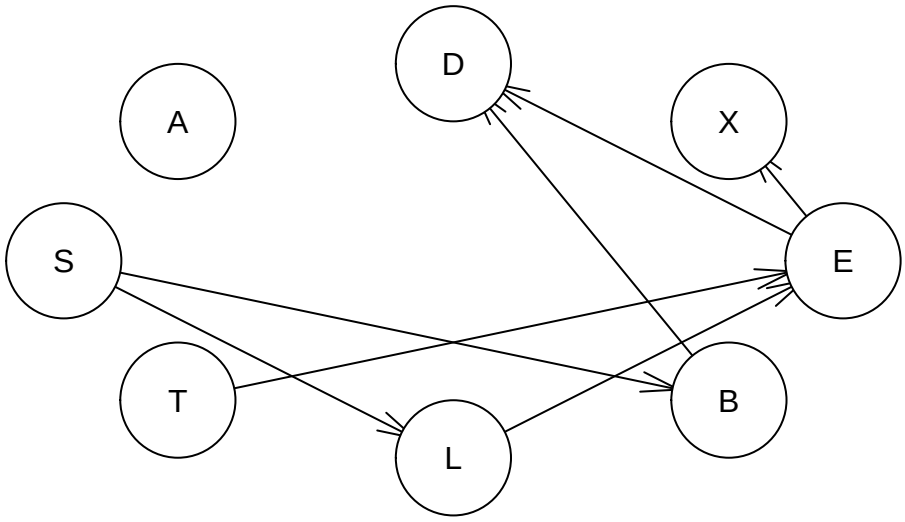
LAB 1

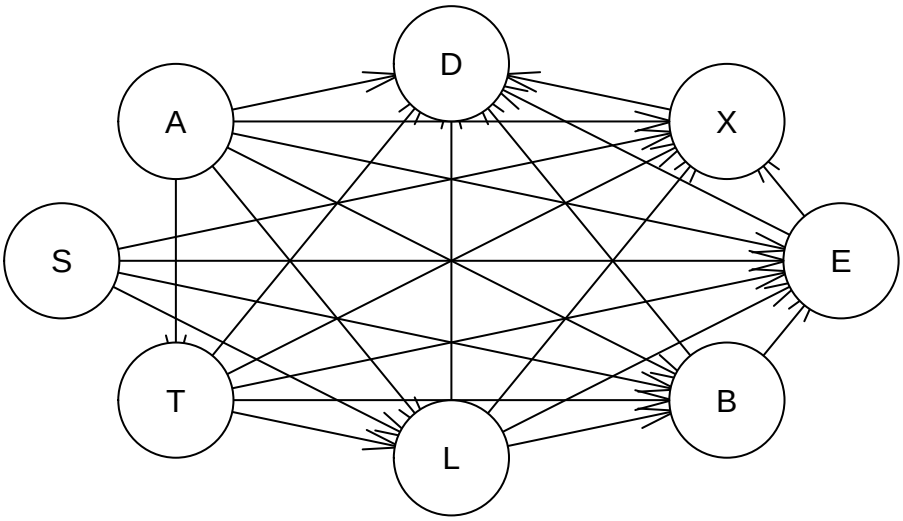
1.1

Show that multiple runs of the hill-climbing algorithm can return non-equivalent Bayesian network (BN) structures. Explain why this happens. Use the Asia dataset, which is included in the `bnlearn` package. To load the data, run `data("asia")`. Recall from the lectures that the concept of non-equivalent BN structures has a precise meaning. - Hint: Check the function `hc` in the `bnlearn` package. Note that you can specify the initial structure, the number of random restarts, the score, and the equivalent sample size (a.k.a imaginary sample size) in the BDeu score. You may want to use these options to answer the question. You may also want to use the functions `plot`, `arcs`, `vstructs`, `cpdag`, and `all.equal`.

```
rm(list=ls())
library(bnlearn)
set.seed(12345)
data("asia")

bn1 = hc(asia, start = NULL, score = "bde", restart = 1, iss = 1)
# arcs(bn1)
# vstructs(bn1)
plot(bn1)
```




```
bn2 = hc(asia, start = NULL, score = "bde", restart = 1, iss = 1000)
# arcs(bn2)
# vstructs(bn2)
plot(bn2)
```




```
# Cpdag to see equivalent class to get.
# cat("Different ISS gives non-equivalent classes: ", all.equal(cpdag(bn1), cpdag(bn2)))
```

Answer: Definition of equivalent BN structure: "[...] the same adjacencies and unshielded colliders" They are non-equivalent. For example, in the first BN A,D is not adjacent, which they are in the second. The reason for the second graph being way more complex is due to the iss term being set very high, reducing regularization. This can happen due to HC finding local optimas, not global.

```
start1 = random.graph(colnames(asia))
startbn1 = hc(asia, start = start1)
start2 = random.graph(colnames(asia))
startbn2 = hc(asia, start = start2)

cat("\nDifferent start gives non-equivalent classes: ", all.equal(cpdag(startbn1), cpdag(startbn2)))
```

```
##
## Different start gives non-equivalent classes:  Different number of directed/undirected arcs
```

1.2

Learn a BN from 80% of the Asia dataset. Learn both the structure and the parameters. Use any learning algorithm and settings that you consider appropriate. Use the BN learned to classify the remaining 20% of the dataset in two classes: S = yes and S = no. Compute the posterior probability

distribution of S for each case and classify it in the most likely class. - You may use exact or approximate inference with the help of the `bnlearn` and `gRain` packages. Report the confusion matrix (true/false positives/negatives). Compare your results with those of the true Asia BN by running: `dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")`.

```
rm(list=ls())
library(bnlearn)
library(gRain)
```

```
## Loading required package: gRbase
```

```
##
## Attaching package: 'gRbase'
```

```
## The following objects are masked from 'package:bnlearn':
##
##     ancestors, children, nodes, parents
```

```
set.seed(12345)
data("asia")

n=dim(asia)[1]
id=sample(1:n, floor(n*0.8))
train_data=asia[id,]
test_data=asia[-id,]

bn = hc(train_data)
dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")

fit = bn.fit(x = bn, data = train_data)
true_fit = bn.fit(x = dag, data = train_data)

grain = as.grain(fit)
true_grain = as.grain(true_fit)

compile = compile(grain)
true_compile = compile(true_grain)

predict = function (network){
  predictions = c()
  for (i in 1:nrow(test_data)) {
    evidence = setEvidence(network, colnames(test_data)[-2], as.vector(as.matrix(test_data[i, -2])))
    query = querygrain(evidence, colnames(test_data)[2])$S

    predictions = c(predictions, ifelse(query["yes"]>query["no"], "yes", "no"))
  }
  return(predictions)
}

table(predict(compile), test_data$S)
```

```
##
```

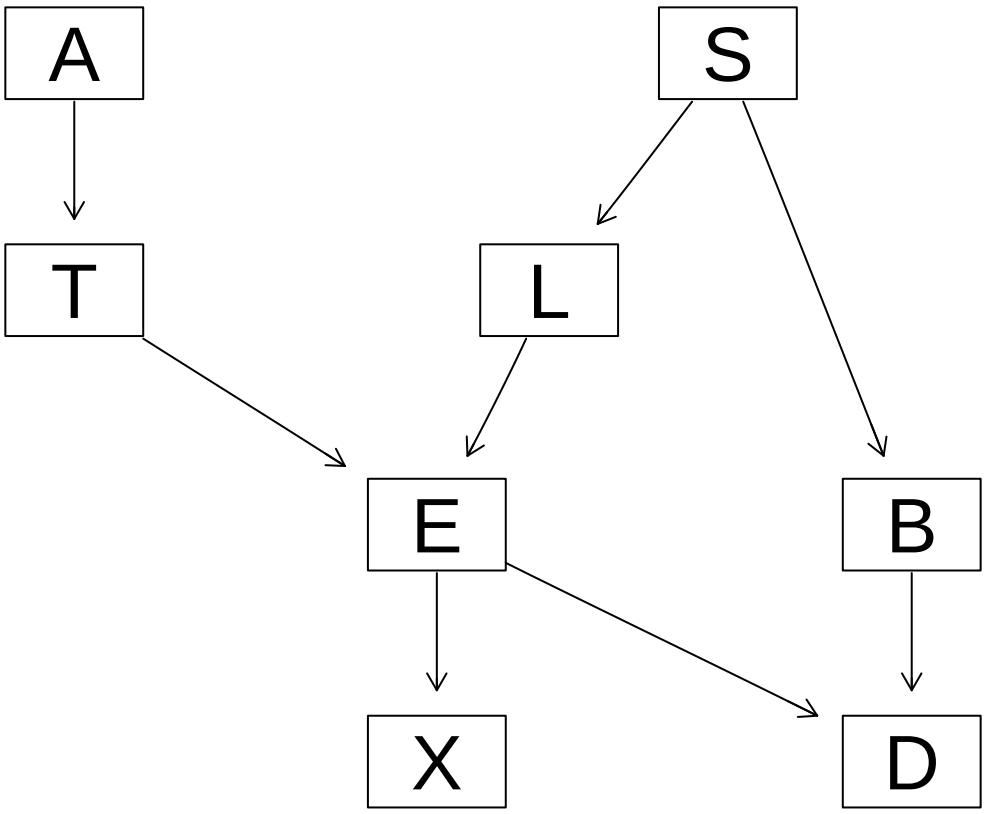
```
##        no yes
##   no  337 121
##   yes 176 366
```

```
table(predict(true_compile), test_data$S)
```

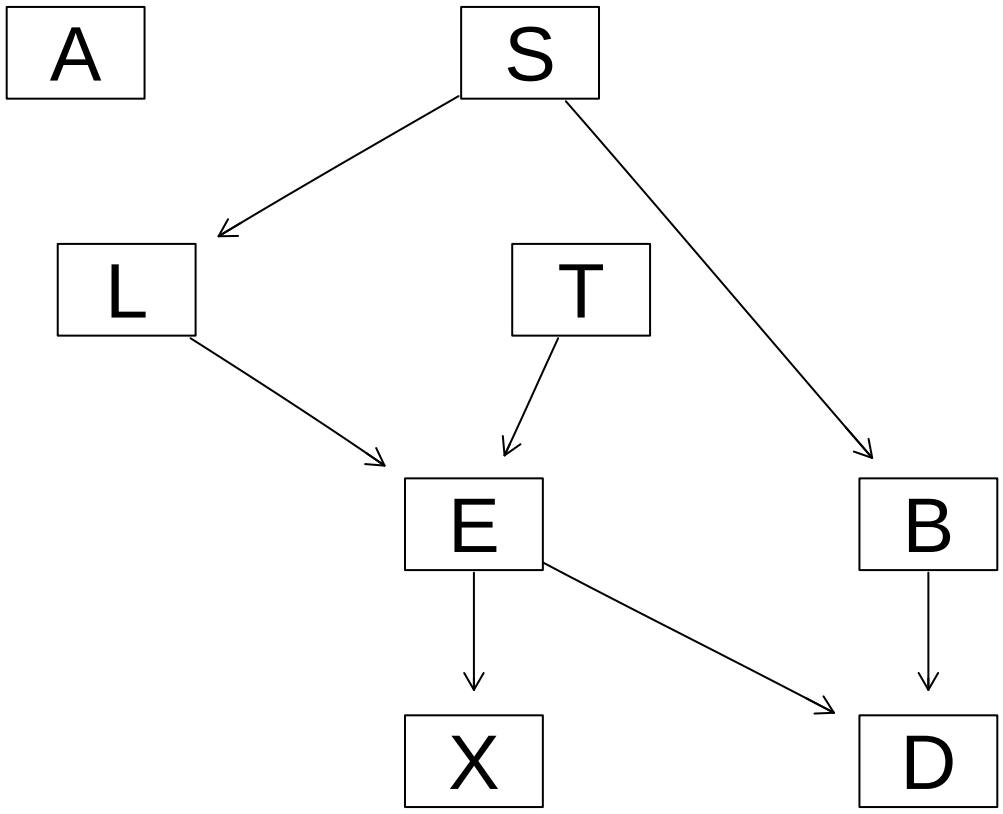
```
##
##        no yes
##   no  337 121
##   yes 176 366
```

```
graphviz.plot(dag)
```

```
## Loading required namespace: Rgraphviz
```




```
graphviz.plot(bn)
```



Answer: The confusion matrices are very similar. Only difference is edge A->T in the true graph.

1.3

Classify S given observations only for the Markov blanket of S (its parents, children, and the parents of its children minus S itself). Report the confusion matrix again. - Hint: You may want to use the function `mb` from the `bnlearn` package.

Definition: "Find the minimal set of nodes that separates a given node from the rest. This set is called the Markov blanket of the given node."

```
mb = mb(fit, c("S"))
true_mb = mb(true_fit, c("S"))

predict_mb = function (network, m_b){
  predictions = c()
  for (i in 1:nrow(test_data)) {
    evidence = setEvidence(network, m_b, as.vector(as.matrix(test_data[i, m_b])))
    query = querygrain(evidence, colnames(test_data)[2])$S

    predictions = c(predictions, ifelse(query["yes"]>query["no"], "yes", "no"))
  }
  return(predictions)
}

table(predict_mb(compile, mb), test_data$S)
```

```
##
##        no yes
##   no  337 121
##   yes 176 366
```

```
table(predict_mb(true_compile, true_mb), test_data$S)
```

```
##
##        no yes
##   no  337 121
##   yes 176 366
```

Answer: The confusion matrices are once again similar. Even identical now.

1.4

Repeat exercise (2) using a naive Bayes classifier. Model it as a BN (Bayesian Network). Create the BN by hand; do not use `naive.bayes` from the `bnlearn` package. - Hint: Check http://www.bnlearn.com/examples/dag/ for information on how to create a BN by hand.

```
# Create the Naive Bayes structure manually
nb_network = model2network("[S][A|S][B|S][D|S][E|S][L|S][T|S][X|S]")

# Learn the parameters from the training data
nb_fit <- bn.fit(nb_network, data = train_data)

# Convert to gRain object and compile it for inference
nb_grain <- compile(as.grain(nb_fit))

# Use the same prediction function as in Question 2
table(predict(nb_grain), test_data$S)
```

```
##
##        no yes
##   no  359 180
##   yes 154 307
```

```
table(predict(true_compile), test_data$S)
```

```
##
##        no yes
##   no  337 121
##   yes 176 366
```

Answer: The confusion matrices now differ.

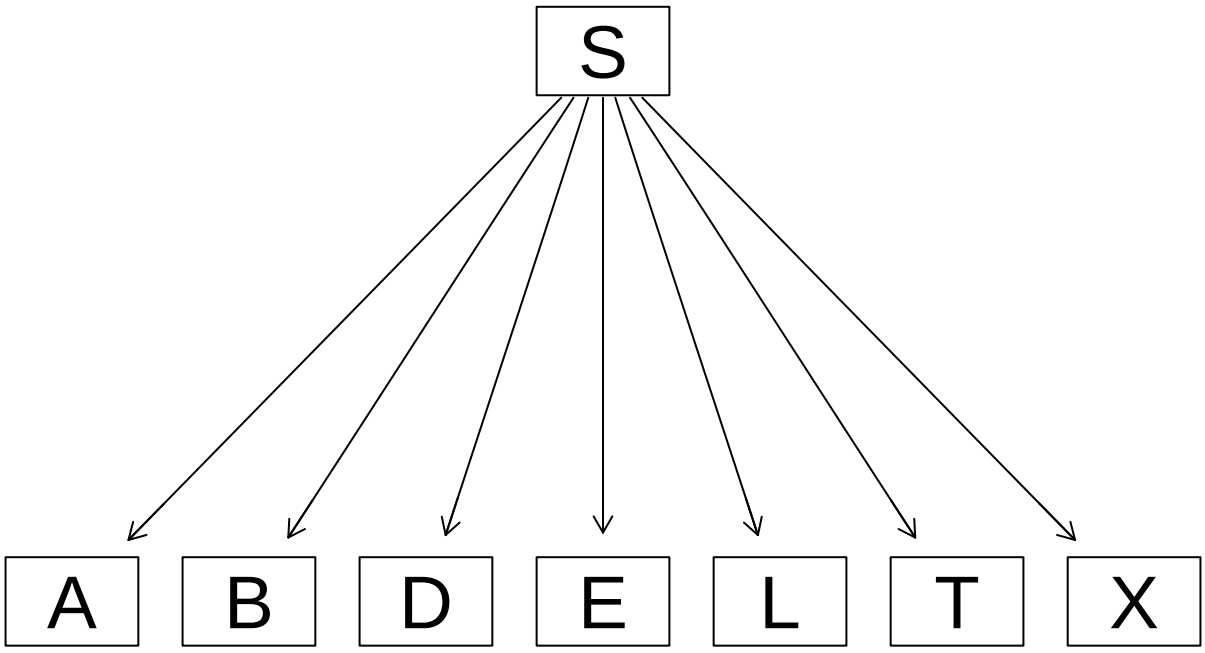
1.5

Explain why you obtain the same or different results in exercises (2)-(4).

Answer: In question 3 we get a similar result as to question 2, because when you know the states of the nodes in the Markov blanket, no other node outside this set can provide any additional information about the node in question (S), as it is then separated form the network.

Question 4 gives a different result because of the restriction "the predictive variables are independent given the class variable", which restricts us to a too underfitted network. See below.

```
graphviz.plot(nb_network)
```



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
```

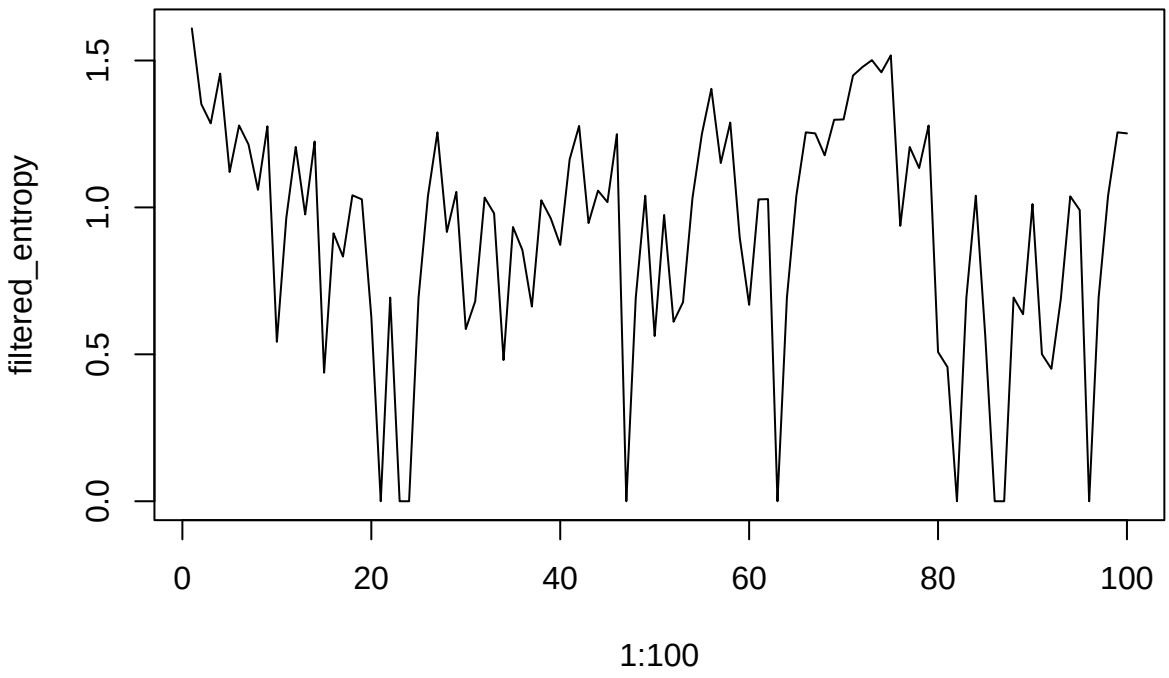
Answer: Smoothed takes into account both past and future observation, while filtered only take into account past. Smoothed takes into account more observations than filtered.

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
```

LAB 3

3.1

- You are provided with a gridworld environment where an agent can move up, down, left, or right. The agent starts each episode at a specific location and receives rewards based on the environment's layout.
- Implement the **Greedy** and **Epsilon-Greedy** policies in the functions `GreedyPolicy` and `EpsilonGreedyPolicy` respectively.
- Implement the **Q-learning** algorithm in the function `q_learning`, which will update the Q-table based on the agent's actions and rewards.

```
rm(list = ls())

# By Jose M. Peña and Joel Oskarsson.
# For teaching purposes.
# jose.m.pena@liu.se.

##########################################################################################################
# Q-learning
##########################################################################################################

# install.packages("ggplot2")
# install.packages("vctrs")
library(ggplot2)
set.seed(1234)

arrows <- c("^", ">", "v", "<")
action_deltas <- list(c(1,0), # up
                      c(0,1), # right
                      c(-1,0), # down
                      c(0,-1)) # left

vis_environment <- function(iterations=0, epsilon = 0.5, alpha = 0.1, gamma = 0.95, beta = 0){

  # Visualize an environment with rewards.
  # Q-values for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  #
  # Args:
  #   iterations, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #   H, W (global variables): environment dimensions.

  df <- expand.grid(x=1:H,y=1:W)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,1],NA),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
```

```
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,2],NA),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,3],NA),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,4],NA),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y)
    ifelse(reward_map[x,y] == 0,arrows[GreedyPolicy(x,y)],reward_map[x,y]),df$x,df$y)
  df$val5 <- as.vector(foo)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,max(q_table[x,y,]),
                                     ifelse(reward_map[x,y]<0,NA,reward_map[x,y])),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
          scale_fill_gradient(low = "white", high = "green", na.value = "red", name = "") +
          geom_tile(aes(fill=val6)) +
          geom_text(aes(label = val1),size = 4,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 4,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 4,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 4,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 10) +
          geom_tile(fill = 'transparent', colour = 'black') +
          ggtitle(paste("Q-table after ",iterations," iterations\n",
                        "(epsilon = ",epsilon,", alpha = ",
                        alpha,"gamma = ",gamma,", beta = ",beta,")")) +
          theme(plot.title = element_text(hjust = 0.5)) +
          scale_x_continuous(breaks = c(1:W),labels = c(1:W)) +
          scale_y_continuous(breaks = c(1:H),labels = c(1:H)))

}

GreedyPolicy <- function(x, y){

  # Get a greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.
  q_values = q_table[x, y, ]

  best_actions = which(q_values == max(q_values))
  if (length(best_actions) > 1) {
    action = sample(best_actions, 1)
  } else {
    action = best_actions
  }
  return (action)
}
```

```r
EpsilonGreedyPolicy <- function(x, y, epsilon){

  # Get an epsilon-greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   epsilon: probability of acting randomly.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.
  if (epsilon >= runif(1)){
    action = sample(1:4, 1)
  } else {
    action = GreedyPolicy(x, y)
  }

  return (action)
}

transition_model <- function(x, y, action, beta){

  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  delta <- sample(-1:1, size = 1, prob = c(0.5*beta,1-beta,0.5*beta))
  final_action <- ((action + delta + 3) %% 4) + 1
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}

q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0){

  #Initialize
  current_state = start_state
  episode_correction = 0
  reward = 0

  repeat{
    # Current state
```

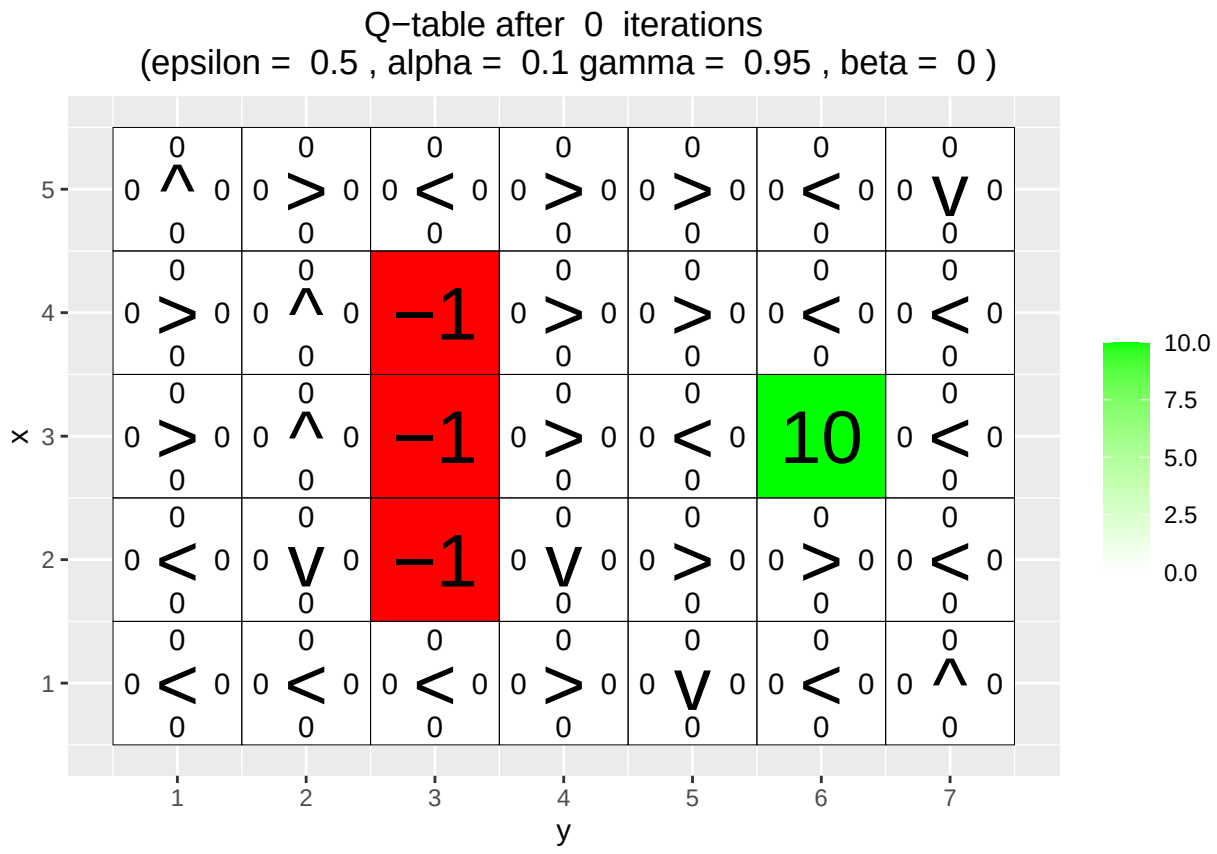
```
    x = current_state[1]
    y = current_state[2]

    # Action
    action = EpsilonGreedyPolicy(x, y, epsilon)

    # Update state
    next_state = transition_model(x, y, action, beta)
    next_x = next_state[1]
    next_y = next_state[2]

    # Reward
    reward = reward_map[next_x, next_y]

    # New max Q value
    max_q_next = max(q_table[next_x, next_y, ])

    # Correction
    correction = reward + gamma * max_q_next - q_table[x, y, action]

    # Update q_table based on correction
    q_table[x, y, action] <<- q_table[x, y, action] + alpha * correction

    # Accumulate corrections
    episode_correction = episode_correction + correction

    # Update state
    current_state = next_state

    # End the episode if a terminal state (non-zero reward) is reached
    if (reward != 0) {
      break
    }
  }

  return (c(reward, episode_correction))
}
```

3.2

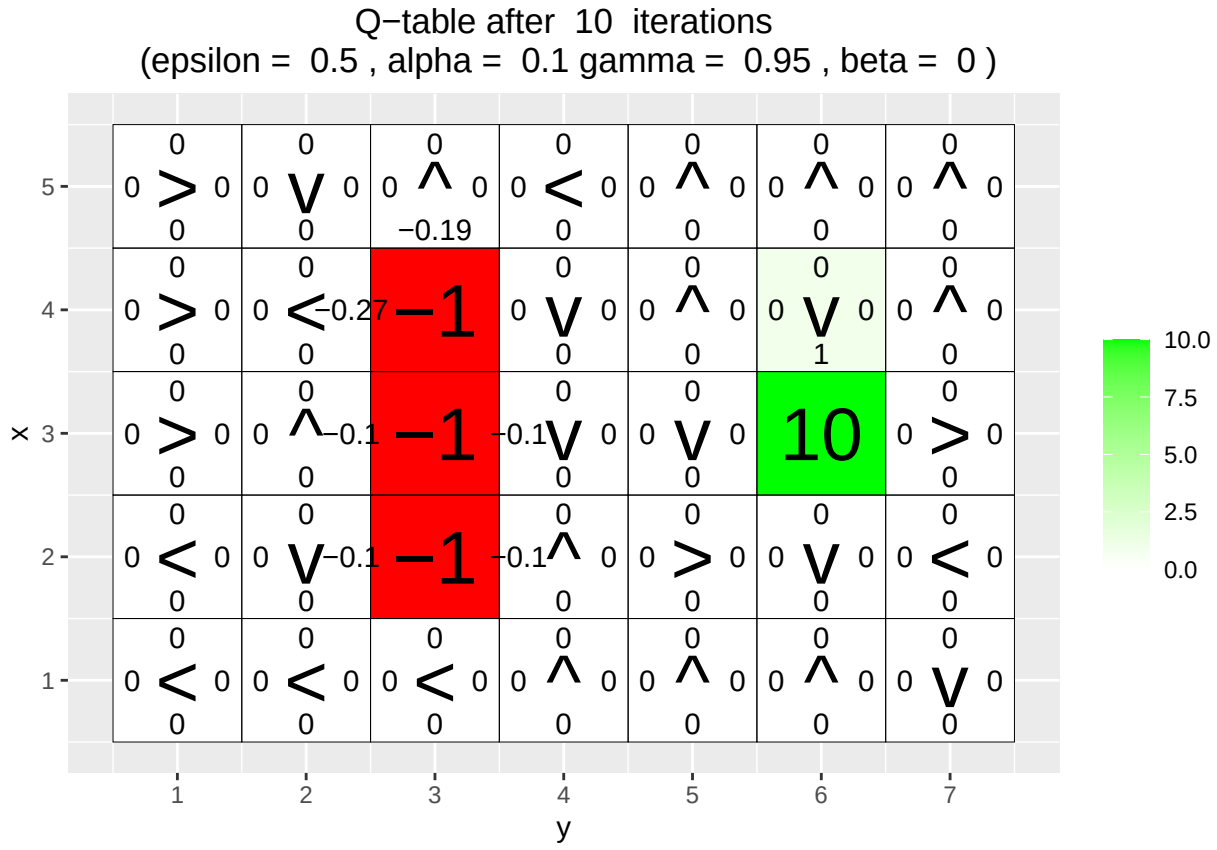
- Run 10000 episodes of Q-learning with $\epsilon = 0.5$, $\beta = 0$, $\alpha = 0.1$, and $\gamma = 0.95$. Visualize the results at various stages (after 10, 100, 1000, and 10000 episodes).

```
# Environment A (learning)
set.seed(1234)
H <- 5
W <- 7

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[3,6] <- 10
reward_map[2:4,3] <- -1
```

```
q_table <- array(0,dim = c(H,W,4))

vis_environment()
```

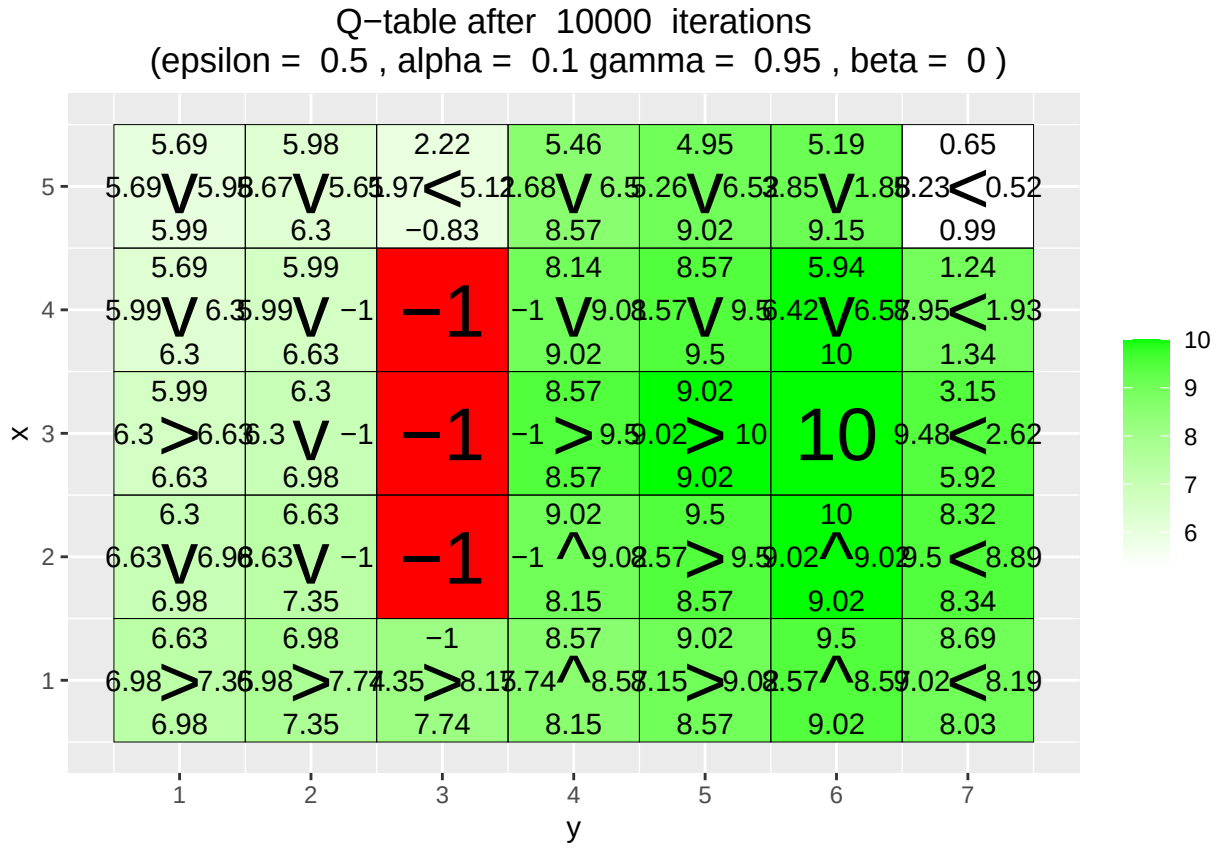



```
for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

  if(any(i==c(10,100,1000,10000)))
    vis_environment(i)
}
```


Q−table after 10 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0
−0.19
−0.27
−0.1
−0.1
−0.1
−0.1
1
10


Q−table after 100 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


Q−table after 1000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0



Q: What has the agent learned after the first 10 episodes ? A: To not enter(4, 3)

Q: Is the final greedy policy (after 10000 episodes) optimal for all states, i.e. not only for the initial state ? Why / Why not ? A: No. For example, the action policy for state (5, 3) is to go left, but the optimal action is right. This is due to the agent not having had the chance of exploring this state enough times.

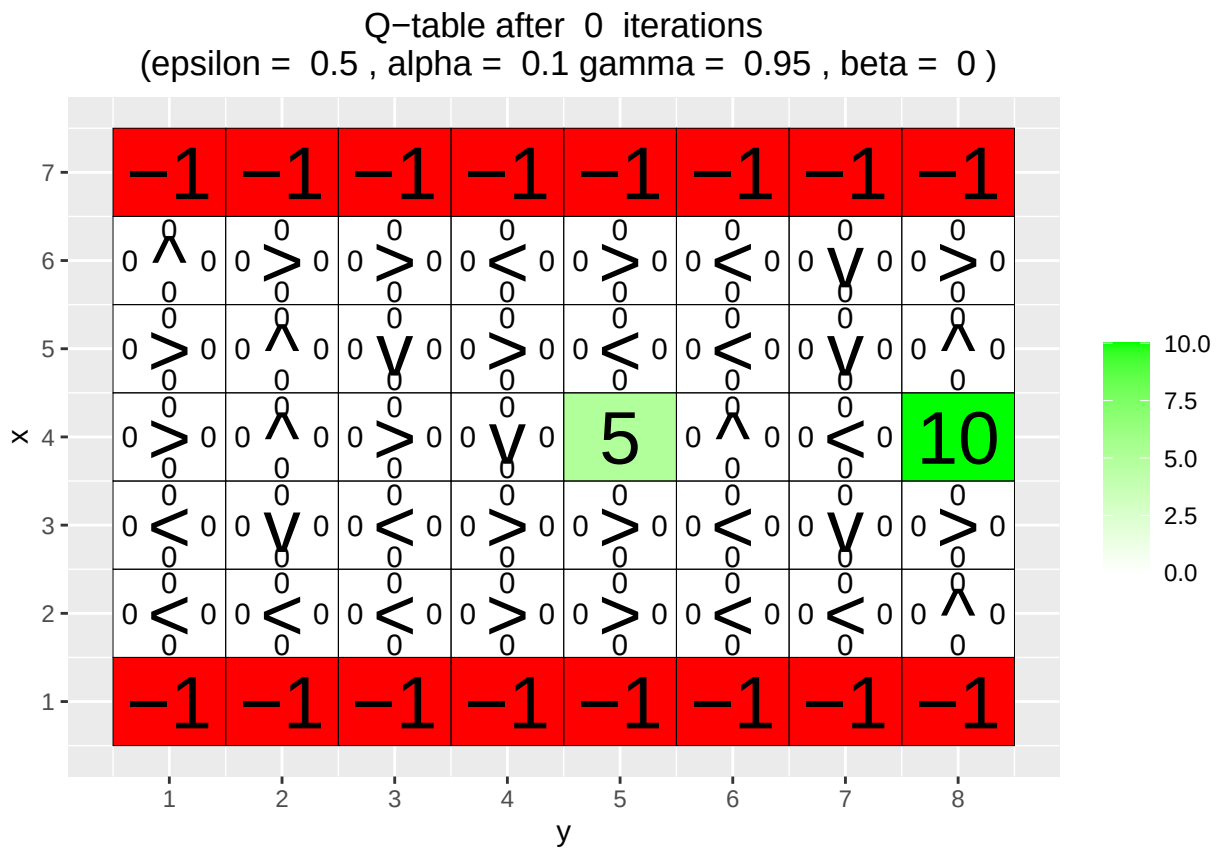
Q: Do the learned values in the Q-table reflect the fact that there are multiple paths (above and below the negative rewards) to get to the positive reward ? If not, what could be done to make it happen ? A: No, it always tries to go below. We could either enforce a greater epsilon, or change start_state to be further up (e.g (1, 5))

3.3

Environment B: - Investigate the effects of different values of ϵ and γ by running 30000 episodes of Q-learning with various configurations.

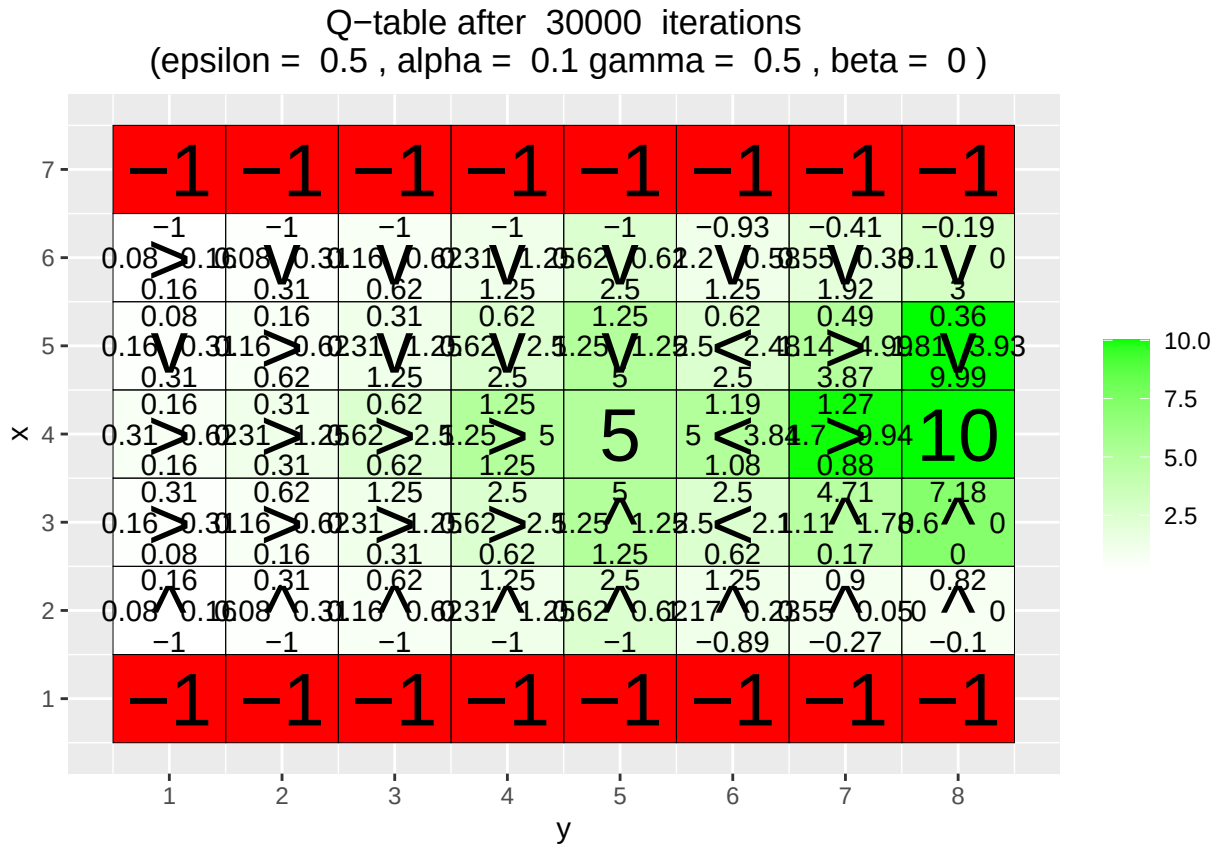
```
# Environment B (the effect of epsilon and gamma)
set.seed(1234)

H <- 7
W <- 8

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,] <- -1
reward_map[7,] <- -1
reward_map[4,5] <- 5
```

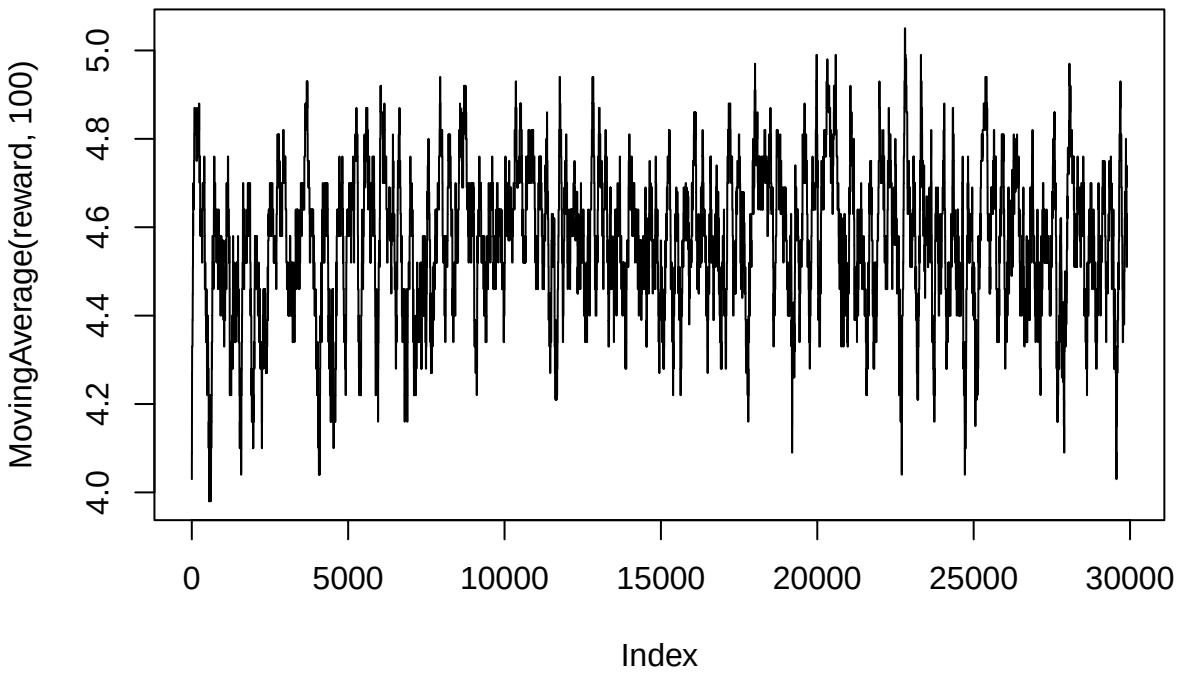
```
reward_map[4,8] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()
```

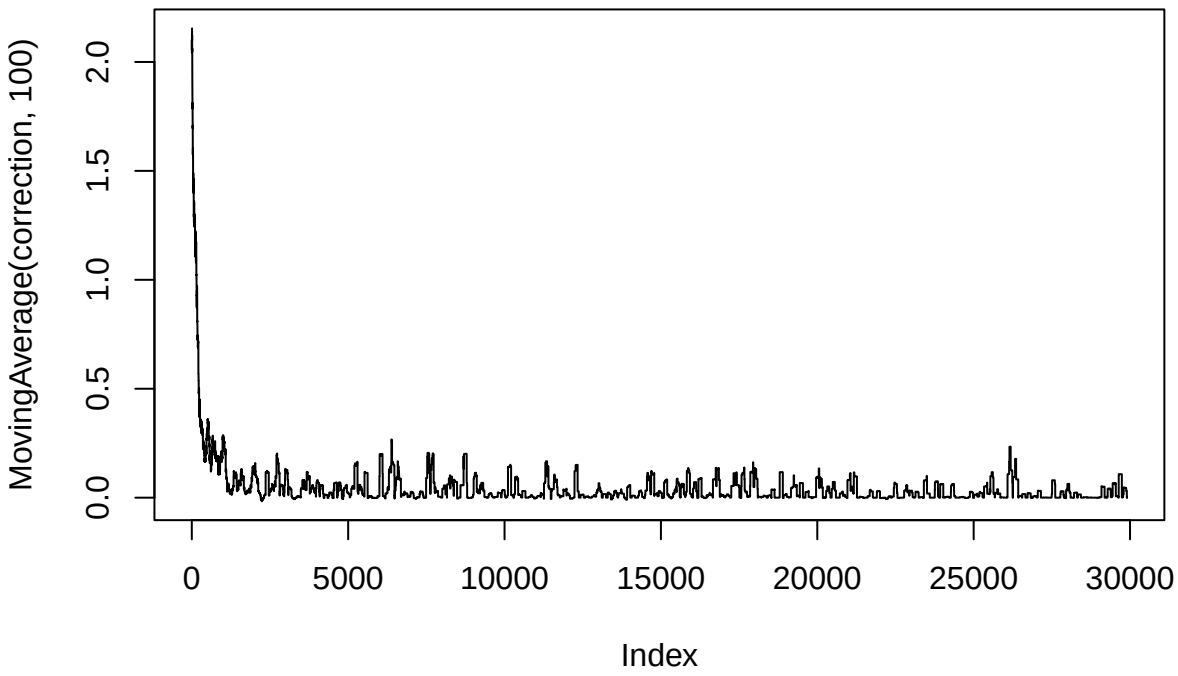


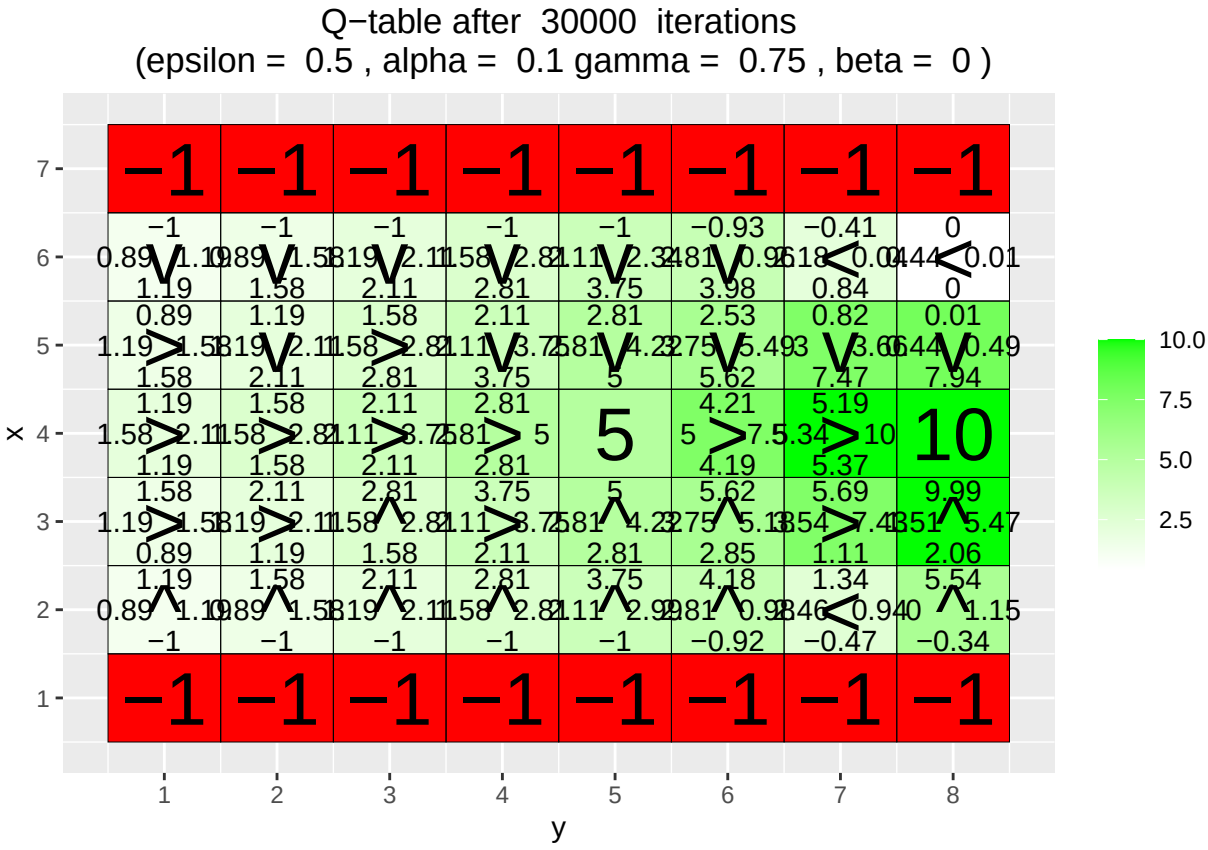

```
MovingAverage <- function(x, n){

  cx <- c(0,cumsum(x))
  rsum <- (cx[(n+1):length(cx)] - cx[1:(length(cx) - n)]) / n

  return (rsum)
}

for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }
```

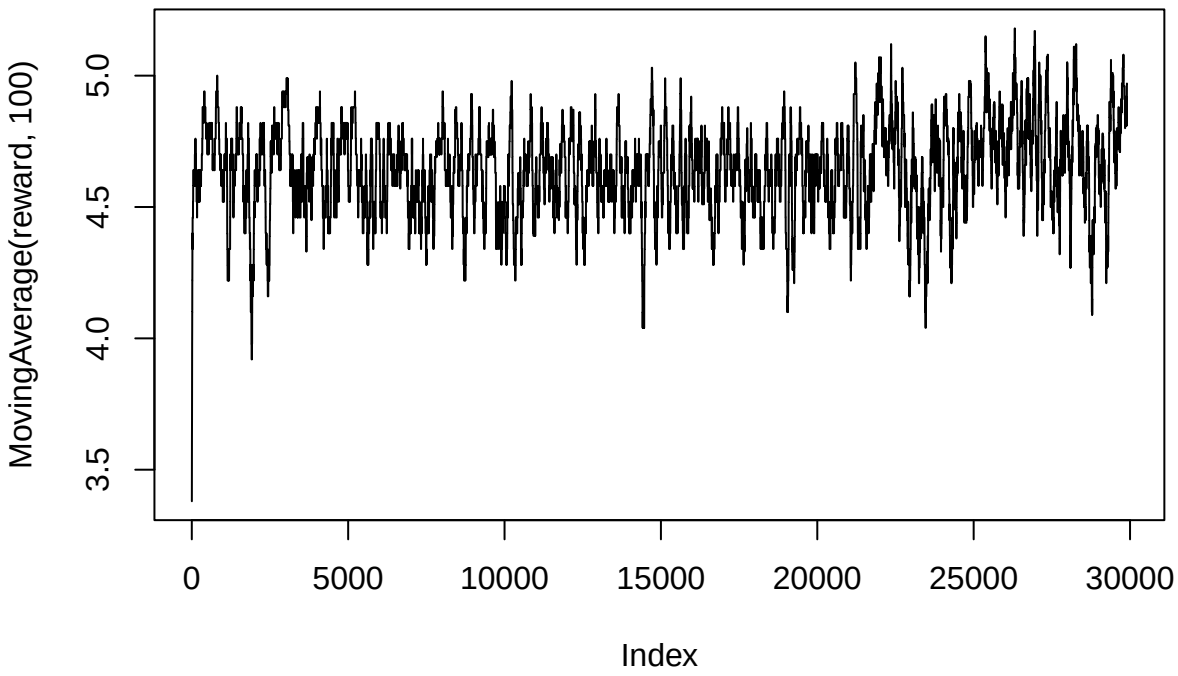
```
  vis_environment(i, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}
```

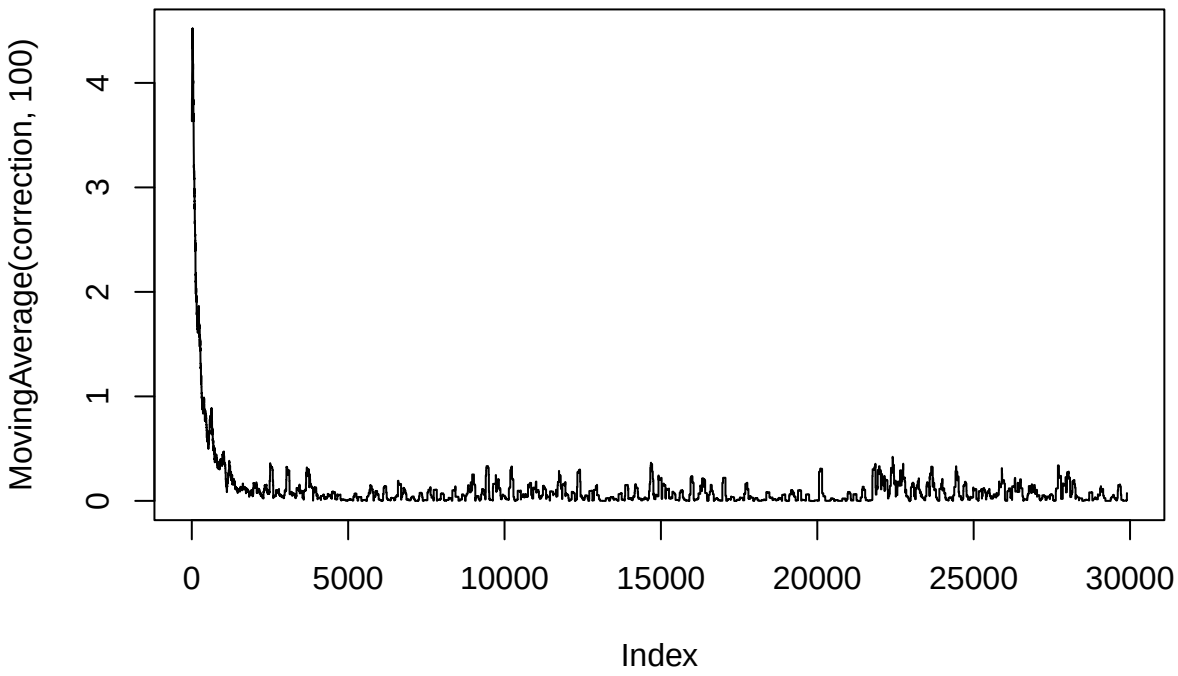


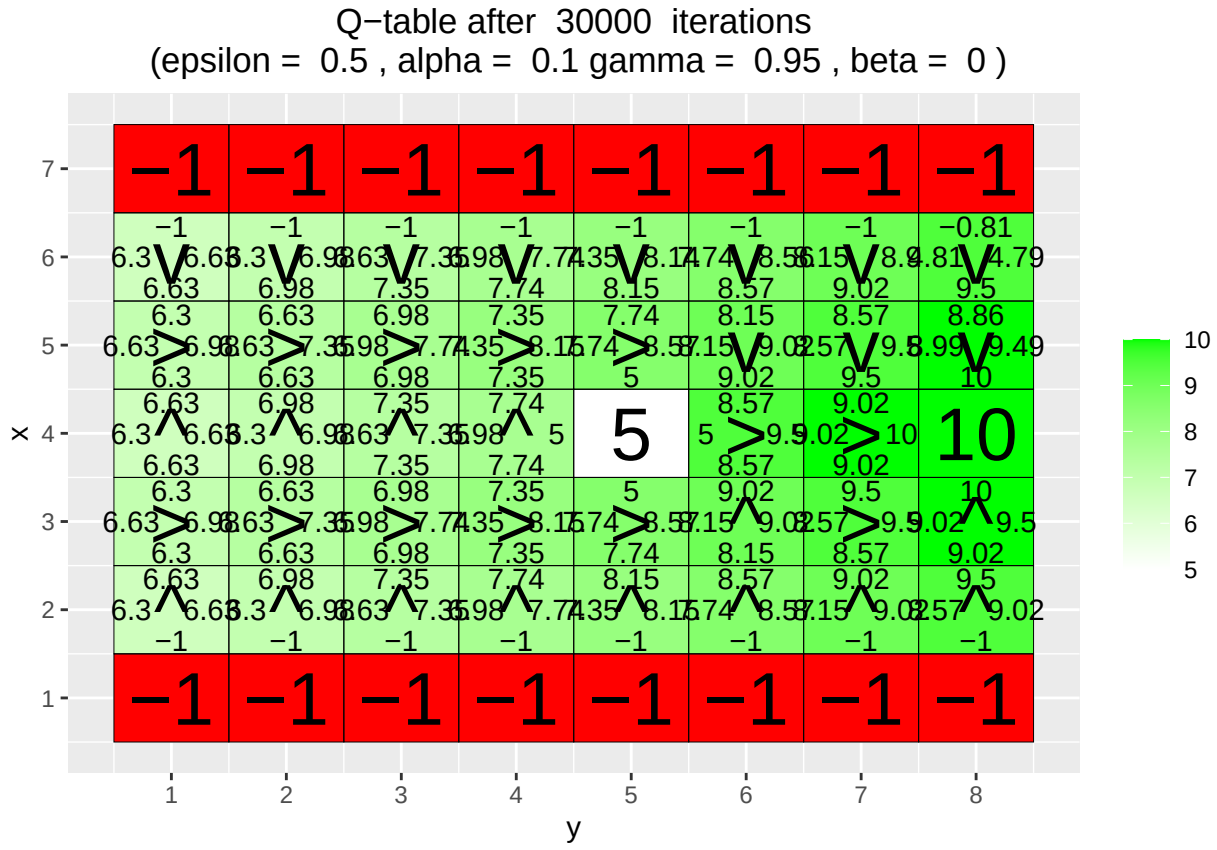

MovingAverage(reward, 100)
4.0
4.2
4.4
4.6
4.8
5.0
0
5000
10000
15000
20000
25000
30000
Index

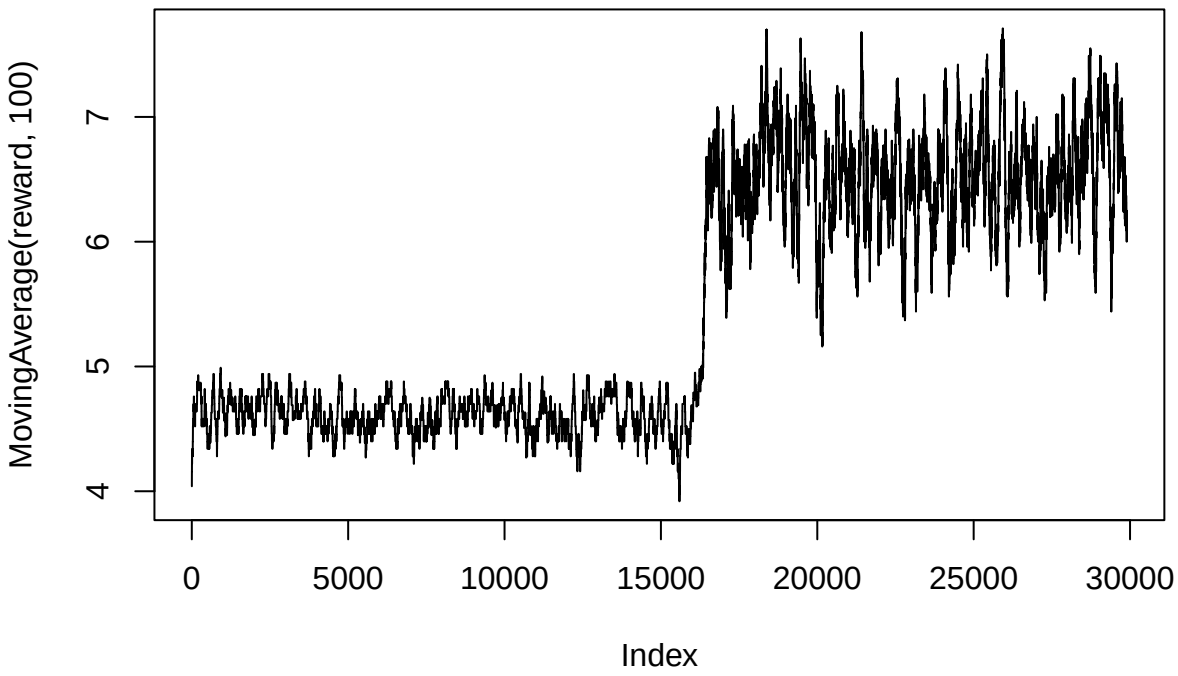

MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
0
5000
10000
15000
20000
25000
30000
Index

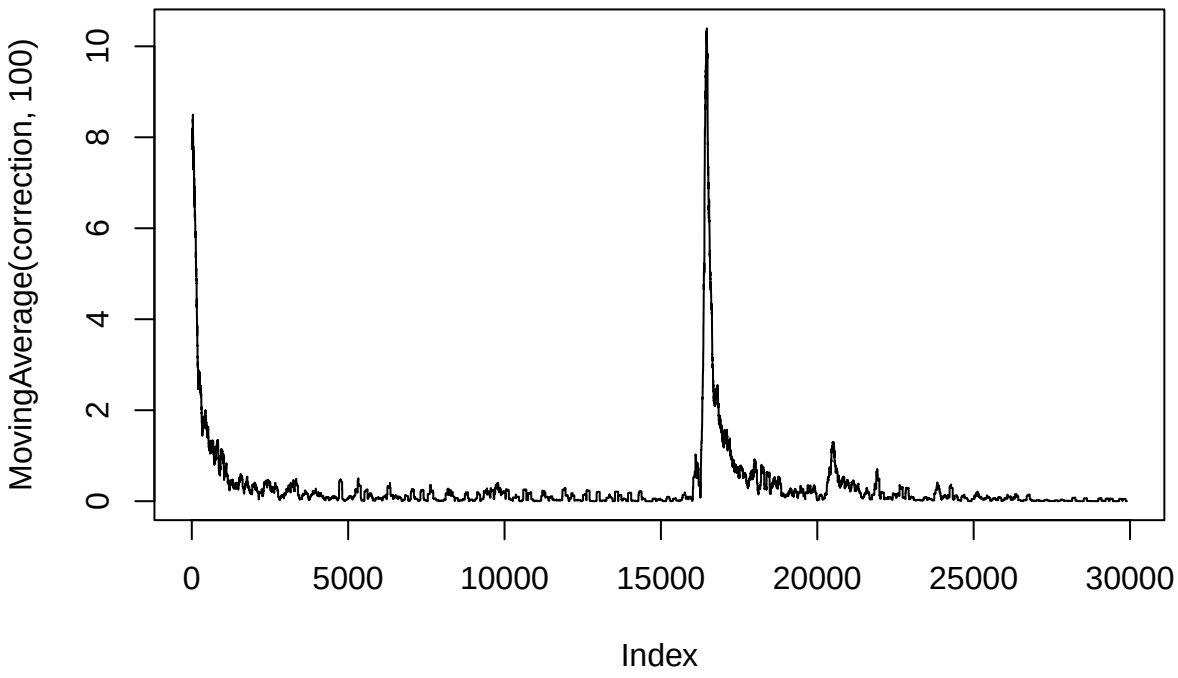

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5


MovingAverage(reward, 100)
3.5
4.0
4.5
5.0
0
5000
10000
15000
20000
25000
30000
Index

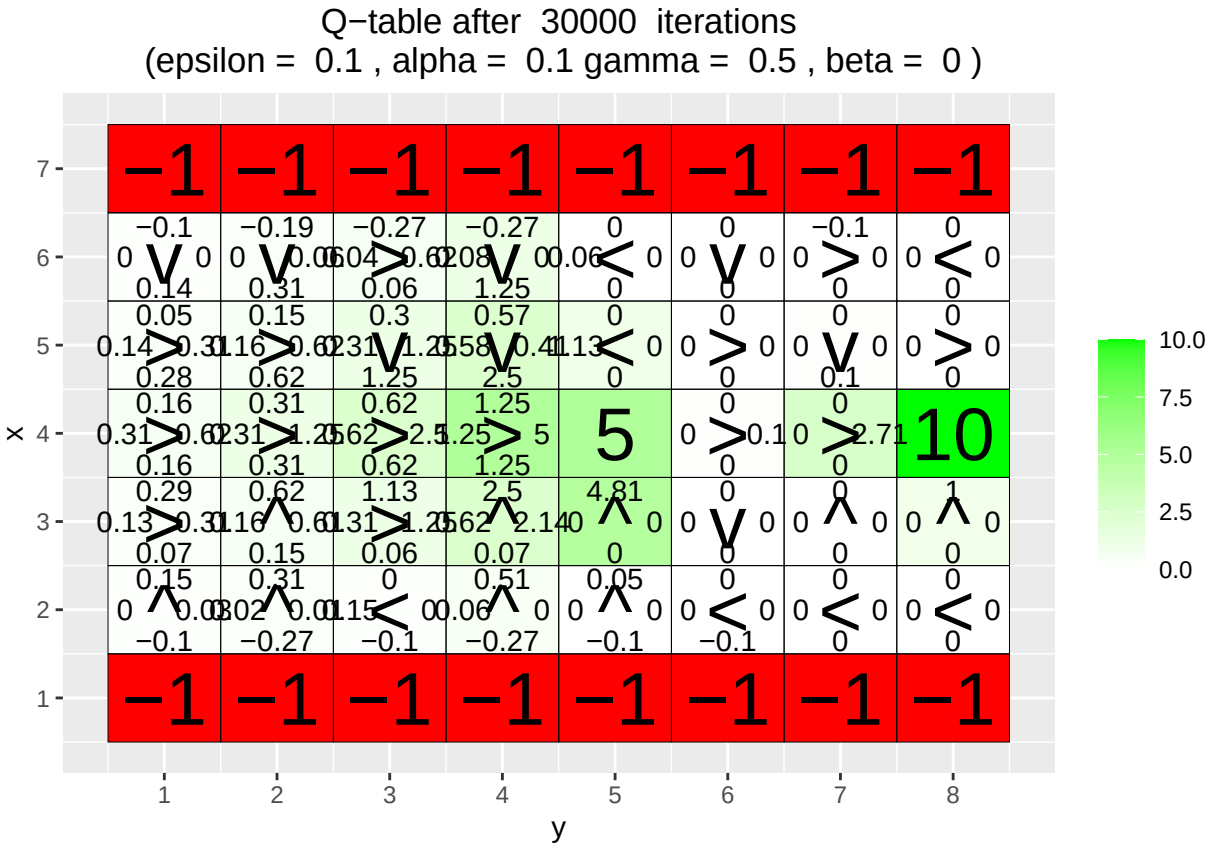

MovingAverage(correction, 100)
0
1
2
3
4
0
5000
10000
15000
20000
25000
30000
Index

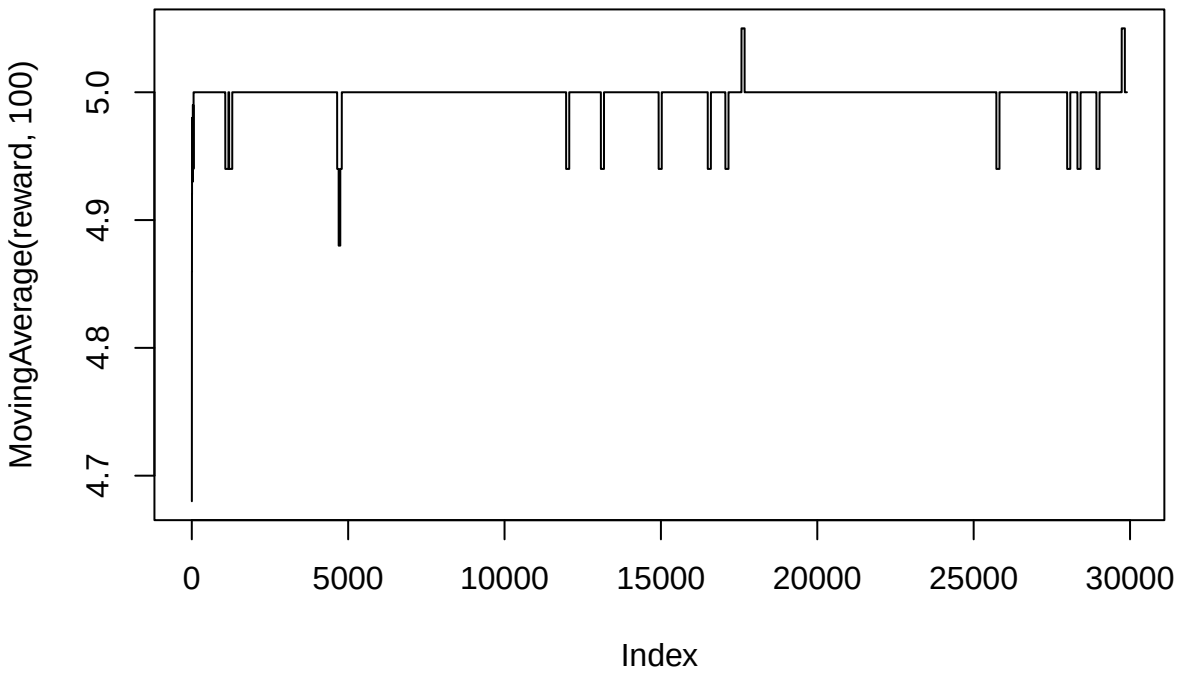

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10
9
8
7
6
5

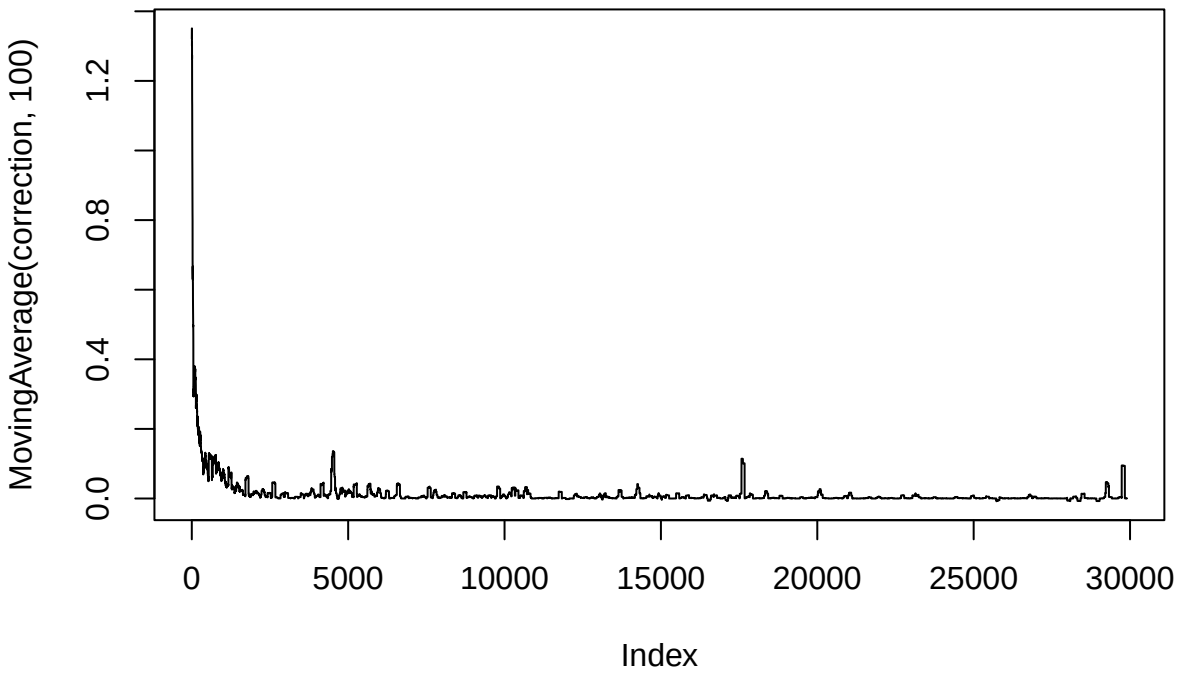

MovingAverage(reward, 100)
4
5
6
7
0
5000
10000
15000
20000
25000
30000
Index

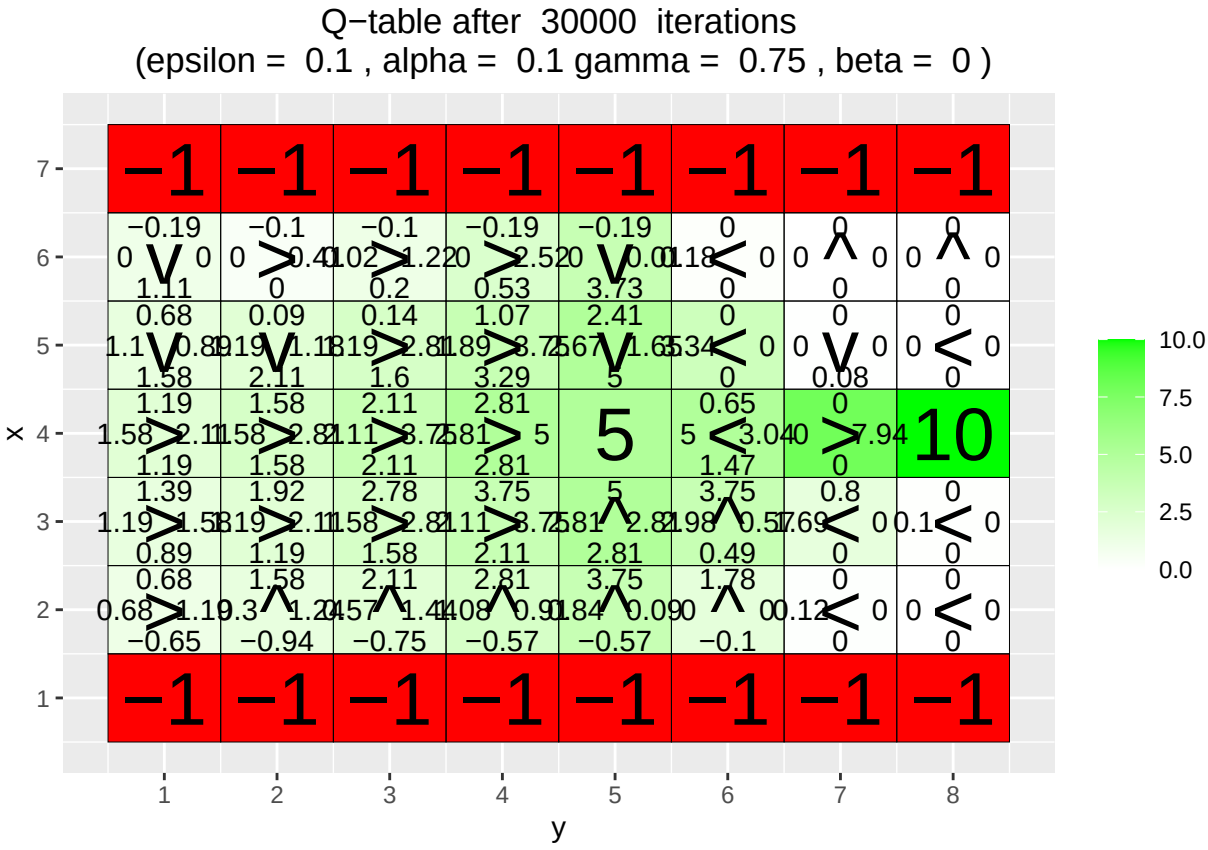


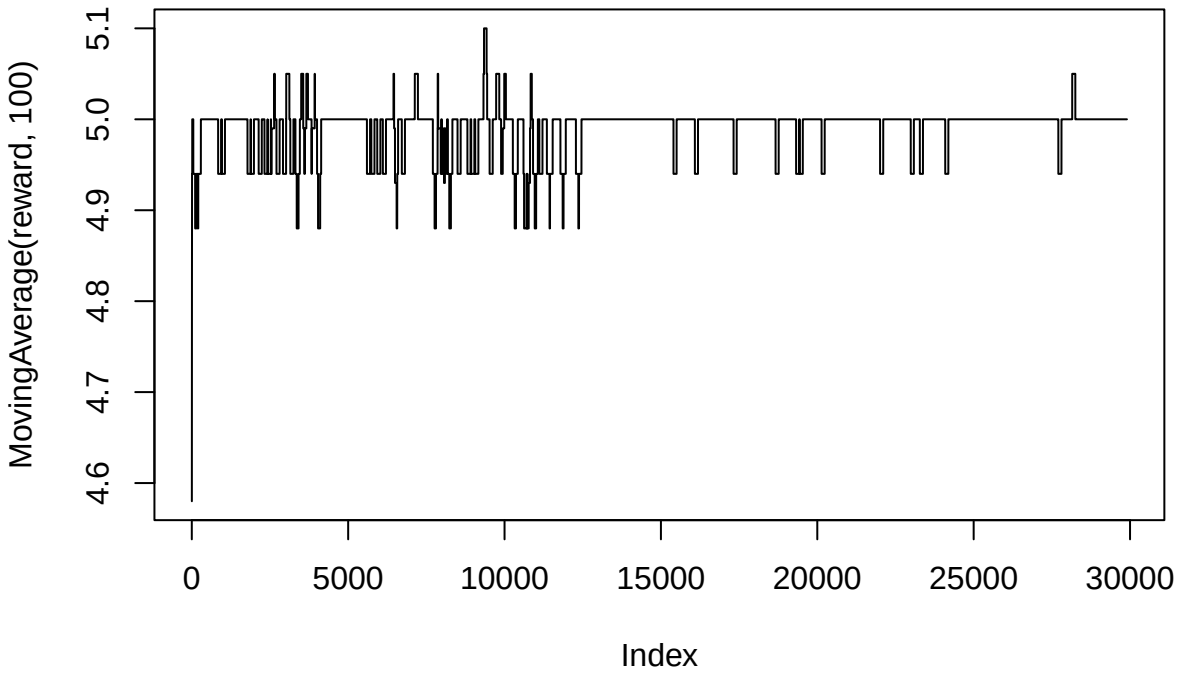

```
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(epsilon = 0.1, gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, epsilon = 0.1, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}
```

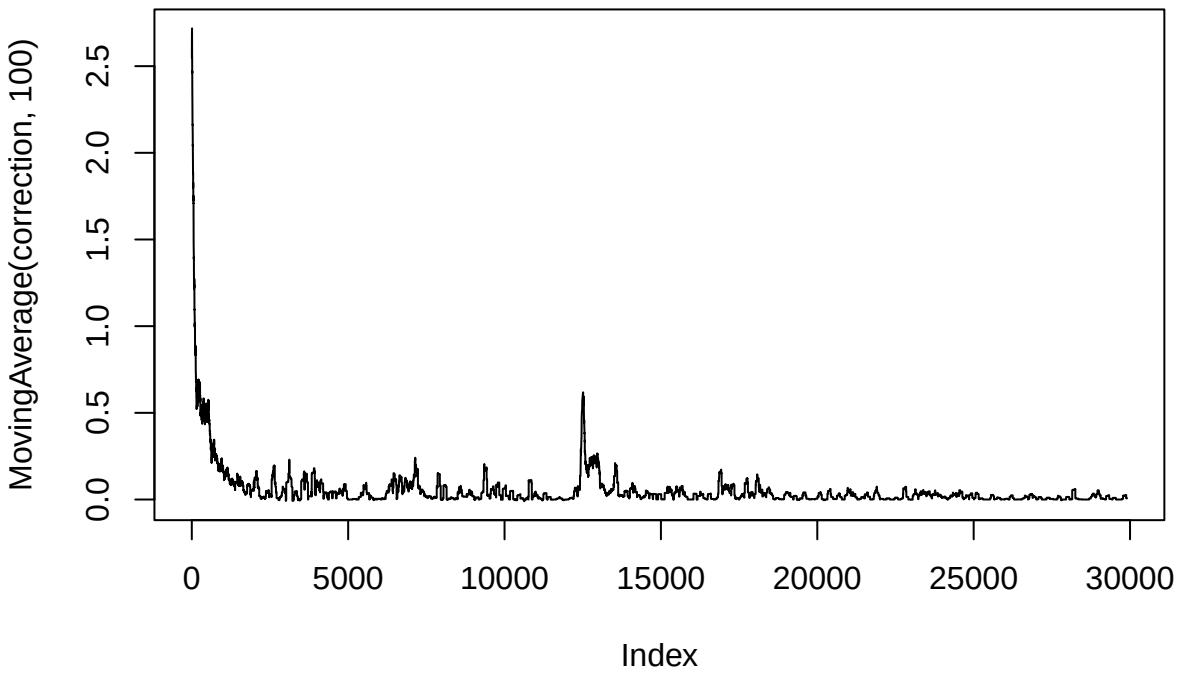

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0

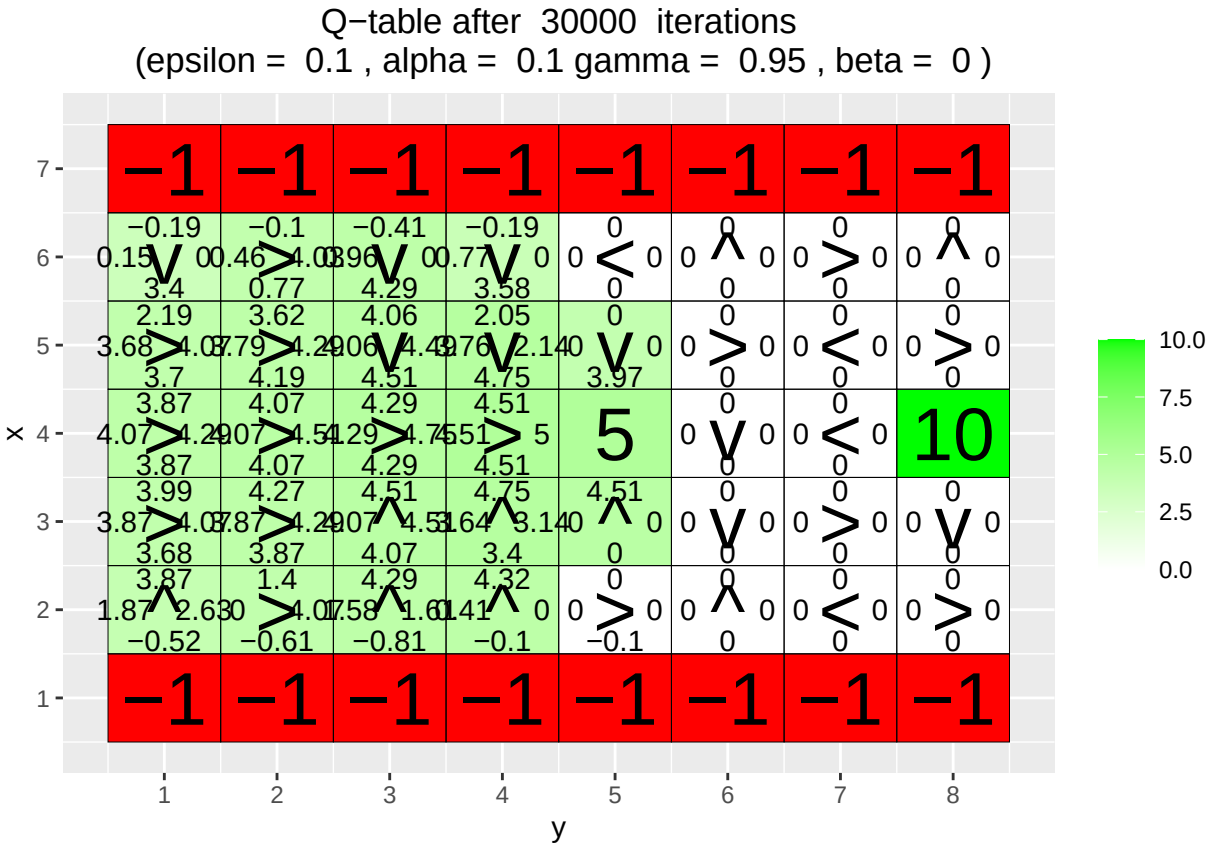

MovingAverage(reward, 100)
4.7
4.8
4.9
5.0
0
5000
10000
15000
20000
25000
30000
Index

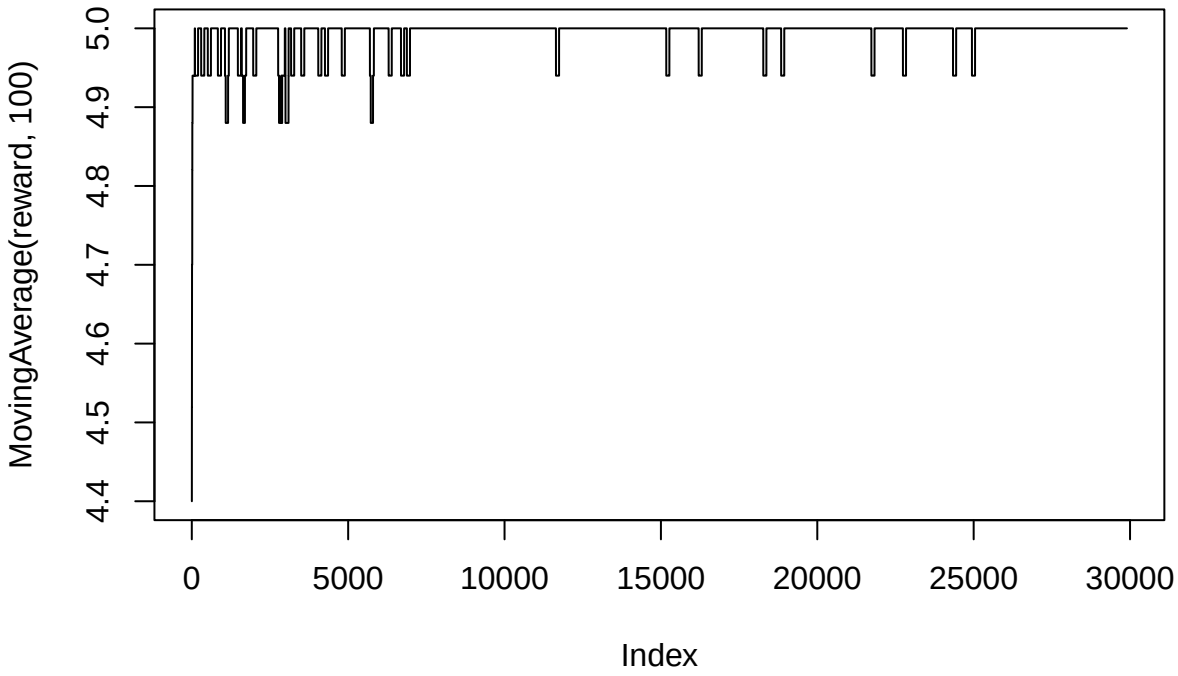

MovingAverage(correction, 100)
0.0
0.4
0.8
1.2
0
5000
10000
15000
20000
25000
30000
Index

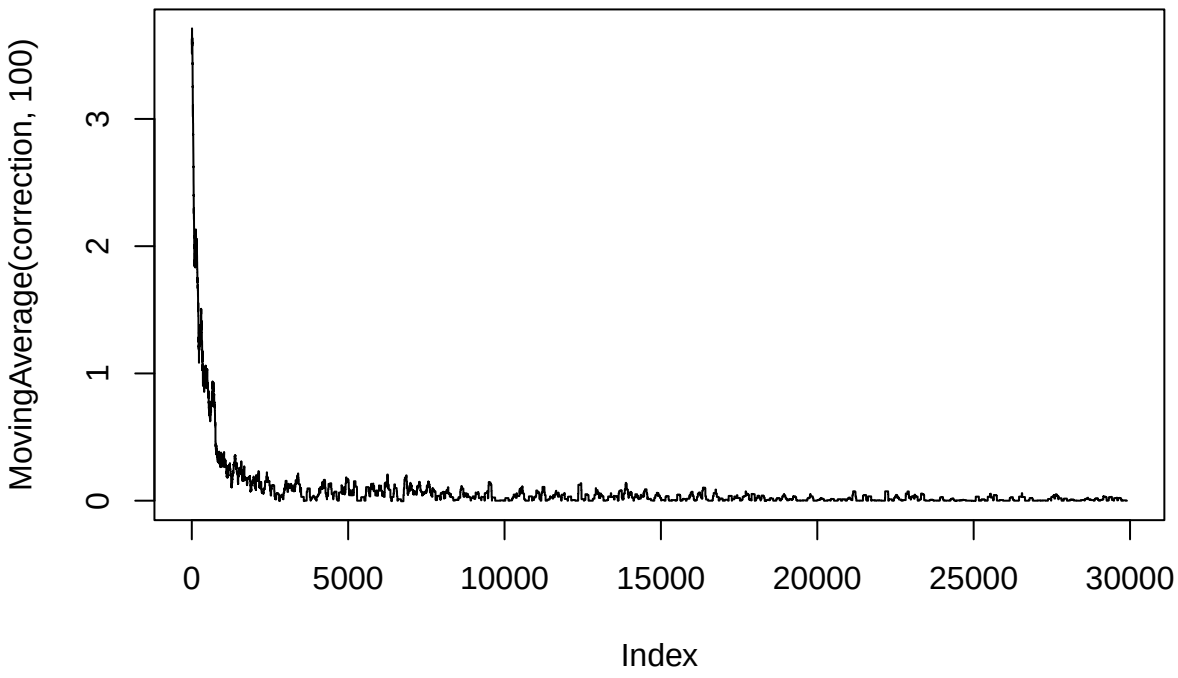

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
4.6
4.7
4.8
4.9
5.0
5.1
0
5000
10000
15000
20000
25000
30000
Index


MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
2.5
0
5000
10000
15000
20000
25000
30000
Index


Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
4.4
4.5
4.6
4.7
4.8
4.9
5.0
0
5000
10000
15000
20000
25000
30000
Index

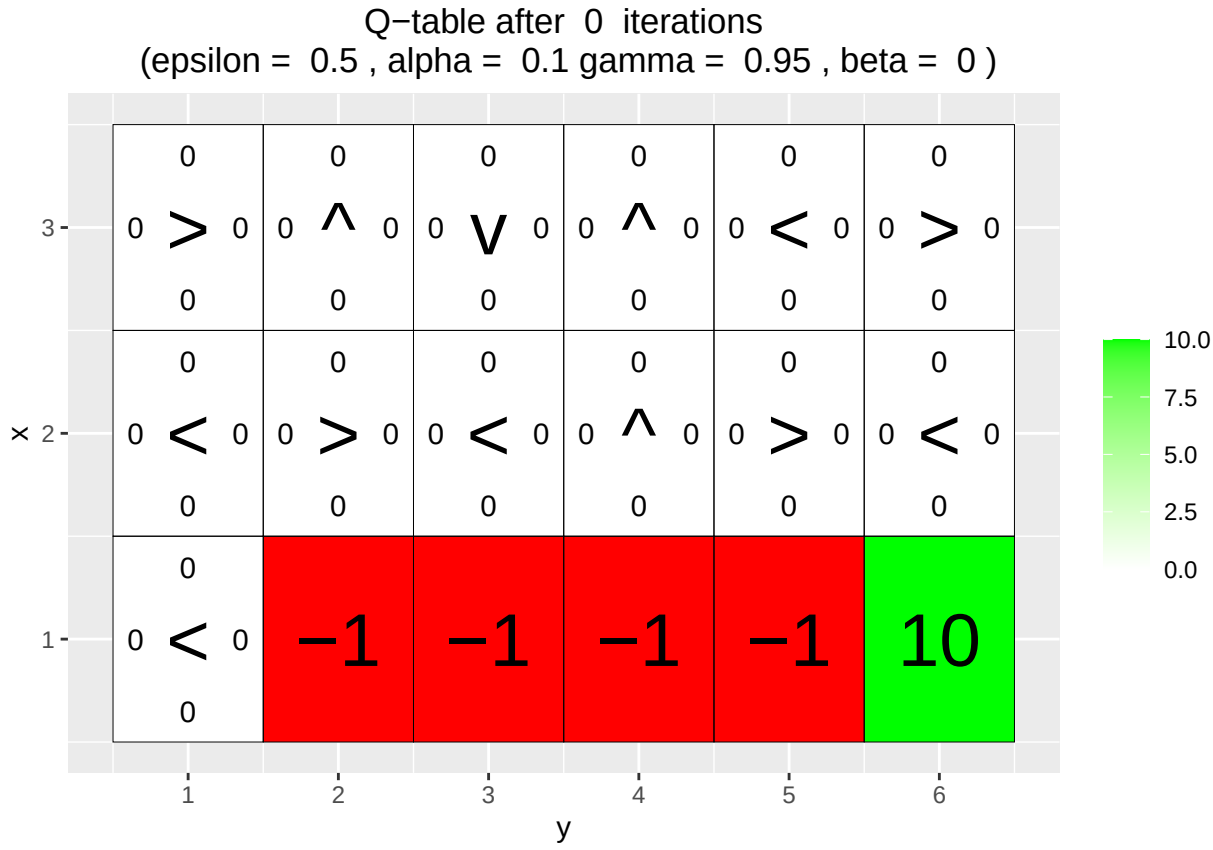


Q: Investigate how the ϵ and γ parameters affect the learned policy A: A high γ changes the action policy to be more prone going towards 10 over 5. A low ϵ leads to the agent not exploring certain states.

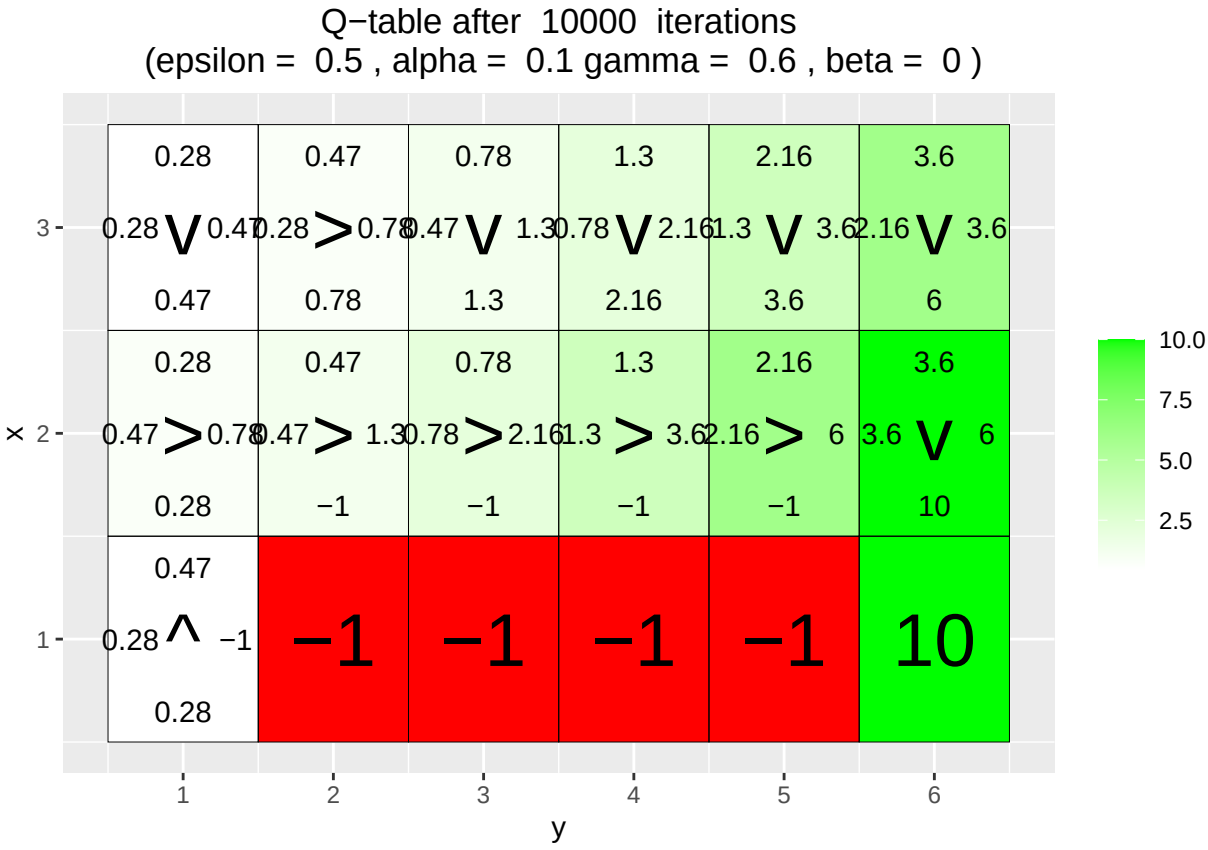
3.4

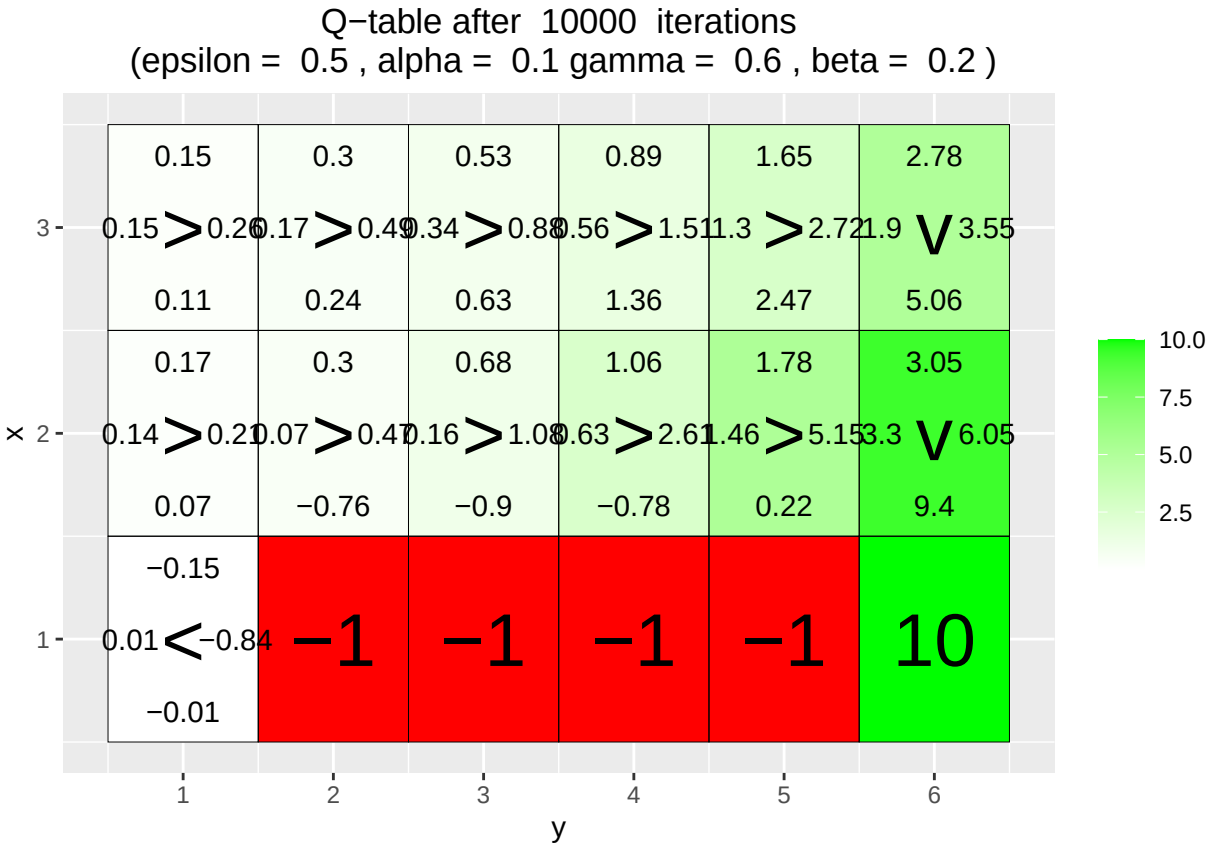
Environment C: - Investigate how the β parameter (which controls the probability of slipping) affects the learned policy by running 10000 episodes with different values of β.

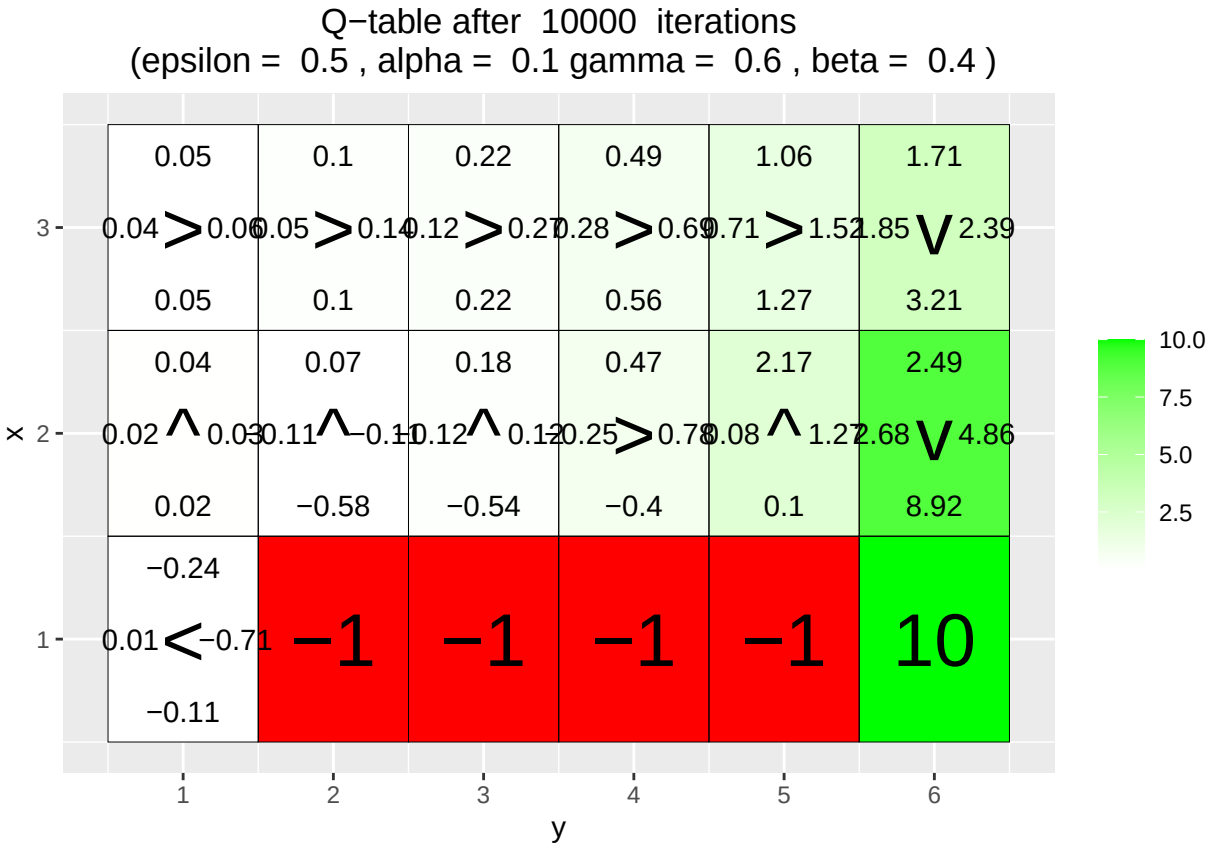
```
# Environment C (the effect of beta).
set.seed(1234)

H <- 3
W <- 6

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,2:5] <- -1
reward_map[1,6] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()
```

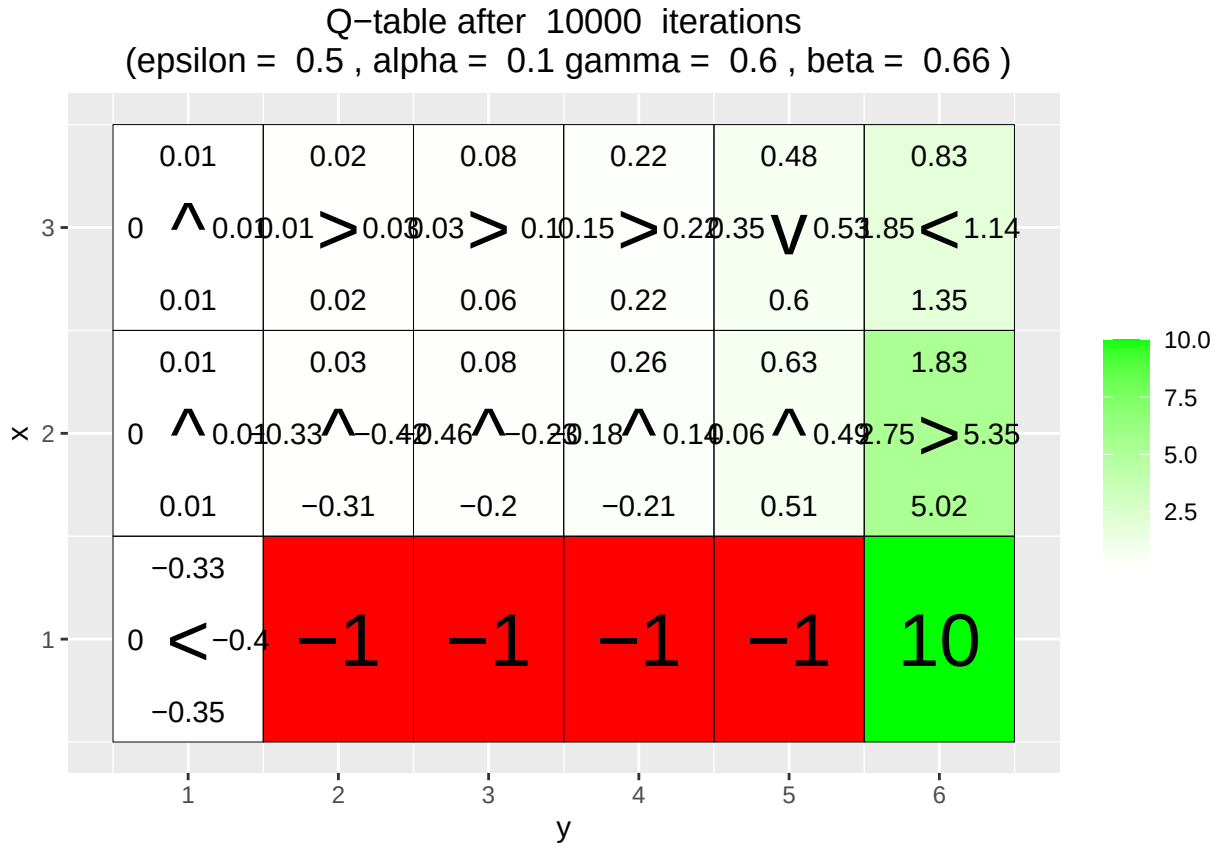



```
for(j in c(0,0.2,0.4,0.66)){
  q_table <- array(0,dim = c(H,W,4))

  for(i in 1:10000)
    foo <- q_learning(gamma = 0.6, beta = j, start_state = c(1,1))

  vis_environment(i, gamma = 0.6, beta = j)
}
```


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0)
0.28
0.47
0.78
1.3
2.16
3.6
0.28 V 0.47
0.28 > 0.78
0.47 V 1.3
0.78 V 2.16
1.3 V 3.6
2.16 V 3.6
0.47
0.78
1.3
2.16
3.6
6
0.28
0.47
0.78
1.3
2.16
3.6
0.47 > 0.78
0.47 > 1.3
0.78 > 2.16
1.3 > 3.6
2.16 > 6
3.6 V 6
0.28
−1
−1
−1
−1
10
0.47
0.28 ^ −1
−1
−1
−1
−1
10
0.28
3
2
1
x
1
2
3
4
5
6
y
10.0
7.5
5.0
2.5


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)
0.15
0.3
0.53
0.89
1.65
2.78
0.15 > 0.26
0.17 > 0.49
0.34 > 0.88
0.56 > 1.51
1.3 > 2.72
1.9 V 3.55
0.11
0.24
0.63
1.36
2.47
5.06
0.17
0.3
0.68
1.06
1.78
3.05
0.14 > 0.21
0.07 > 0.47
0.16 > 1.08
0.63 > 2.61
1.46 > 5.15
3.3 V 6.05
0.07
−0.76
−0.9
−0.78
0.22
9.4
−0.15
0.01 < −0.84
−1
−1
−1
−1
10
−0.01
x
3
2
1
1
2
3
4
5
6
y
10.0
7.5
5.0
2.5


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.4)
0.05
0.1
0.22
0.49
1.06
1.71
0.04 > 0.06
0.05 > 0.14
0.12 > 0.27
0.28 > 0.69
0.71 > 1.52
1.85 V 2.39
0.05
0.1
0.22
0.56
1.27
3.21
0.04
0.07
0.18
0.47
2.17
2.49
0.02 ^ 0.03
0.11 ^ 0.11
0.12 ^ 0.12
0.25 > 0.78
0.08 ^ 1.27
2.68 V 4.86
0.02
−0.58
−0.54
−0.4
0.1
8.92
−0.24
0.01 < −0.71
−1
−1
−1
−1
10
−0.11
x
y
1
2
3
1
2
3
4
5
6
10.0
7.5
5.0
2.5



Q: Investigate how the β parameter affects the learned policy. A: Having a higher β worsen the model. This is due to it becoming more difficult for the agent to understand the task. Choosing an optimal action can lead to a negative reward.

3.6

Environment D: - Evaluate the agent's performance after training on eight goal positions and validating on the remaining eight. Answer questions regarding the learned policy's effectiveness.

https://github.com/STIMALiU/AdvMLCourse/blob/master/ReinforcementLearning/RL_Lab2_Colab.ipynb

Q: Has the agent learned a good policy? Why / Why not ? A: Yes, it is performing well finding the optimal policy. It seems to have learnt a policy taking to account the goal target.

Q: Could you have used the Q-learning algorithm to solve this task ? A: In theory yes, if it got to explore all combinations of states and goals, but Q-learning does not generalize.

2.7

Environment E: - Analyze the agent's policy when trained on goals from the top row and validated on goals from the lower rows. Compare the results with Environment D.

In this task, the goals for training are all from the top row of the grid. The validation goals are three positions from the rows below.

Q: Has the agent learned a good policy? Why / Why not ? A: No, the agent tends to want to move upward, since that's what had the best return during training.

Q: If the results obtained for environments D and E differ, explain why. A: The agent in E is only trained on goals to the top, causing its policy to perform bad for goals at the bottom.

LAB4

4.1 Implementing GP Regression

This first exercise will have you writing your own code for the Gaussian process regression model:

$$y = f(x) + \epsilon \quad \text{with} \quad \epsilon \sim \mathcal{N}(0, \sigma_n^2) \quad \text{and} \quad f \sim \mathcal{GP}(0, k(x, x'))$$

You must implement **Algorithm 2.1** on page 19 of Rasmussen and Williams' book. The algorithm uses the Cholesky decomposition (`chol` in R) to attain numerical stability. Note that L in the algorithm is a lower triangular matrix, whereas the R function returns an upper triangular matrix. So, you need to transpose the output of the R function.

In the algorithm, the notation A/b means the vector x that solves the equation $Ax = b$ (see p. xvii in the book). This is implemented in R with the help of the function `solve`.

4.1.1

Write your own code for simulating from the posterior distribution of f using the squared exponential kernel. The function (name it `posteriorGP`) should return a vector with the posterior mean and variance of f, both evaluated at a set of x-values (X^*). You can assume that the prior mean of f is zero for all x. The function should have the following inputs:

- **X**: Vector of training inputs.
- **y**: Vector of training targets/outputs.
- **XStar**: Vector of inputs where the posterior distribution is evaluated, i.e., X^*.
- **sigmaNoise**: Noise standard deviation σ_n.
- **k**: Covariance function or kernel. That is, the kernel should be a separate function (see the file *GaussianProcesses.R* on the course web page).

```
# Covariance function
SquaredExpKernel <- function(x1,x2,sigmaF=1,l=3){
  n1 <- length(x1)
  n2 <- length(x2)
  k <- matrix(NA,n1,n2)
  for (i in 1:n2){
    k[,i] <- sigmaF**2*exp(-0.5*( (x1-x2[i])/l)**2 )
  }
  return(k)
}

posteriorGP = function(x, y, XStar, sigmaNoise, k, ...){
  K = k(x, x, ...)
  L = t(chol(K + diag(sigmaNoise**2, length(x))))
  alpha = solve(t(L), solve(L,y))
```

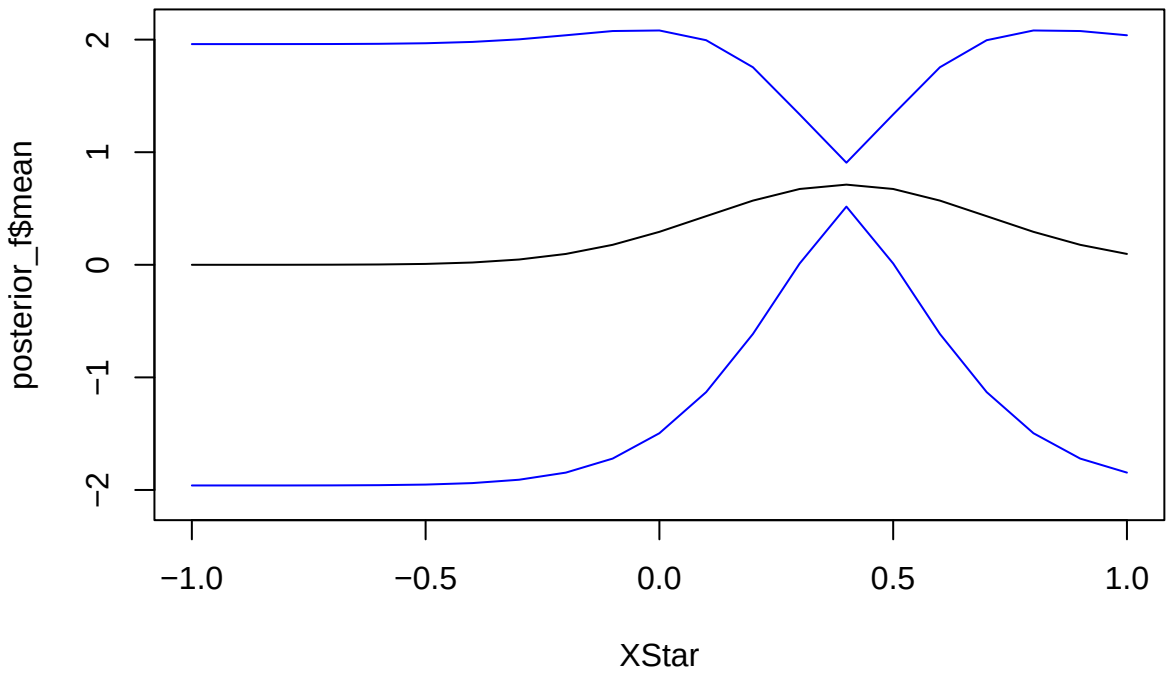
```
  kStar = k(x, XStar, ...)
  predictive_mean = t(kStar)%*%alpha

  v = solve(L, kStar)
  predictive_variance = k(XStar, XStar, ...) - t(v)%*%v

  return(list(mean = predictive_mean, variance = predictive_variance))
}
```

4.1.2

Now, let the prior hyperparameters be $\sigma_f = 1$ and $\ell = 0.3$. Update this prior with a single observation: $(x, y) = (0.4, 0.719)$. Assume that $\sigma_n = 0.1$.

- Plot the posterior mean of f over the interval $x \in [-1, 1]$.
- Also plot 95% probability (pointwise) bands for f.

```
sigmaF = 1
l = 0.3
x = 0.4
y = 0.719
sigmaNoise = 0.1
XStar = seq(-1, 1, 0.1)

posterior_f = posteriorGP(x, y, XStar, sigmaNoise, k=SquaredExpKernel, sigmaF, l)
plot(XStar, posterior_f$mean, type = "l", ylim = c(-2.1, 2.1))
lines(XStar, posterior_f$mean - 1.96*sqrt(diag(posterior_f$variance)), col = "blue")
lines(XStar, posterior_f$mean + 1.96*sqrt(diag(posterior_f$variance)), col = "blue")
```



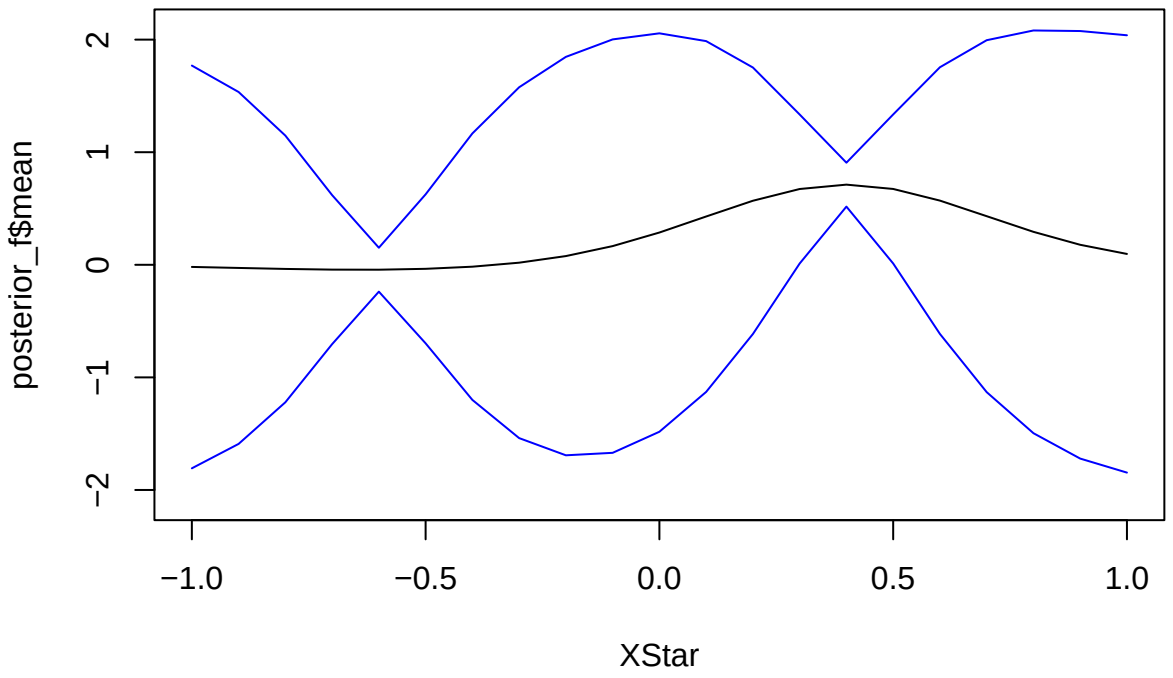
4.1.3

Update your posterior from (2) with another observation: $(x, y) = (-0.6, -0.044)$.

- Plot the posterior mean of f over the interval $x \in [-1, 1]$.
- Also plot 95% probability (pointwise) bands for f.

Hint: Updating the posterior after one observation with a new observation gives the same result as updating the prior directly with the two observations.

```
sigmaF = 1
l = 0.3
x = c(0.4, -0.6)
y = c(0.719, -0.044)
sigmaNoise = 0.1
XStar = seq(-1, 1, 0.1)

posterior_f = posteriorGP(x, y, XStar, sigmaNoise, k=SquaredExpKernel, sigmaF, l)
plot(XStar, posterior_f$mean, type = "l", ylim = c(-2.1, 2.1))
lines(XStar, posterior_f$mean - 1.96*sqrt(diag(posterior_f$variance)), col = "blue")
lines(XStar, posterior_f$mean + 1.96*sqrt(diag(posterior_f$variance)), col = "blue")
```



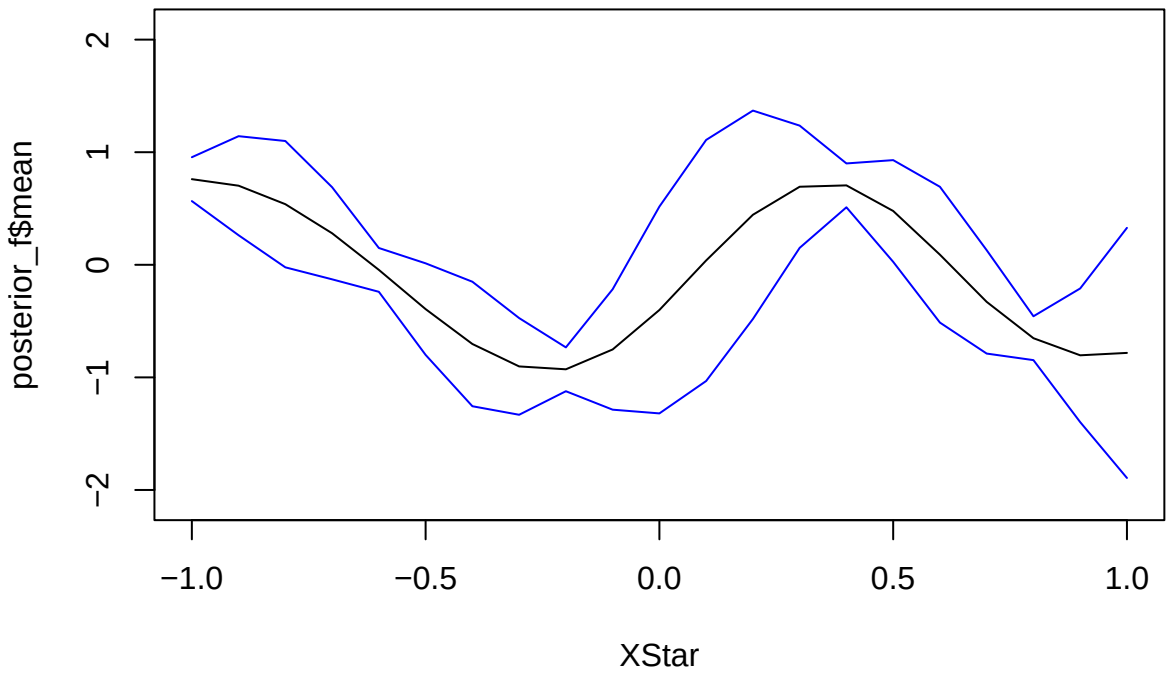
4.1.4

Compute the posterior distribution of f using all the five data points in the table below (note that the two previous observations are included in the table).

x	-1.0	-0.6	-0.2	0.4	0.8
y	0.768	-0.044	-0.940	0.719	-0.664

- Plot the posterior mean of f over the interval $x \in [-1, 1]$.
- Also plot 95% probability (pointwise) bands for f.

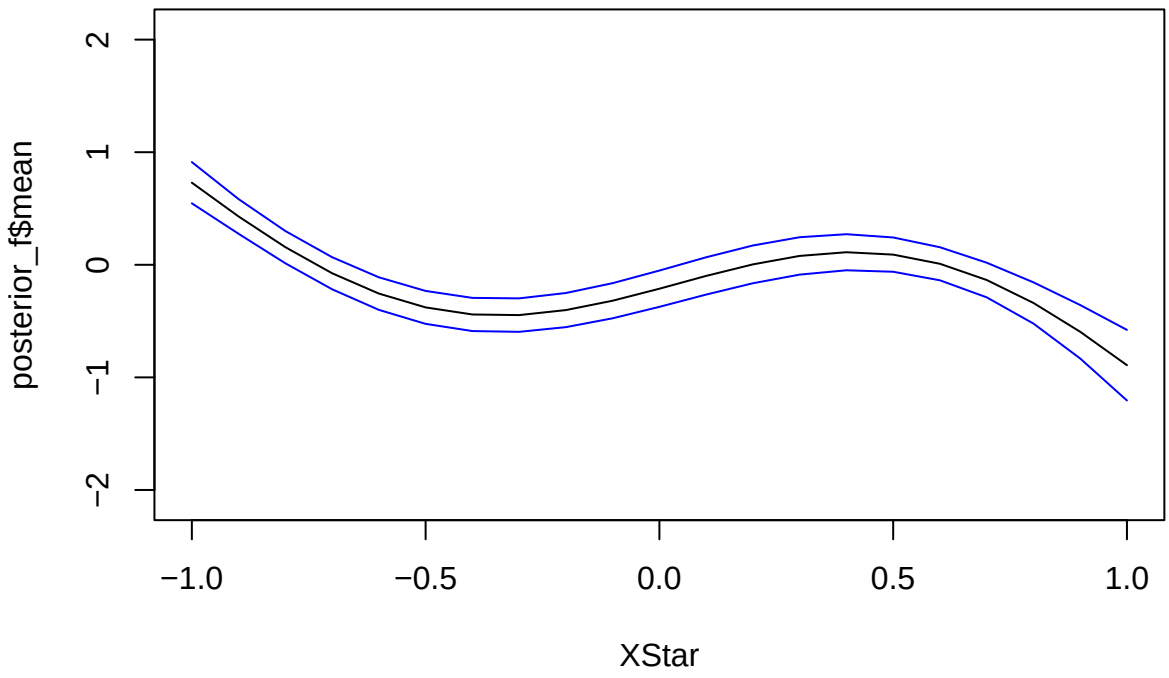
```
sigmaF = 1
l = 0.3
x = c(-1.0, -0.6, -0.2, 0.4, 0.8)
y = c(0.768, -0.044, -0.940, 0.719, -0.664)
sigmaNoise = 0.1
XStar = seq(-1, 1, 0.1)

posterior_f = posteriorGP(x, y, XStar, sigmaNoise, k=SquaredExpKernel, sigmaF, l)
plot(XStar, posterior_f$mean, type = "l", ylim = c(-2.1, 2.1))
lines(XStar, posterior_f$mean - 1.96*sqrt(diag(posterior_f$variance)), col = "blue")
lines(XStar, posterior_f$mean + 1.96*sqrt(diag(posterior_f$variance)), col = "blue")
```



4.1.5

Repeat the previous step using the five data points in the table, but this time with hyperparameters $\sigma_f = 1$ and $\ell = 1$.

```
l = 1

posterior_f = posteriorGP(x, y, XStar, sigmaNoise, k=SquaredExpKernel, sigmaF, l)
plot(XStar, posterior_f$mean, type = "l", ylim = c(-2.1, 2.1))
lines(XStar, posterior_f$mean - 1.96*sqrt(diag(posterior_f$variance)), col = "blue")
lines(XStar, posterior_f$mean + 1.96*sqrt(diag(posterior_f$variance)), col = "blue")
```



Q: Compare the results.
A: Gives a smoother graph since we increased the scale of the kernel, causing points further away to affect the given point more.

4.2 GP Regression with kernlab

In this exercise, you will work with the daily mean temperature in Stockholm (Tullinge) during the period January 1, 2010 - December 31, 2015. The leap day (February 29, 2012) has been removed to simplify the dataset.

Create the variable **time** which records the day number since the start of the dataset (i.e., time $= 1, 2, \ldots, 365 \times 6 = 2190$). Also, create the variable **day** that records the day number since the start of each year (i.e., day $= 1, 2, \ldots, 365, 1, 2, \ldots, 365$). Estimating a GP on 2190 observations can take some time on slower computers, so let us subsample the data and use only every fifth observation. This means that your time and day variables are now time $= 1, 6, 11, \ldots, 2186$ and day $= 1, 6, 11, \ldots, 361, 1, 6, 11, \ldots, 361$.

```
rm(list = ls())
data = read.csv(
  "https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/TempTullinge.csv",
  header=TRUE, sep=";")

data$time = 1:nrow(data)
data$day = ((data$time - 1) %% 365) + 1
data = data[seq(1, nrow(data), by=5), ]
```

4.2.1

Go through the file *KernLabDemo.R* available on the course website. You will need to understand it. Now, define your own square exponential kernel function (with parameters ℓ (ell) and σ_f (sigmaf)), evaluate it in the point $x = 1$, $x' = 2$, and use the `kernelMatrix` function to compute the covariance matrix $K(X, X^*)$ for the input vectors $X = (1, 3, 4)^T$ and $X^* = (2, 3, 4)^T$.

```
library(kernlab)

SquareExponential <- function(sigmaf, ell)
{
  rval <- function(x, y = NULL) {
    n1 <- length(x)
    n2 <- length(y)
    k <- matrix(NA,n1,n2)
    for (i in 1:n2){
      k[,i] <- sigmaf**2*exp(-0.5*( (x-y[i])/ell)**2 )
    }
    return(k)
  }
  class(rval) <- "kernel"
  return(rval)
}

X <- matrix(c(1, 3, 4))
Xstar <- matrix(c(2, 3, 4))

SEkernel = SquareExponential(sigmaf = 1, ell = 1) # SEkernel is a kernel FUNCTION.
SEkernel(1, 2) # Evaluating the kernel in x=1, x'=2.
```

```
##           [,1]
## [1,] 0.6065307
```

```
# Computing the whole covariance matrix K from the kernel.
kernelMatrix(kernel = SEkernel, x = X, y = Xstar) # So this is K(X,Xstar).
```

```
## An object of class "kernelMatrix"
##           [,1]      [,2]      [,3]
## [1,] 0.6065307 0.1353353 0.0111090
## [2,] 0.6065307 1.0000000 0.6065307
## [3,] 0.1353353 0.6065307 1.0000000
```

```
# kernelMatrix(kernel = SquareExponential(1,2), x = X, y = Xstar) # Also works.
```

4.2.2 Gaussian Process Regression Model

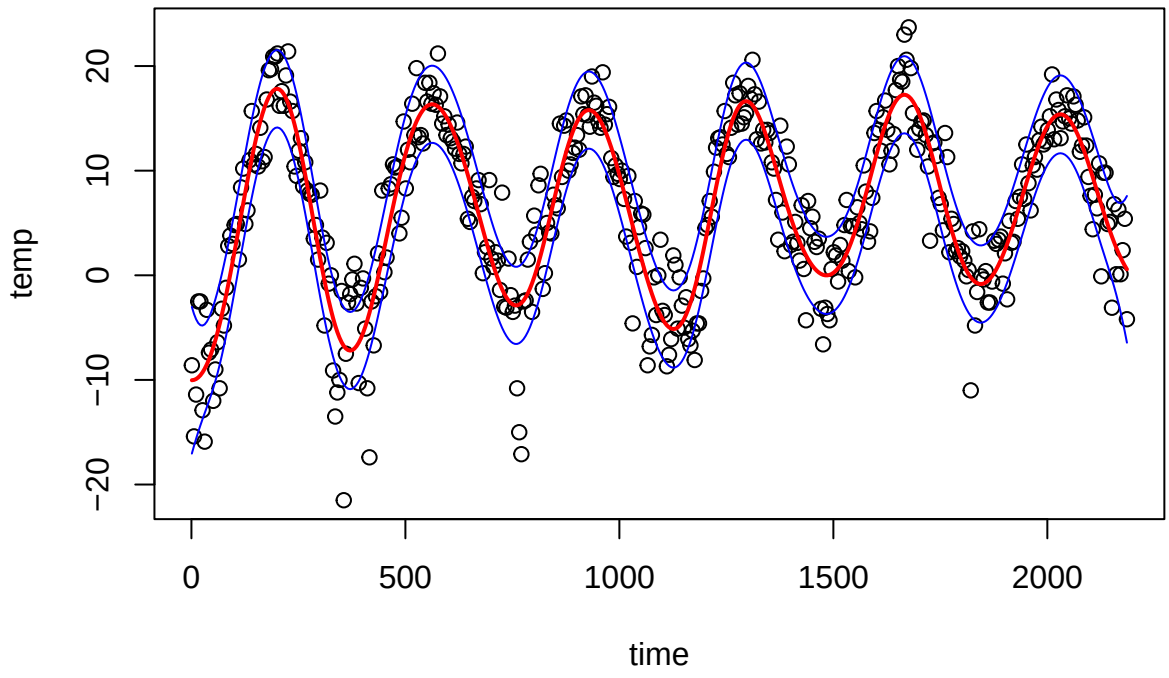
Consider first the following model:

$$\text{temp} = f(\text{time}) + \epsilon \quad \text{with} \quad \epsilon \sim \mathcal{N}(0, \sigma_n^2) \quad \text{and} \quad f \sim \mathcal{GP}(0, k(\text{time}, \text{time}'))$$

Let σ_n^2 be the residual variance from a simple quadratic regression fit (using the `lm` function in R). Estimate the above Gaussian process regression model using the `gausspr` function with the squared exponential

function from (1) with $\sigma_f = 20$ and $\ell = 100$ (use the option `scaled=FALSE` in the `gausspr` function, otherwise these σ_f and ℓ values are not suitable). Use the `predict` function in R to compute the posterior mean at every data point in the training dataset. Make a scatterplot of the data and superimpose the posterior mean of f as a curve (use `type="l"` in the `plot` function). Plot also the 95% probability (pointwise) bands for f. Play around with different values on σ_f and ℓ (no need to write this in the report though).

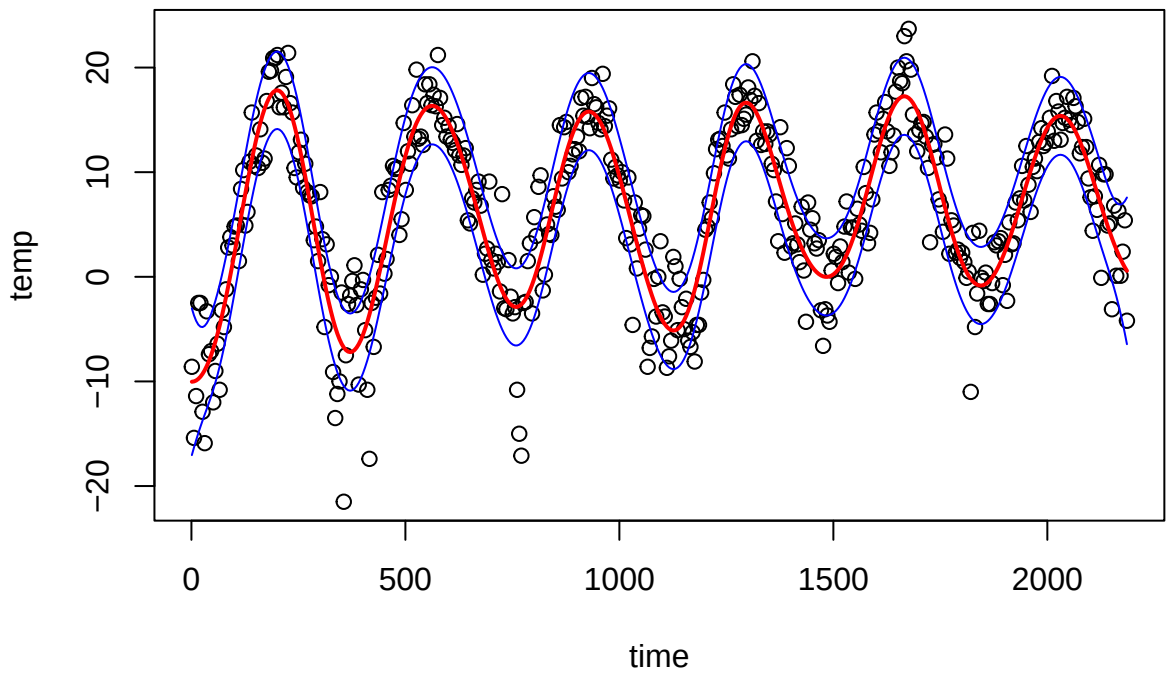
```
temp = data$temp
time = data$time

polyFit = lm(temp~I(time) + I(time**2))
sigmaNoise = sd(polyFit$residuals)
plot(time, temp)

GPfit = gausspr(time, temp, kernel = SquareExponential(ell = 100, sigmaf = 20),
                var = sigmaNoise**2, variance.model = TRUE,scaled=FALSE)

time_meanPred <- predict(GPfit, time)
lines(time, time_meanPred, col="red", lwd = 2)
lines(time, time_meanPred+1.96*predict(GPfit,time, type="sdeviation"),col="blue")
lines(time, time_meanPred-1.96*predict(GPfit,time, type="sdeviation"),col="blue")
```



4.2.3 Algorithm 2.1

Repeat the previous exercise, but now use Algorithm 2.1 on page 19 of Rasmussen and Williams' book to compute the posterior mean and variance of f.

```
posteriorGP = function(x, y, XStar, sigmaNoise, k, ...){
  K = k(x, x, ...)
  L = t(chol(K + diag(sigmaNoise**2, length(x))))
  alpha = solve(t(L), solve(L,y))

  kStar = k(x, XStar, ...)
  predictive_mean = t(kStar)%*%alpha

  v = solve(L, kStar)
  predictive_variance = k(XStar, XStar, ...) - t(v)%*%v

  return(list(mean = predictive_mean, variance = predictive_variance))
}

predictions = posteriorGP(x = time, y = temp, XStar = time, sigmaNoise = sigmaNoise,
                          k = SquareExponential(sigmaf = 20, ell = 100))
plot(time, temp)
lines(time, predictions$mean, col="red", lwd = 2)
lines(time, predictions$mean+1.96*sqrt(diag(predictions$variance)),col="blue")
lines(time, predictions$mean-1.96*sqrt(diag(predictions$variance)),col="blue")
```



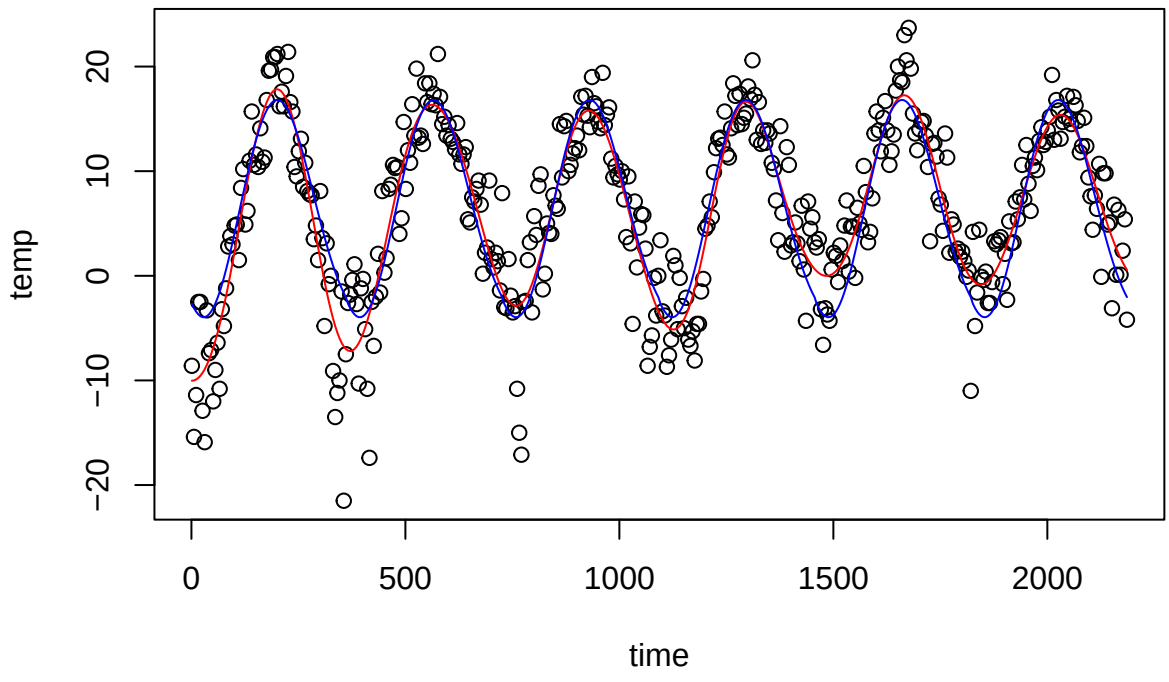
4.2.4 New Model Based on Day

Consider now the following model:

$$\text{temp} = f(\text{day}) + \epsilon \quad \text{with} \quad \epsilon \sim \mathcal{N}(0, \sigma_n^2) \quad \text{and} \quad f \sim \mathcal{GP}(0, k(\text{day}, \text{day}'))$$

Estimate the model using the `gausspr` function with the squared exponential function from (1) with $\sigma_f = 20$ and $\ell = 100$ (use the option `scaled=FALSE` in the `gausspr` function, otherwise these σ_f and ℓ values are not suitable). Superimpose the posterior mean from this model on the posterior mean from the model in (2). Note that this plot should also have the time variable on the horizontal axis.

```
temp = data$temp
day = data$day

polyFit = lm(temp~I(day) + I(day**2))
sigmaNoise = sd(polyFit$residuals)

GPfit = gausspr(day, temp, kernel = SquareExponential(ell = 100, sigmaf = 20),
                var = sigmaNoise**2, variance.model = TRUE,scaled=FALSE)

day_meanPred <- predict(GPfit, day)
plot(time, temp)
lines(time, time_meanPred, col = "red")
lines(time, day_meanPred, col = "blue")
```



Q: Compare the results of both models. What are the pros and cons of each model?

A: The model with time as its input feature seem to have slight more variance than the one with day. Day values repeats itself, meaning that the 1st of January 2010 is the same value as 1st of January 2011. The

model therefore gets multiple temperatures to train on for the same day value, while for the model with time, 1st of Jan each year is distinct.

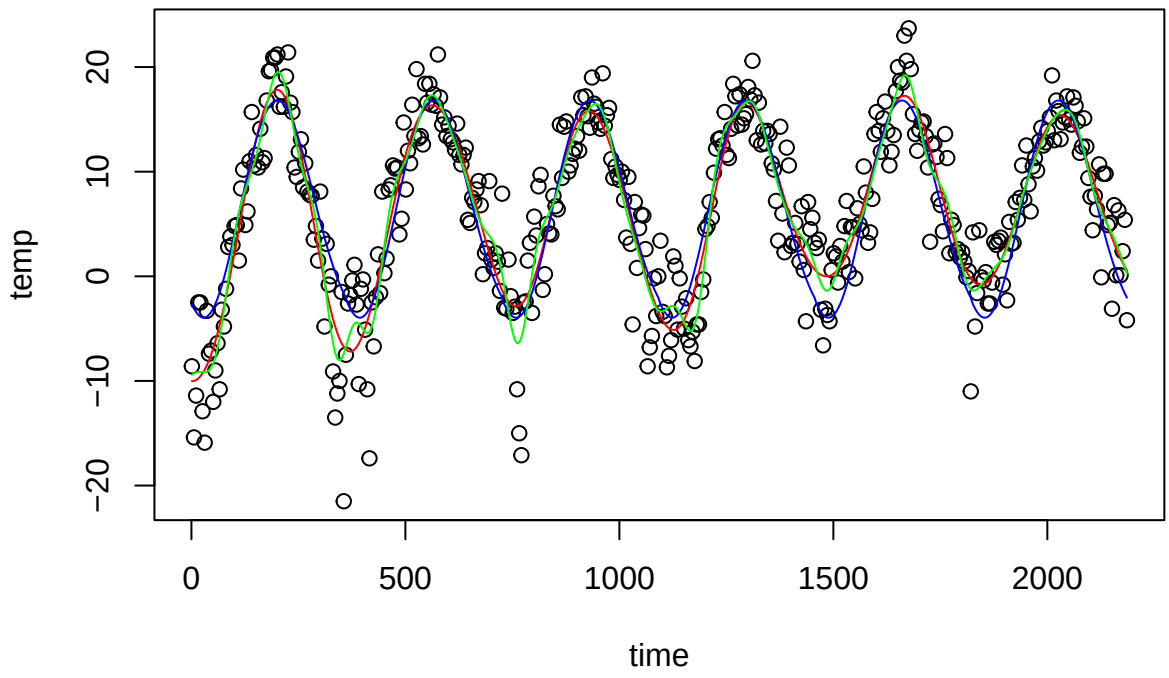
4.2.5 Extended Squared Exponential Kernel

Implement the following periodic kernel (a.k.a. locally periodic kernel):

$$k(x, x') = \sigma_f^2 \exp\left\{-\frac{2\sin^2(\pi|x - x'|/d)}{\ell_1^2}\right\} \exp\left\{-\frac{1}{2}\frac{|x - x'|^2}{\ell_2^2}\right\}$$

Note that we have two different length scales in the kernel. Intuitively, ℓ_1 controls the correlation between two days in the same year, and ℓ_2 controls the correlation between the same day in different years. Estimate the GP model using the time variable with this kernel and hyperparameters $\sigma_f = 20$, $\ell_1 = 1$, $\ell_2 = 100$, and $d = 365$. Use the `gausspr` function with the option `scaled=FALSE`, otherwise these σ_f, ℓ_1, and ℓ_2 values are not suitable.

```
periodic <- function(sigmaf, ell1, ell2, d) {
    rval <- function(x1, y = NULL) {
      n1 <- length(x1)
      n2 <- length(y)
      K <- matrix(NA,n1,n2)
      for (i in 1:n1){
        for (j in 1:n2){
          r = sqrt(crossprod(x1[i]-y[j]))
          factor1 = sigmaf**2*exp(-0.5*r**2/ell2**2)
          factor2 = exp(-2*(sin(pi*r/d)**2)/ell1**2)
          K[i,j] <- factor1*factor2
        }
      }
      return(K)
    }

  class(rval) <- "kernel"
  return(rval)
}

GPfit <- gausspr(time, temp, kernel = periodic(sigmaf=20, ell1=1, ell2=100, d=365),
                 var = sigmaNoise**2, variance.model = TRUE, scaled=FALSE)

periodic_meanPred = predict(GPfit, time)

plot(time, temp)
lines(time, time_meanPred, col = "red")
lines(time, day_meanPred, col = "blue")
lines(time, periodic_meanPred, col = "green")
```



Q: Compare the fit to the previous two models (with $\sigma_f = 20$ and $\ell = 100$). Discuss the results. A: The new model with a periodic kernel allows to capture periodic patterns, creating a model with higher variance.

4.3 GP Classification with kernlab

Choose 1000 observations as training data.

```
rm(list = ls())
data <- read.csv(
  "https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/banknoteFraud.csv",
  header=FALSE, sep=",")
names(data) <- c("varWave","skewWave","kurtWave","entropyWave","fraud")
data[,5] <- as.factor(data[,5])
set.seed(111); SelectTraining <- sample(1:dim(data)[1], size = 1000, replace = FALSE)
train_data = data[SelectTraining, ]
test_data = data[-SelectTraining, ]
```

4.3.1.

Use the R package `kernlab` to fit a Gaussian process classification model for fraud on the training data. Use the default kernel and hyperparameters. Start using only the covariates `varWave` and `skewWave` in the model. Plot contours of the prediction probabilities over a suitable grid of values for `varWave` and `skewWave`. Overlay the training data for `fraud = 1` (as blue points) and `fraud = 0` (as red points). You can reuse code from the file *KernLabDemo.R* available on the course website. Compute the confusion matrix for the classifier and its accuracy.

```r
library(kernlab)
library(AtmRay)
GPfitFraud <- gausspr(fraud ~ varWave + skewWave, data=train_data)
```

```
## Using automatic sigma estimation (sigest) for RBF or laplace kernel
```

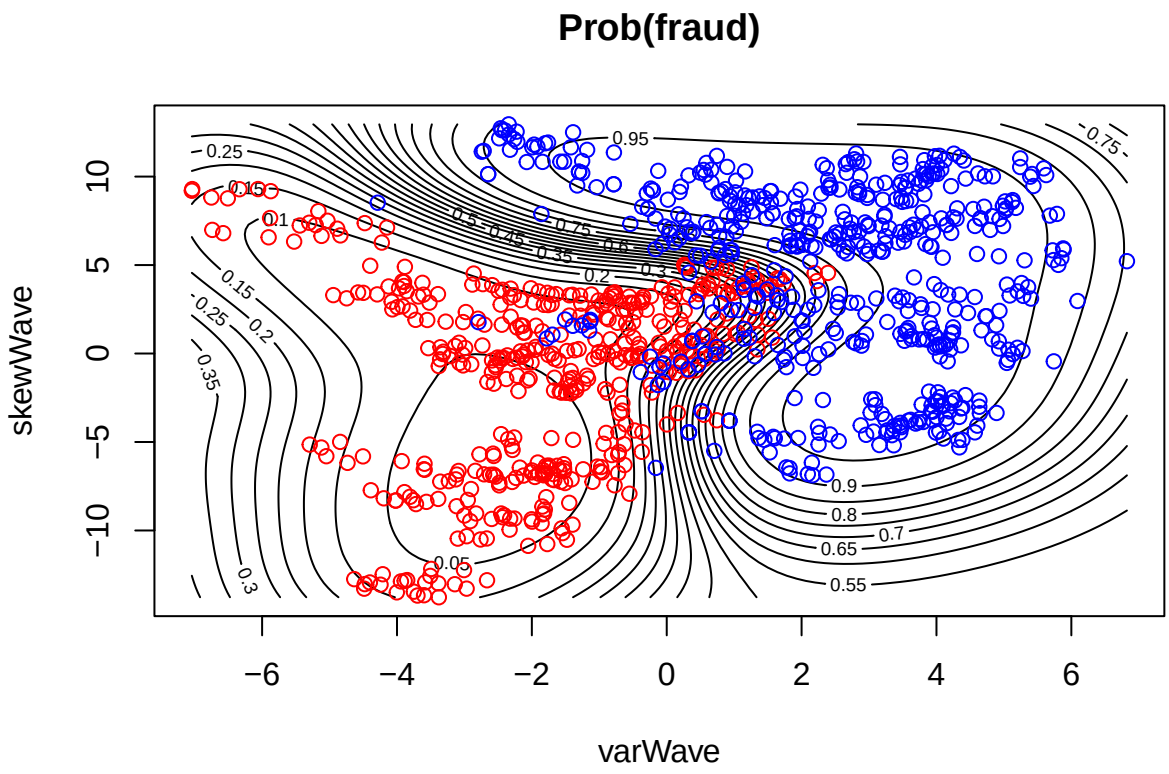
```r
classPreds = predict(GPfitFraud, newdata=train_data)
cm = table(classPreds, train_data$fraud) # confusion matrix
cm
```

```
##
## classPreds   0   1
##          0 503  18
##          1  41 438
```

```r
sum(diag(cm))/sum(cm)
```

```
## [1] 0.941
```

```r
x1 <- seq(min(train_data[,1]),max(train_data[,1]),length=100)
x2 <- seq(min(train_data[,2]),max(train_data[,2]),length=100)
gridPoints <- meshgrid(x1, x2)
gridPoints <- cbind(c(gridPoints$x), c(gridPoints$y))

gridPoints <- data.frame(gridPoints)
names(gridPoints) <- names(train_data)[1:2]
probPreds <- predict(GPfitFraud, gridPoints, type="probabilities")

# Plotting for Prob(fraud)
contour(x1,x2,matrix(probPreds[,1],100,byrow = TRUE), 20, xlab = "varWave",
        ylab = "skewWave", main = 'Prob(fraud)')
points(train_data[train_data[,5]==1,1],train_data[train_data[,5]==1,2],col="red")
points(train_data[train_data[,5]==0,1],train_data[train_data[,5]==0,2],col="blue")
```



4.3.2.

Using the estimated model from 3.1, make predictions for the test set. Compute the accuracy.

```
classPreds = predict(GPfitFraud, newdata=test_data)
cm = table(classPreds, test_data$fraud) # confusion matrix
cm
```

```
##
## classPreds   0   1
##          0 199   9
##          1  19 145
```

```
sum(diag(cm))/sum(cm)
```

```
## [1] 0.9247312
```

4.3.3.

Train a model using all four covariates. Make predictions on the test set and compare the accuracy to the model with only two covariates.

```
GPfitFraud <- gausspr(fraud ~ ., data=train_data)
```

```
## Using automatic sigma estimation (sigest) for RBF or laplace kernel
```

```
classPreds = predict(GPfitFraud, newdata=test_data)
cm = table(classPreds, test_data$fraud) # confusion matrix
cm
```

```
##
## classPreds   0   1
##          0 216   0
##          1   2 154
```

```
sum(diag(cm))/sum(cm)
```

```
## [1] 0.9946237
```

The accuracy is a lot better (0.9946237 vs. 0.9247312)