", calculate_accuracy(predict_filtered, simres$states))
```

```
## Filtered:  0.56
```

```
cat(", Smoothed: ", calculate_accuracy(predict_smoothed, simres$states))
```

```
## , Smoothed:  0.72
```

```r
cat(", Viterbi: ", calculate_accuracy(viterbi, simres$states))
```

```
## , Viterbi:  0.51
```

2.5

Repeat the previous exercise with different simulated samples. Explain why the smoothed distributions are generally more accurate than the filtered distributions and the most probable paths.

```r
simres = simHMM(hmm_model, 100)

simobv = simres$observation
alpha = exp(forward(hmm_model, simobv))
beta = exp(backward(hmm_model, simobv))

## Filtered
filtered = matrix(0,10, length(simobv))
for (i in 1:length(simobv)) {
  filtered[,i] = alpha[,i] / sum(alpha[,i])
}

## Smoothed
smoothed = matrix(0,10, length(simobv))
for (i in 1:length(simobv)) {
  smoothed[,i] = (alpha[,i]*beta[,i]) / sum(alpha[,i]*beta[,i])
}

## Viterbi
viterbi = viterbi(hmm_model, simobv)

predict_filtered = apply(filtered, MARGIN = 2, FUN = which.max)
predict_smoothed = apply(smoothed, MARGIN = 2, FUN = which.max)

calculate_accuracy = function(predicted_values, true_values) {
  cm = table(predicted_values, true_values)
  correct_predictions = sum(diag(cm))
  total_predictions = sum(cm)
  accuracy = correct_predictions / total_predictions

  return(accuracy)
}

cat("Filtered: ", calculate_accuracy(predict_filtered, simres$states))
```

```
## Filtered:  0.51
```

```r
cat(", Smoothed: ", calculate_accuracy(predict_smoothed, simres$states))
```

```
## , Smoothed:  0.69
```

```r
cat(", Viterbi: ", calculate_accuracy(viterbi, simres$states))
```

```
## , Viterbi:  0.51
```

Answer: Smoothed takes into account both past and future observation, while filtered only take into account past. Smoothed takes into account more observations than filtered.

Smoothed is more accurate than Viterbi, due to Viterbi having the constraint of predicting a valid path. Viterbi might miss important alternative states that are highly probable, but not part of the most likely path.

2.6

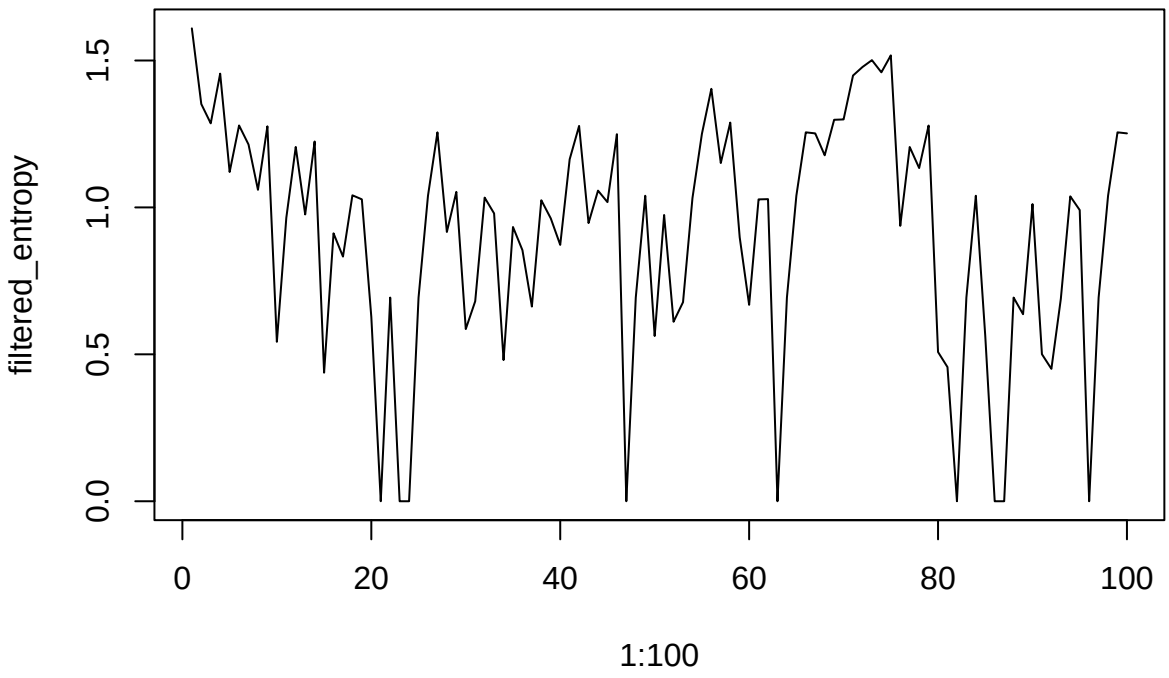
Is it always true that the more observations you receive, the better you can predict where the robot is?

```r
library(entropy)

# Function to compute entropy for each time step
compute_entropy = function(filtered_distribution) {
  entropies = apply(filtered_distribution, 2, entropy.empirical)
  return(entropies)
}

filtered_entropy = compute_entropy(filtered)

plot(1:100, filtered_entropy, type="l")
```



Answer: It is not true that the later in time the better you know where the robot is. The entropy doesn't seem to be affected by the amount of observations, as seen in the graph.

Since there is ambiguity in the observations, more observations will not necessarily improve accuracy. If the observations were of high quality, it is likely that more observations would lower the entropy. In the plot, it does not seem to increase or decrease, however there are some cases where the entropy is 0, indicating that the probability distribution is concentrated on a single state

2.7

For one of the samples of length 100, compute the probabilities of the hidden states for time step 101.

```
filtered_100 = filtered[, 100]
predicted_101 = trans_probs %*% filtered_100
print(predicted_101)
```

```
##              [,1]
##  [1,] 0.00000000
##  [2,] 0.00000000
##  [3,] 0.00000000
##  [4,] 0.00000000
##  [5,] 0.13333333
##  [6,] 0.33333333
##  [7,] 0.33333333
##  [8,] 0.16666667
```

```
##  [9,] 0.03333333
## [10,] 0.00000000
```