TDDE15 - all labs merged

Oscar Hoffmann

October 23, 2024

Lab1

Q1 Multiple Runs of Hill-Climbing Algorithm

Show that multiple runs of the hill-climbing algorithm can return non-equivalent Bayesian network (BN) structures. Explain why this happens.

```
# Clear the workspace
rm(list = ls())
# Libraries and load the data
library(bnlearn)
library(gRain)
```

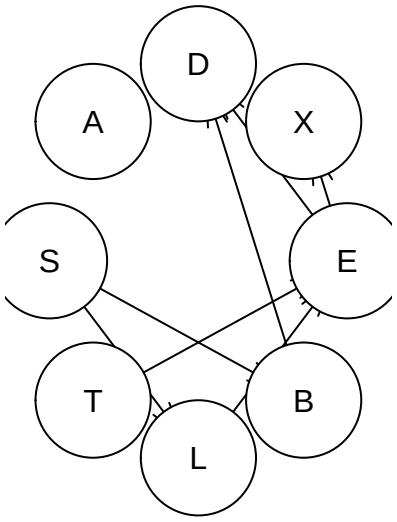
```
## Loading required package: gRbase
```

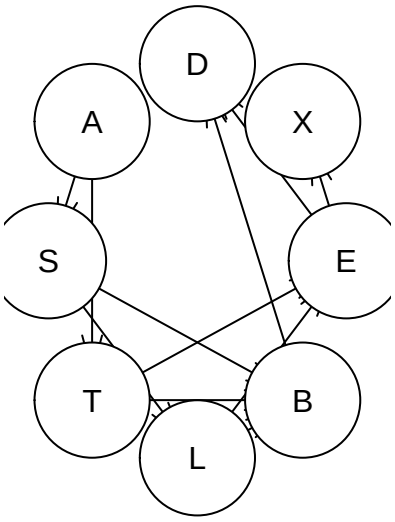
```
##
## Attaching package: 'gRbase'
```

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

```
data("asia")
```

```
#Two models, vary score method
model1 = hc(asia, score = "bic")
model2 = hc(asia, score = "aic")

par(mfrow = c(1, 2))
plot(model1, main = "Hill-climbing with BIC Score")
plot(model2, main = "Hill-climbing with AIC Score")
```



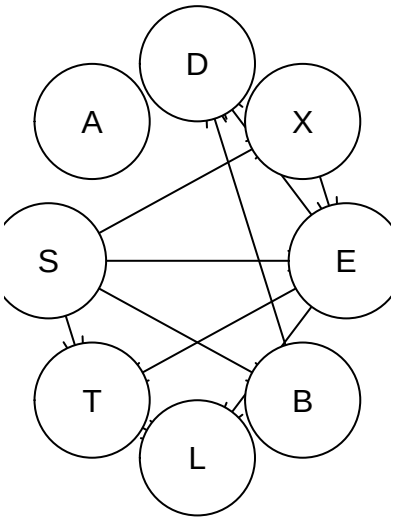



```
all.equal(cpdag(model1), cpdag(model2))
```

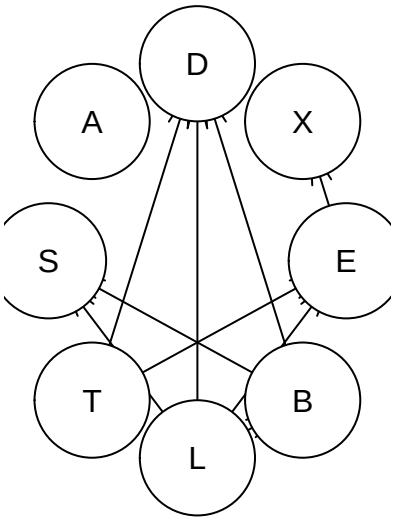
```
## [1] "Different number of directed/undirected arcs"
```

```
#Two models, vary initial structure
set.seed(12345)
model1 = hc(asia, start = random.graph(colnames(asia)))
set.seed(1234)
model2 = hc(asia, start = random.graph(colnames(asia)))

par(mfrow = c(1, 2))
plot(model1, main = "Random init 1")
plot(model2, main = "Random init 2")
```

Random init 1



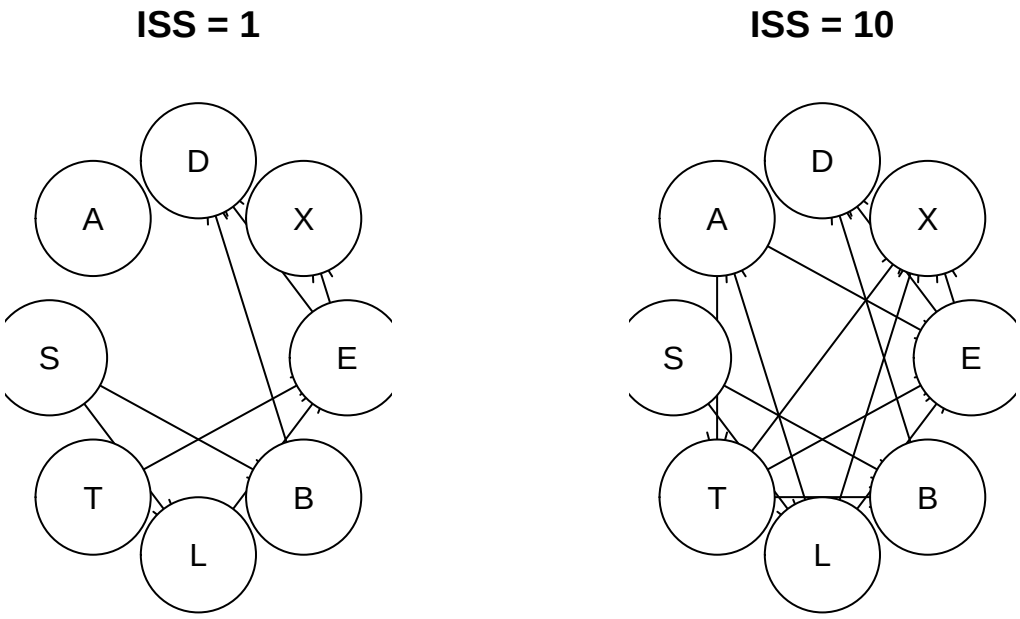
Random init 2




```
all.equal(cpdag(model1), cpdag(model2))
```

```
## [1] "Different number of directed/undirected arcs"
```

```
#Two models, vary imaginary sample size
model1 = hc(asia, score = "bde", iss=1)
model2 = hc(asia, score = "bde", iss=10)

par(mfrow = c(1, 2))
plot(model1, main = "ISS = 1")
plot(model2, main = "ISS = 10")
```




```
all.equal(cpdag(model1), cpdag(model2))
```

```
## [1] "Different number of directed/undirected arcs"
```

Here I have shown that by varying score method, initial structure and ISS the HC algorithm can yield non-equivalent Bayesian networks. This can been seen in the plots (run the code to see) but also explicitly by converting the models with cpdag and comparing them with all.equals. Cpdag converts the networks to equivalence classes which need to be done to compare them. This is because two Bayesian networks could have different edge directions between nodes, but if the conditional independence properties are the same, they are considered equivalent. However, if they do not encode the same independences, they are non-equivalent. The reason that we end up with non-equivalent BNs is that the HC algorithm is a greedy algorithm that is not guaranteed to find the global optima but rather often get stuck in local optimas.

Q2 Learning a BN and Classification

Learn a BN from 80% of the Asia dataset. Use the BN learned to classify the remaining 20% of the dataset in two classes: S = yes and S = no. Perform inference and report Confusion Matrix.

```
rm(list = ls())
data("asia")

#divide into 80% train and 20% test
n=dim(asia)[1]
set.seed(12345)
```

```r
id=sample(1:n, floor(n*0.8))
train=asia[id,]
test=asia[-id,]
```

```r
#learn both structure and init true_dag
model_struct = hc(train, restart = 10)
true_dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")

# learn params
model_params = bn.fit(model_struct, data = train)
true_dag_params = bn.fit(true_dag, data = train)

# Exact inference, transform into gRain objects
model_grain = as.grain(model_params)
model_compiled = compile(model_grain)

true_dag_grain = as.grain(true_dag_params)
true_dag_compiled = compile(true_dag_grain)

# Convert test to a data frame to utilize apply
test = as.data.frame(test)

# Define the nodes (excluding S)
nodes = colnames(test)[-2]

# Function to classify using a given compiled model
classify_with_model <- function(compiled_model, mb_or_S, nodes) {
  #The function here is applied over whole dataframe, 1 means rows, 2 is cols
  apply(test, 1, function(row) {
    # Set evidence for the current row
    evidence <- setEvidence(object = compiled_model,
                            nodes = nodes,
                            states = as.character(unlist(row[mb_or_S]))
                            )

    # Query the posterior probability of S
    query <- querygrain(evidence, nodes = "S")$S

    # Classify based on the probability of "yes"
    ifelse(query["yes"] > 0.5, "yes", "no")
  }
  )
}

# Get predictions for the learned model + cm
pred_model <- classify_with_model(model_compiled, -2, nodes)
table(pred_model, test$S)
```

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

```r
# Get predictions for the true DAG model
pred_true_dag <- classify_with_model(true_dag_compiled, -2, nodes)
table(pred_true_dag, test$S)
```

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

The confusion matrices are identical so learnt models is as good at classifing as the true dag

Q3 Markov Blanket Classification

Classify S given observations only for its Markov blanket. Report the confusion matrix.

```r
#Use Markov blanket to do inference more efficiently
mb = mb(model_params, node = "S")
true_mb = mb(true_dag_params, node ="S")

# Get predictions for the learned model with mb
mb_pred <- classify_with_model(model_compiled, mb, mb)
table(mb_pred, test$S)
```

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

```r
# Get predictions for the true DAG model with mb
true_mb_pred <- classify_with_model(true_dag_compiled, true_mb, true_mb)
table(true_mb_pred, test$S)
```

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

Using only the Markov blanket variables to classify S yields the same performance as using all other variables. This is expected because the Markov blanket contains all the information needed to predict S.

Q4 Naive Bayes Classifier

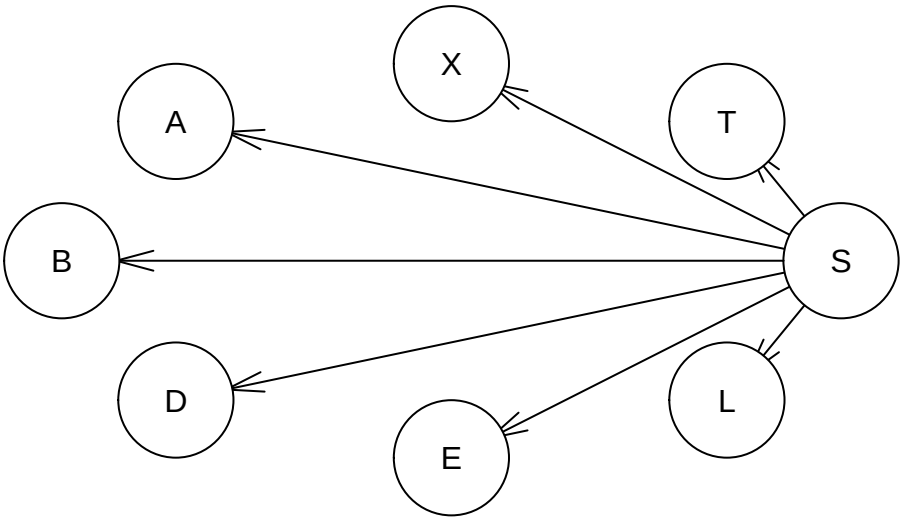
Repeat exercise 2 using a naive Bayes classifier. Model the naive Bayes classifier as a BN created by hand.

```r
# Naive Bayes classifier
nb <- model2network("[S][A|S][B|S][D|S][E|S][L|S][T|S][X|S]")
nbc <- bn.fit(x = nb, data = train)

# Transform into gRain obj
nbc <- as.grain(nbc)
```

```r
nbc <- compile(nbc)

# Confusion matrix
table(classify_with_model(nbc, -2, nodes), test$S)
```

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

```r
plot(nb)
```



Q5 Explanation of Results

Q: Explain why you obtain the same or different results in exercises 2-4.

A: Compared to the confusion matrices from exercises 2, the naive Bayes classifier shows different results, with a decrease in classification performance. The differences arise because the naive Bayes model does not capture the true dependency structure among the variables, leading to less accurate predictions.

Lab2

Q1: Build the hidden markov model

```r
# Load the HMM package
library(HMM)
library(entropy)
```

```
##
## Attaching package: 'entropy'

## The following object is masked from 'package:bnlearn':
##
##     discretize
```

```r
rm(list = ls())

# Define the hidden states and observation symbols
states <- 1:10
symbols <- 1:10

# Initialize the initial state probabilities: the robot is equally likely to start in any sector
start_probs <- rep(1/10, 10)  # A vector of 0.1's for each state

# Initialize the transition probability matrix with zeros
trans_probs <- matrix(0, nrow = 10, ncol = 10)

# can also be done manually with:
# trans_probs = matrix(c(X,X,X..), nrow=length(States), ncol=length(States), byrow = TRUE)

# or by:

# colnames(trans_probs) = states
# rownames(trans_probs) = states
# trans_probs["S1 C2", "S1 C1"] = 1

# Fill in the transition probabilities
for (i in 1:10) {
  # Stay in the current sector with probability 0.5
  trans_probs[i, i] <- 0.5

  # Move to the next sector with probability 0.5
  # Wrap around from sector 10 to sector 1
  next_sector <- ifelse(i == 10, 1, i + 1)
  trans_probs[i, next_sector] <- 0.5
}

# Initialize the emission probability matrix with zeros
emission_probs <- matrix(0, nrow = 10, ncol = 10)
```

```r
# the dim of the emission matrix is nr of states (rows) x nr of symbols(cols)
# symbols are the observations. they are based on the hidden states but don't reveal them
# Emission Matrix: This matrix defines the probability of observing each symbol given a hidden state.

# Fill in the emission probabilities
for (i in 1:10) {
  # Sectors [i-2, i-1, i, i+1, i+2] with wrap-around
  sectors <- ((i - 3):(i + 1)) %% 10 + 1  # Adjust for 1-based indexing
  # Assign equal probability to each possible observed sector 0.2
  emission_probs[i, sectors] <- 1/5
}

# Initialize the Hidden Markov Model
hmm_model <- initHMM(
  States = states,           # vector of states
  Symbols = symbols,         # vector of observation symbols
  startProbs = start_probs,  # Initial state probabilities
  transProbs = trans_probs,  # Transition probabilities matrix
  emissionProbs = emission_probs  # Emission probabilities matrix
)
```

Q2. Simulate 100 time steps

```r
set.seed(12345)
simulation <- simHMM(hmm_model, length = 100)
```

Q3. Compute filtered, smooth probability distributions and most probable path

```r
# Extract observations of the tracking device
observations <- simulation$observation

# Compute log forward probabilities using the forward algorithm
log_forward_probs <- forward(hmm_model, observations)

# Compute log backward probabilities using the backward algorithm
log_backward_probs <- backward(hmm_model, observations)

# Compute log smoothed probabilities loga + logb = log(a*b)
log_smoothed_probs <- log_forward_probs + log_backward_probs


# Convert log probabilities to normal scale
filtered_probs <- exp(log_forward_probs)
smoothed_probs <- exp(log_smoothed_probs)

# Compute the most probable path using the Viterbi algorithm
viterbi_path <- viterbi(hmm_model, observations)
```

Q4. Compute Accuracies

```r
# normalize
filtered_probs <- apply(filtered_probs, 2, prop.table)
smoothed_probs <- apply(smoothed_probs, 2, prop.table)

# Find the most probable state at each time step
pred_filtered <- apply(filtered_probs, 2, which.max)
pred_smoothed <- apply(smoothed_probs, 2, which.max)

# Calculate accuracies
calc_accuracy = function(predicted_values, true_states) {
  accuracy <- mean(predicted_values == true_states) * 100
  return(accuracy)
}

# Output the accuracies
cat("\nAccuracy of the Filtered Probabilities:", calc_accuracy(pred_filtered, simulation$states), "%\n"
```

```
##
## Accuracy of the Filtered Probabilities: 53 %
```

```r
cat("Accuracy of the Smoothed Probabilities:", calc_accuracy(pred_smoothed, simulation$states), "%\n")
```

```
## Accuracy of the Smoothed Probabilities: 74 %
```

```r
cat("Accuracy of the Viterbi Path:", calc_accuracy(viterbi_path, simulation$states), "%\n")
```

```
## Accuracy of the Viterbi Path: 56 %
```

Q5. Repeat with different simulated samples

```r
n_simulations <- 10

# Initialize vectors to store accuracies
accuracy_filtered <- numeric(n_simulations)
accuracy_smoothed <- numeric(n_simulations)
accuracy_viterbi <- numeric(n_simulations)

# Run the simulation multiple times
for (s in 1:n_simulations) {

  sim <- simHMM(hmm_model, length = 100)
  observations <- sim$observation

  log_forward_probs <- forward(hmm_model, observations)
  log_backward_probs <- backward(hmm_model, observations)

  log_smoothed_probs <- log_forward_probs + log_backward_probs
```

```
  filtered_probs <- exp(log_forward_probs)
  smoothed_probs <- exp(log_smoothed_probs)

  filtered_probs <- apply(filtered_probs, 2, prop.table)
  smoothed_probs <- apply(smoothed_probs, 2, prop.table)

  viterbi_path <- viterbi(hmm_model, observations)

  pred_filtered <- apply(filtered_probs, 2, which.max)
  pred_smoothed <- apply(smoothed_probs, 2, which.max)

  # Calculate and store accuracies
  accuracy_filtered[s] <- calc_accuracy(pred_filtered, sim$states)
  accuracy_smoothed[s] <- calc_accuracy(pred_smoothed, sim$states)
  accuracy_viterbi[s] <- calc_accuracy(viterbi_path, sim$states)
}

# Output the results
cat("Filtered Probabilities:", mean(accuracy_filtered), "%\n")
```

```
## Filtered Probabilities: 51.3 %
```

```
cat("Smoothed Probabilities:", mean(accuracy_smoothed), "%\n")
```

```
## Smoothed Probabilities: 69.5 %
```

```
cat("Viterbi Path:", mean(accuracy_viterbi), "%\n")
```

```
## Viterbi Path: 53.6 %
```

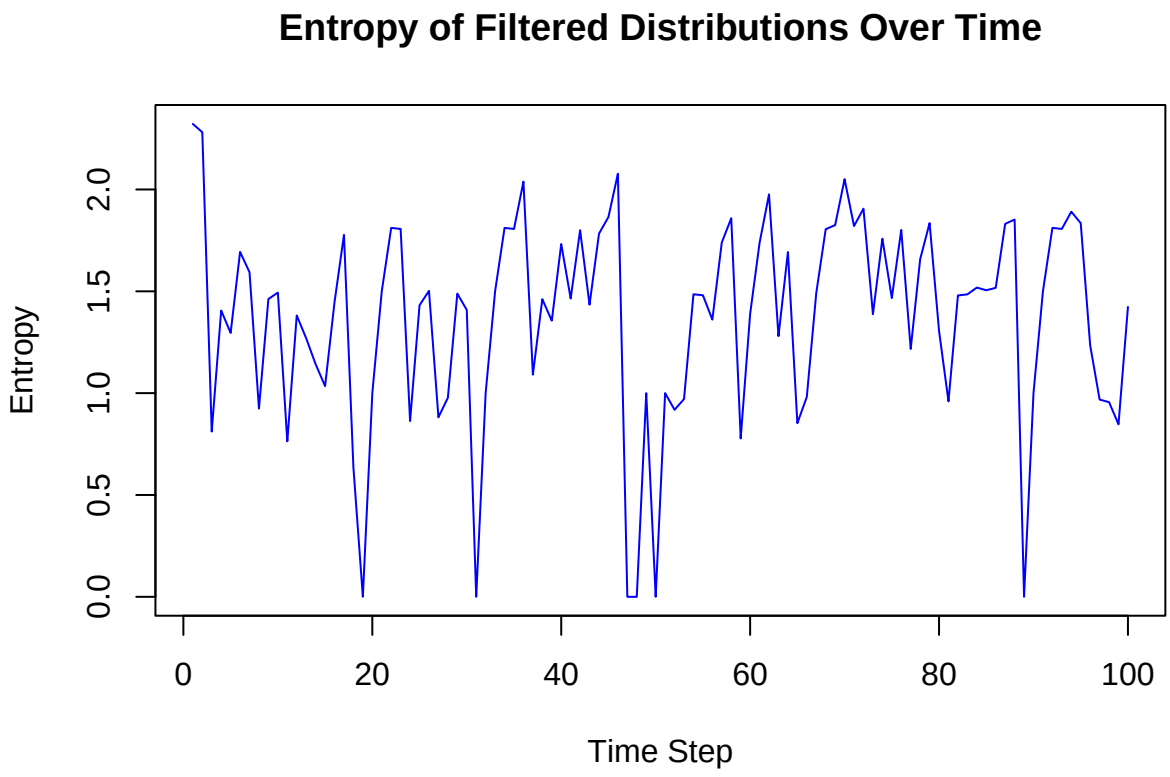
```
# Display accuracies for each simulation
results_df <- data.frame(
  Simulation = 1:n_simulations,
  Filtered = accuracy_filtered,
  Smoothed = accuracy_smoothed,
  Viterbi = accuracy_viterbi
)
print(results_df)
```

```
##    Simulation Filtered Smoothed Viterbi
## 1           1       46       68      61
## 2           2       49       77      65
## 3           3       49       58      56
## 4           4       60       80      65
## 5           5       54       64      39
## 6           6       54       69      53
## 7           7       52       67      55
## 8           8       52       67      50
## 9           9       45       68      45
## 10         10       52       77      47
```

Smoothed distributions are generally more accurate because they leverage more information (both past and future observations). This is supported by the average accuracies attained in this task. The Viterbi path is less accurate than smoothed distributions because it commits to a single sequence of states, which don't account for the uncertainty in the observations.

Q6. Entropy

```
entropy_filtered <- numeric(ncol(filtered_probs))  # Initialize vector to store entropy

# Calculate the entropy for each time step
for (t in 1:ncol(filtered_probs)) {
  # Calculate the empirical entropy for each time step's probability distribution
  entropy_filtered[t] <- entropy.empirical(filtered_probs[, t], unit = "log2")
}

# Plot entropy over time
plot(1:length(entropy_filtered), entropy_filtered, type = "l", col = "blue", xlab = "Time Step", ylab =
```



As seen in the graph the entropy seems to be fluctuating with no clear trend which suggest that later in time with more observations doesn't mean better predictions. Likely because the observations i-2:i+2 are too "nosy".

Q7. Compute the probabilities of the hidden states for the time step 101

```
filtered_prob_t100 <- filtered_probs[, 100]  # Vector of probabilities for each state at t=100

# Compute the probabilities for time step 101 by multiplying with the transition matrix
prob_t101 <- trans_probs %*% filtered_prob_t100
print(prob_t101)
```

```
##             [,1]
##  [1,] 0.31842105
##  [2,] 0.06842105
##  [3,] 0.00000000
##  [4,] 0.00000000
##  [5,] 0.00000000
##  [6,] 0.00000000
##  [7,] 0.00000000
##  [8,] 0.00000000
##  [9,] 0.18157895
## [10,] 0.43157895
```

Lab3

2.1 Q-Learning

Complete the Q-learning algorithm for a grid-world environment where the agent navigates using four actions (up, down, left, right) and faces a probability of slipping. The agent must learn the optimal policy using a model-free approach, as the transition model is unknown.

```
rm(list =ls())
set.seed(1234)
library(ggplot2)

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){

  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 the Q-values for all actions at state (x, y)
  q_values <- q_table[x, y, ]

  # Find the max Q-value
  max_q <- max(q_values)

  # Identify all actions with maximum Q-value
  max_actions <- which(q_values == max_q)

  # Check and resolve ties
  if (length(max_actions) > 1) {
    action <- sample(max_actions, 1)
  } else {
    action <- max_actions
  }
  return(action)
}

EpsilonGreedyPolicy <- function(x, y, epsilon){
  # Generate a random numb
  rand_num <- runif(1)
  if (rand_num < epsilon){
   #select a random action
    action <- sample(1:4, 1)
  } else {
  # use the greedy policy
    action <- GreedyPolicy(x, y)
  }
```

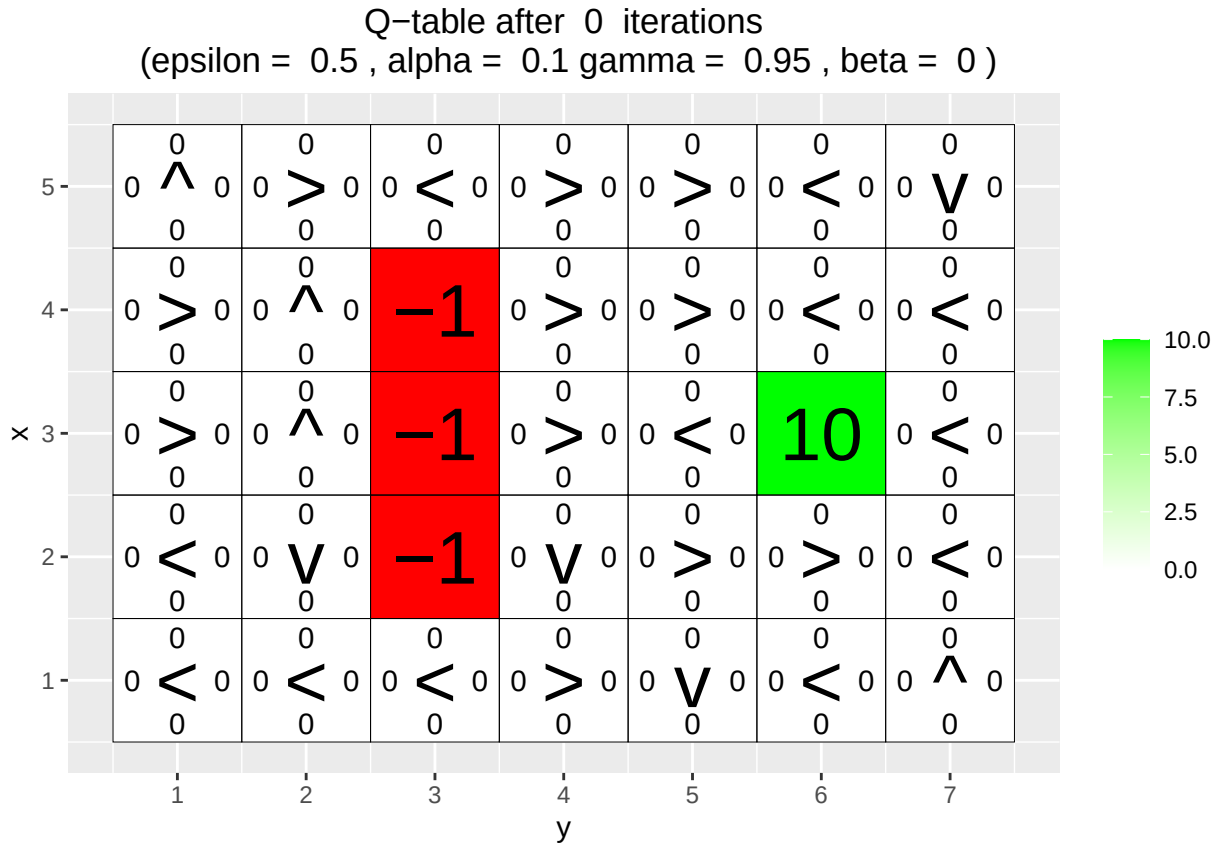
```r
  return(action)
}

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

  # Perform one episode of Q-learning. The agent should move around in the
  # environment using the given transition model and update the Q-table.
  # The episode ends when the agent reaches a terminal state.
  #
  # Args:
  #   start_state: array with two entries, describing the starting position of the agent.
  #   epsilon (optional): probability of acting randomly.
  #   alpha (optional): learning rate.
  #   gamma (optional): discount factor.
  #   beta (optional): slipping factor.
  #   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.
  #
  # Returns:
  #   reward: reward received in the episode.
  #   correction: sum of the temporal difference correction terms over the episode.
  #   q_table (global variable): Recall that R passes arguments by value. So, q_table being
  #   a global variable can be modified with the superassigment operator <<-.

  # Initialize state
  x <- start_state[1]
  y <- start_state[2]

  # Initialize variables to track episode reward and TD corrections
  episode_reward <- 0
  episode_correction <- 0

  repeat{
    # Choose an action A with epsilon-greedy policy
    action <- EpsilonGreedyPolicy(x, y, epsilon)

    # Observe next state S' & reward R after taking action A
    next_state <- transition_model(x, y, action, beta)
    x_new <- next_state[1]
    y_new <- next_state[2]
    R <- reward_map[x_new, y_new]

    # Get current Q-value Q(S, A)
    Q_SA <- q_table[x, y, action]
```

```
    # if R != 0 it means next state is terminal, this check if for when the terminal state
    # is the first action and the end of the loop hasn't been reached
    if (R != 0){
      max_QSAprime <- 0
    } else {
      # Compute max Q(S', a') over all possible actions a'
      max_QSAprime <- max(q_table[x_new, y_new, ])
    }

    # Calc TD correction term
    TD_correction <- R + gamma * max_QSAprime - Q_SA
    episode_correction <- episode_correction + TD_correction

    # Update Q(S, A)
    q_table[x, y, action] <<- Q_SA + alpha * TD_correction

    # Ep reward accumulated
    episode_reward <- episode_reward + R

    # Next state
    x <- x_new
    y <- y_new

    # Check if the episode has ended (terminal state)
    if (R != 0){
      return (c(episode_reward, episode_correction))
    }
  }
}
```

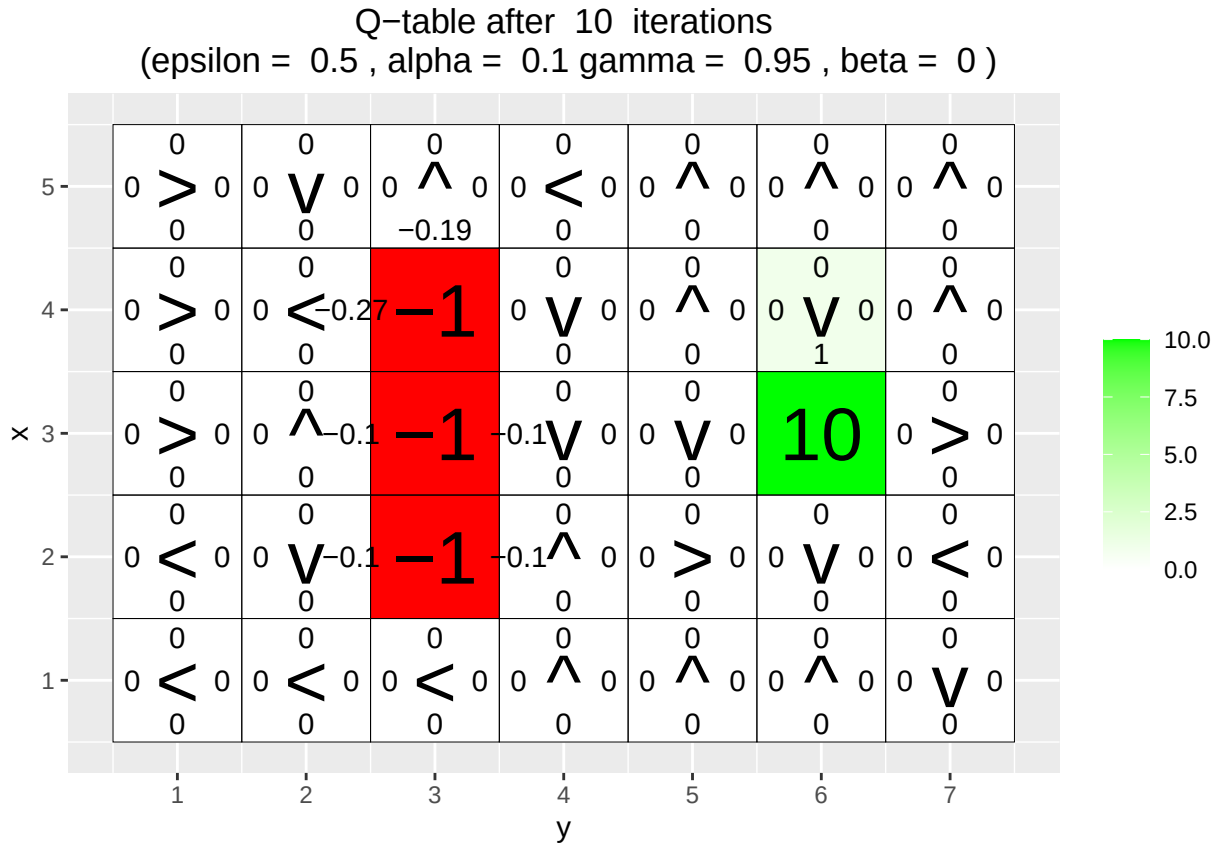
2.2 Environment A

Implement and visualize Q-learning in a 5×7 grid-world environment with specific rewards, and complete tasks including implementing greedy and epsilon-greedy policies, and running 10,000 episodes of Q-learning to analyze the learned policy and Q-table.

```
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()
```

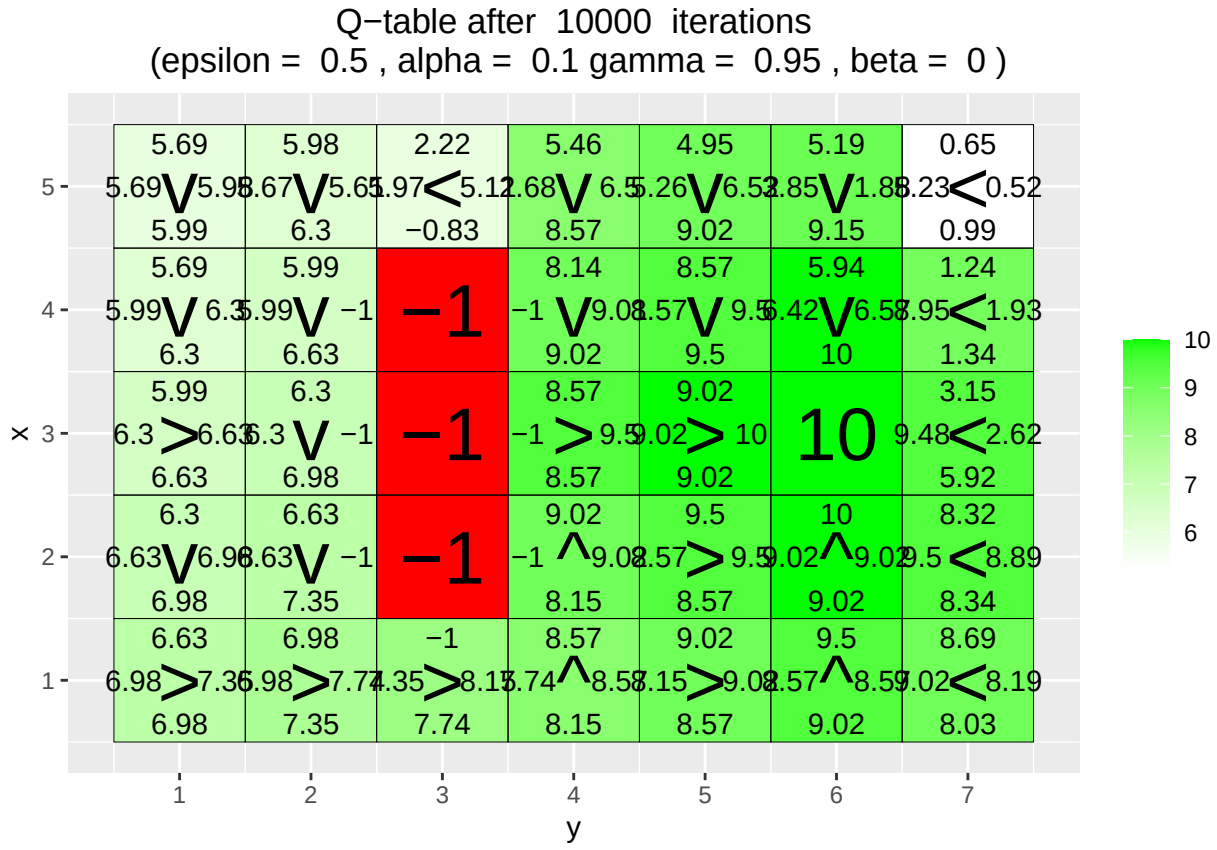



```
for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

  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


Q−table after 100 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
1
2
3
4
5
6
7
10.0
7.5
5.0
2.5
0.0
−1
10


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



What has the agent learned after the first 10 episodes?

The Q-table values are mostly zeros, with slight updates in some states around -1. The greedy policy (arrows on the grid) point randomly due to insufficient learning.

Is the final greedy policy (after 10000 episodes) optimal for all states, i.e. not only for the initial state ? Why / Why not ?

The agent has clearly learned to navigate towards the goal at position, as seen by the arrows pointing rightwards and downwards towards this cell. The policy is optimal for the most states that the agent has visited often during the episodes, however some areas are under explored leading to sub-optimal policies in those areas. An example is the states above the -1 penalty wall where it would be better to go above and not down again.

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 ?

Mostly no, the learned values in the Q-table do not reflect the fact that there are multiple path that lead to the positive reward. The agent clearly shows a strong preference for the lower path over the upper path. This is likely due to the agent's early discovery of the upper path and subsequent exploitation of it.

To counteract this one could:

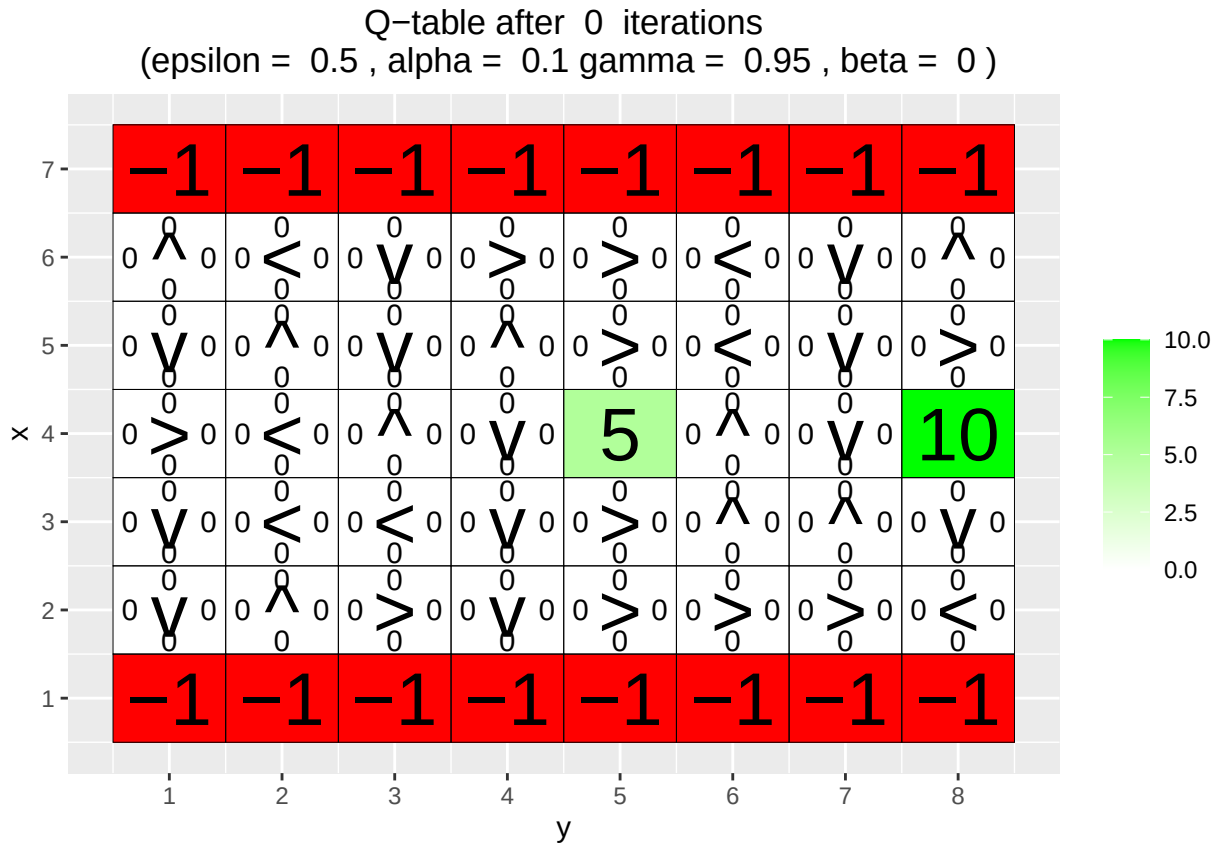
- increase exploration

- Reducing the learning rate

2.3 Environment B

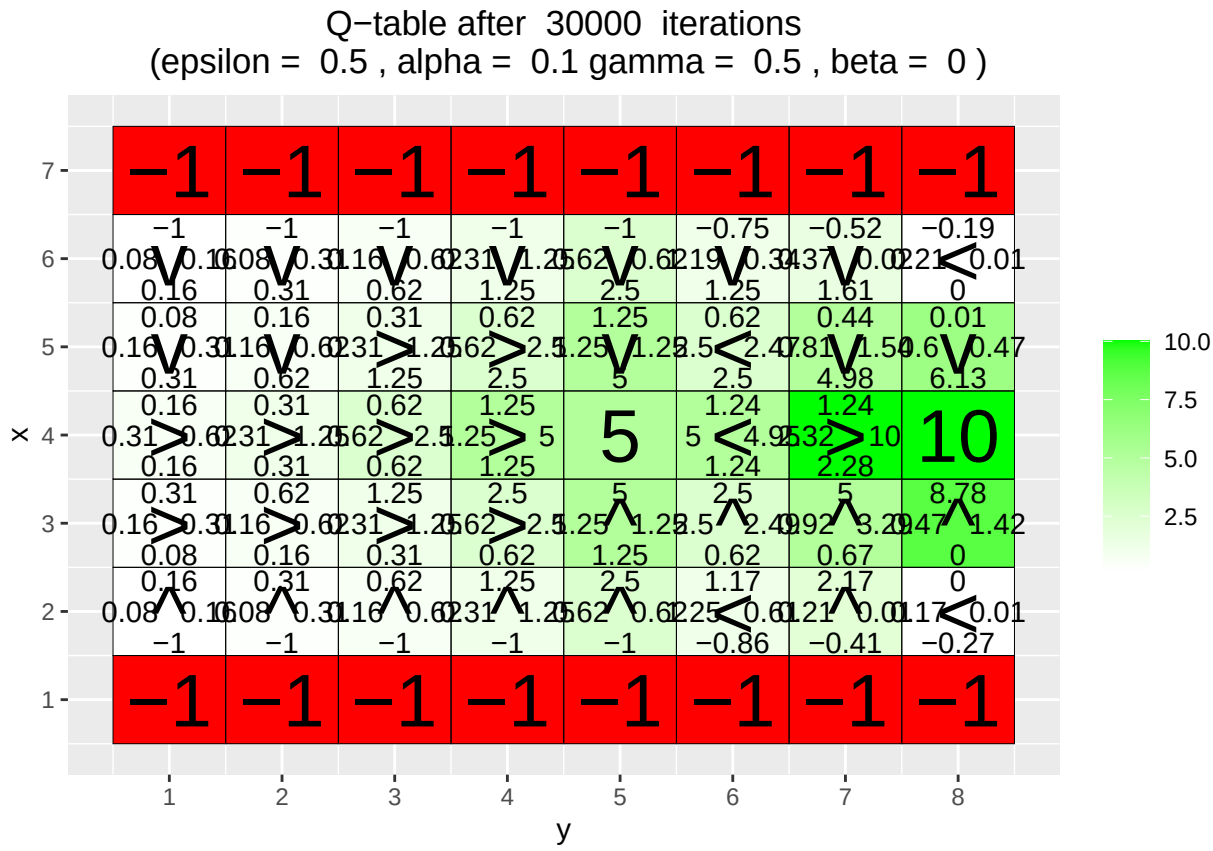
Investigate how the parameters epsilon and gamma affect the learned policy in a 7×8 environment with negative and positive rewards by running 30,000 episodes of Q-learning with different parameter settings.

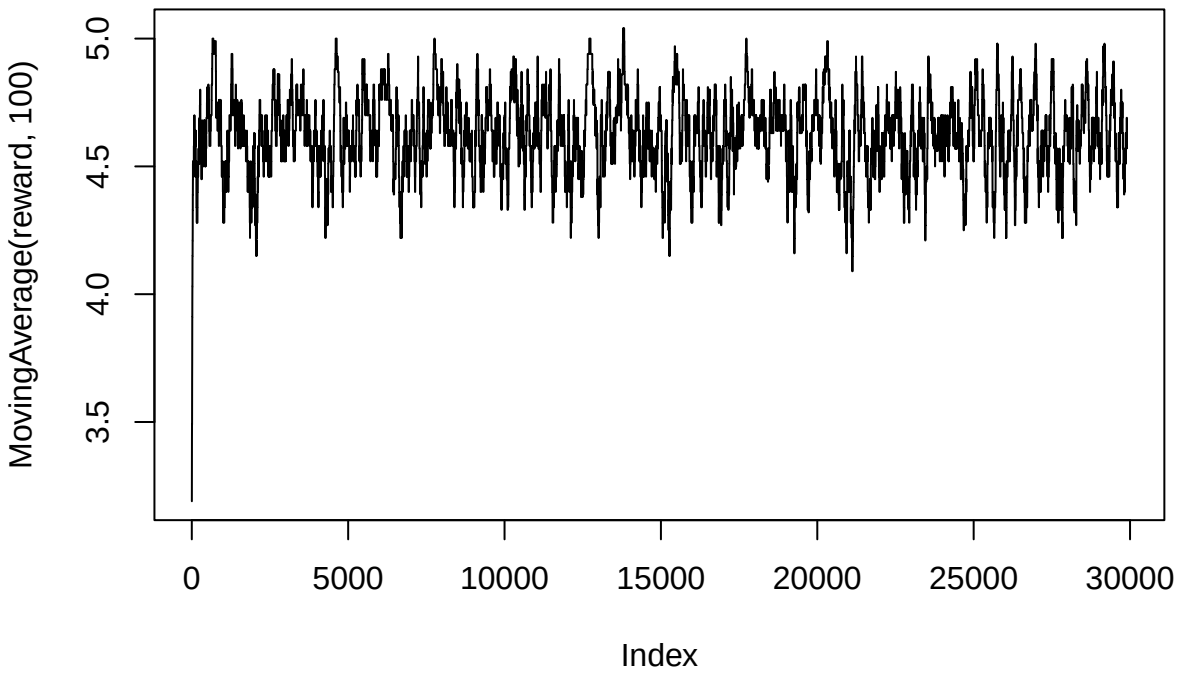
```
H <- 7
W <- 8

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()
```

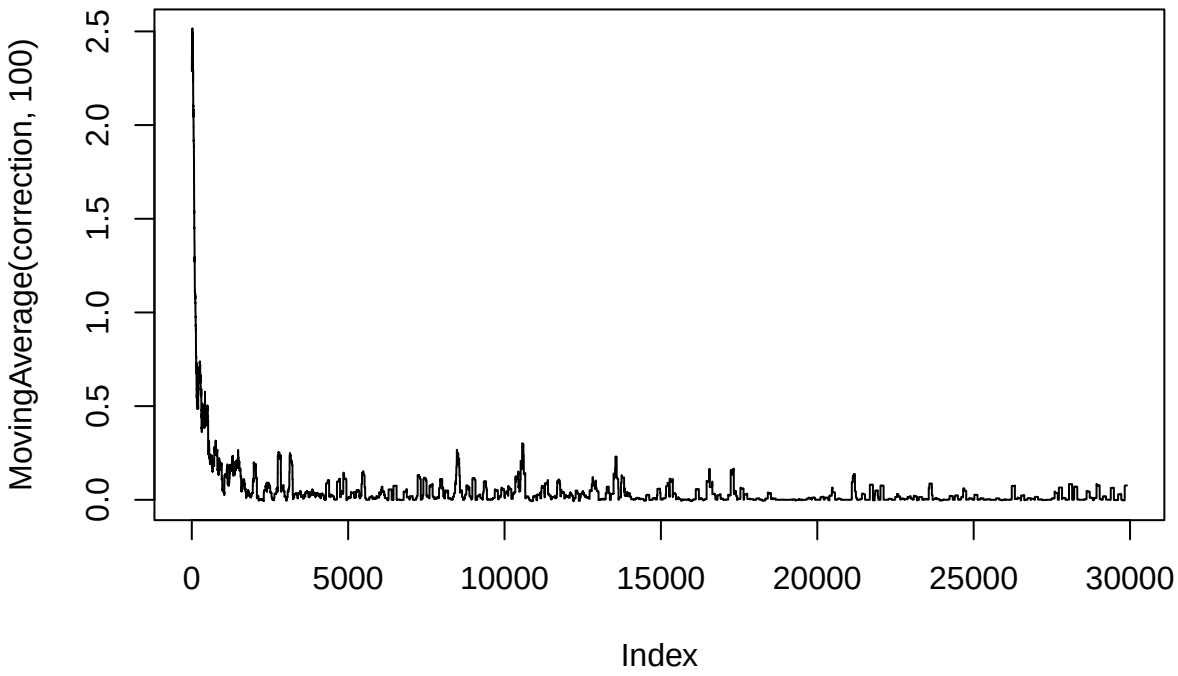


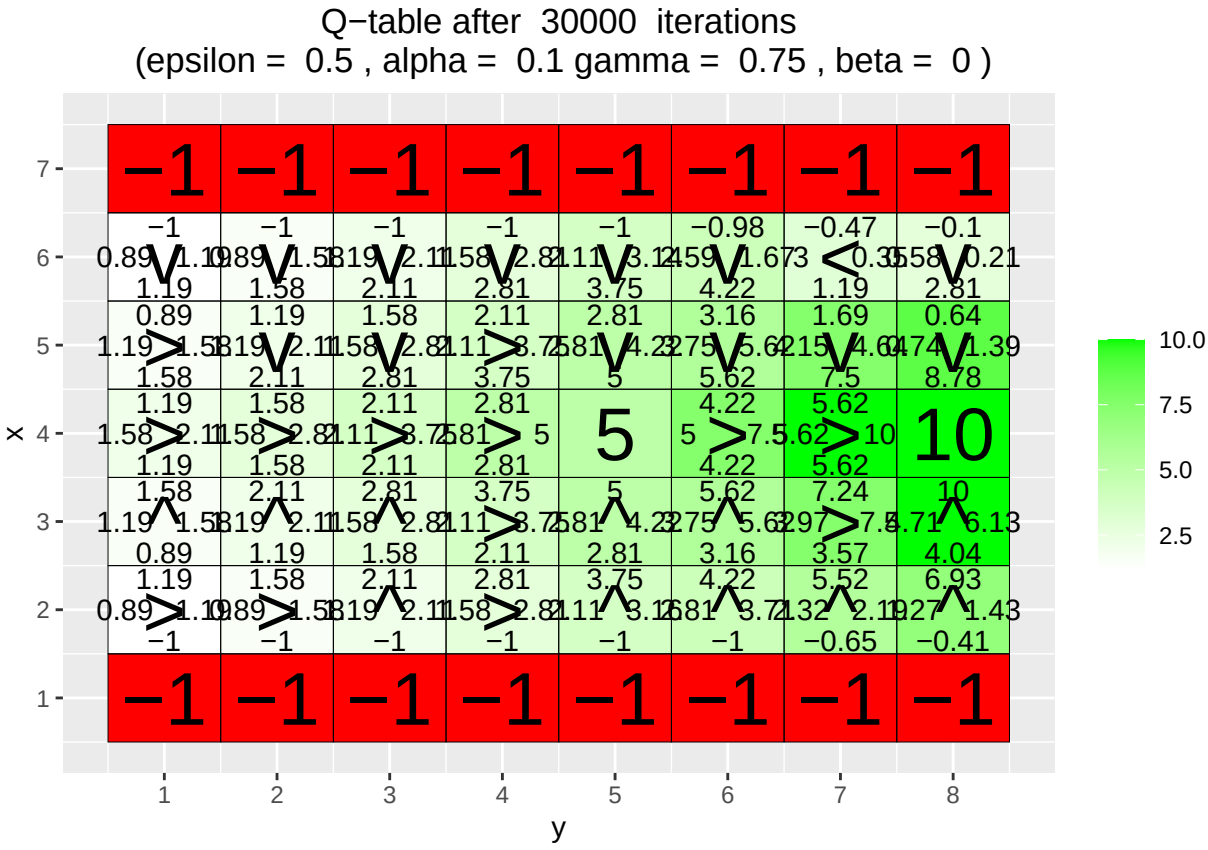

```
MovingAverage <- function(x, n){

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

  return (rsum)
}
```

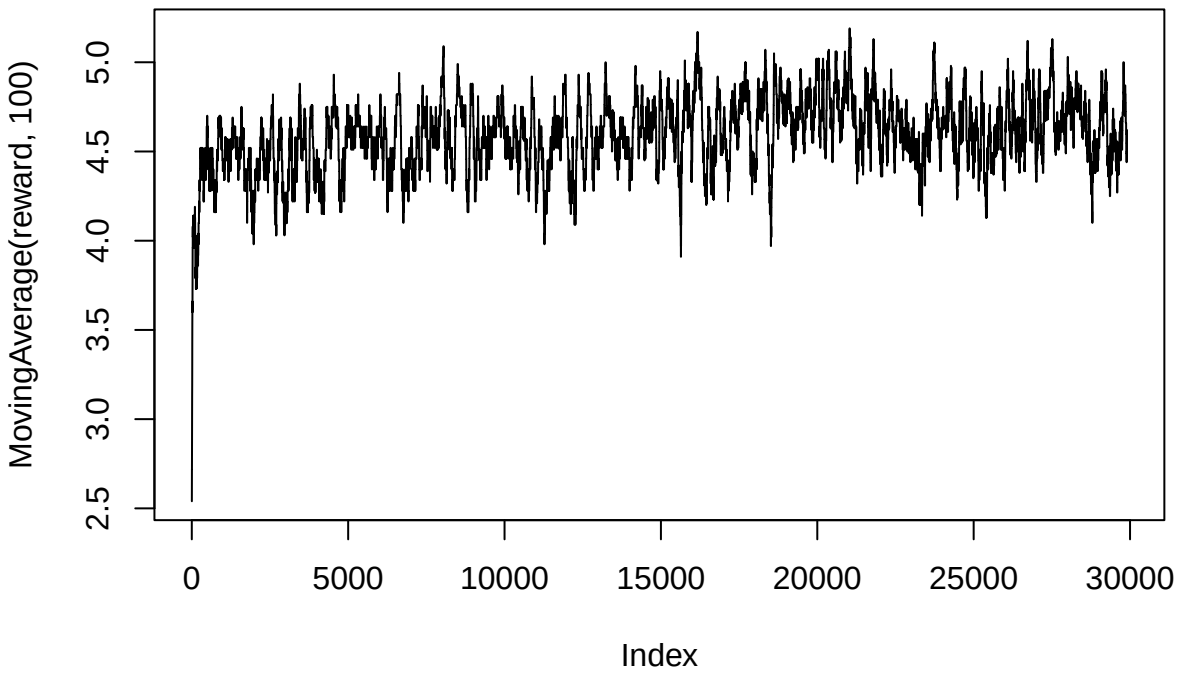
```
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    # This is for epsilon = 0.5 standard
    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")
}
```

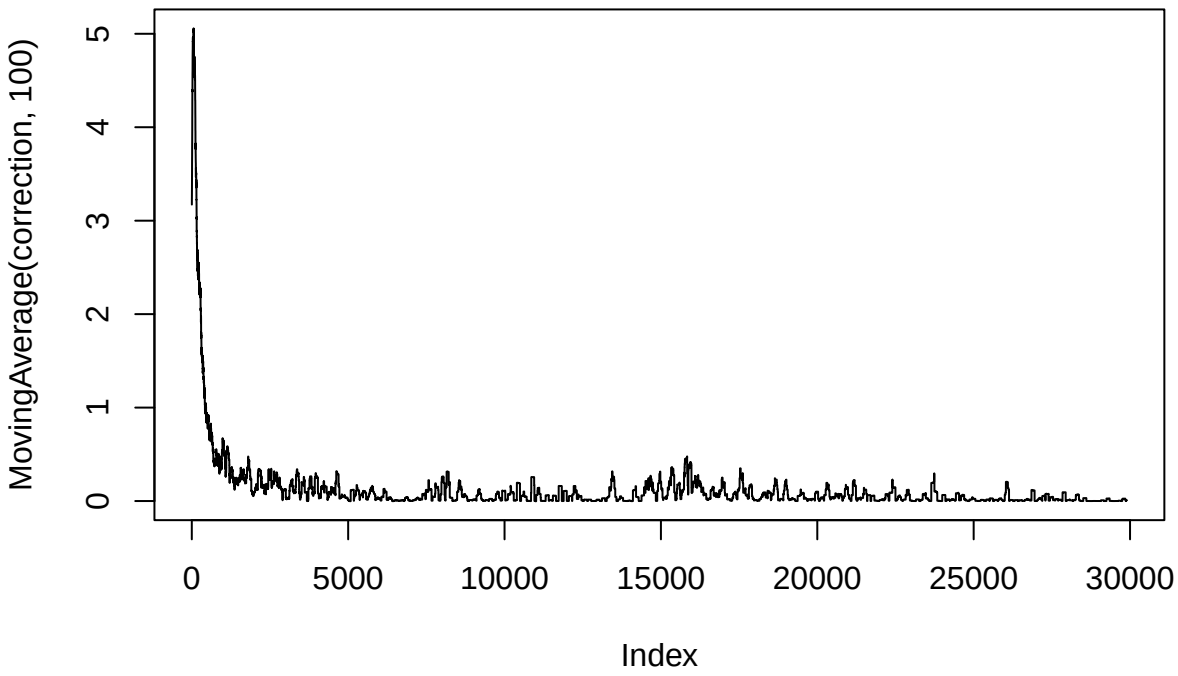


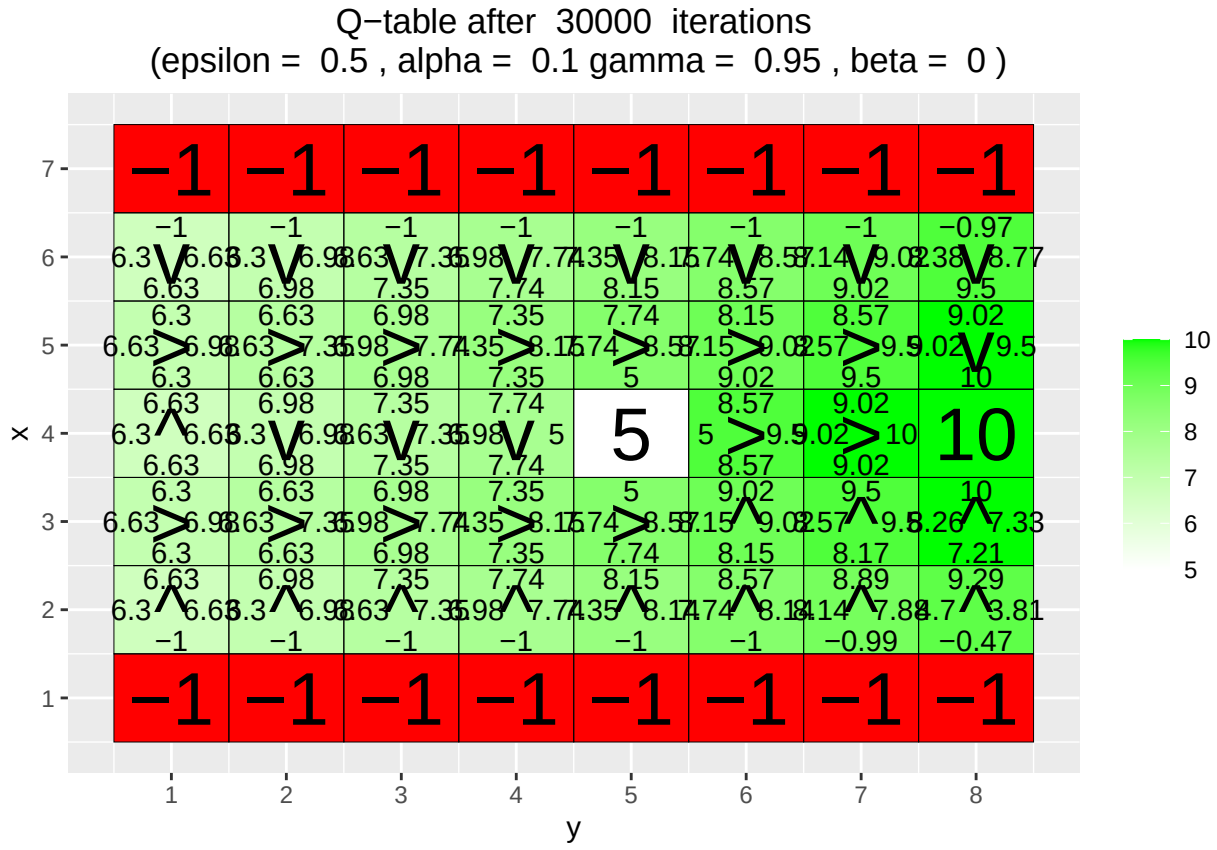

MovingAverage(reward, 100)
3.5
4.0
4.5
5.0
0
5000
10000
15000
20000
25000
30000
Index

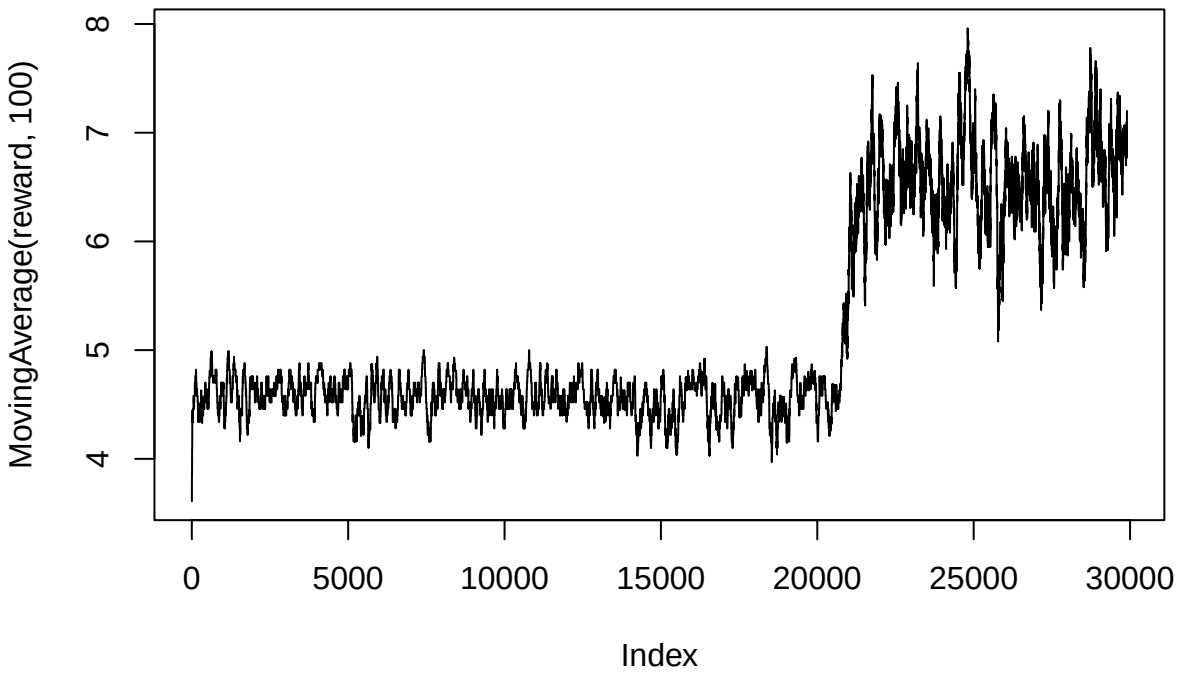

MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
2.5
0
5000
10000
15000
20000
25000
30000
Index

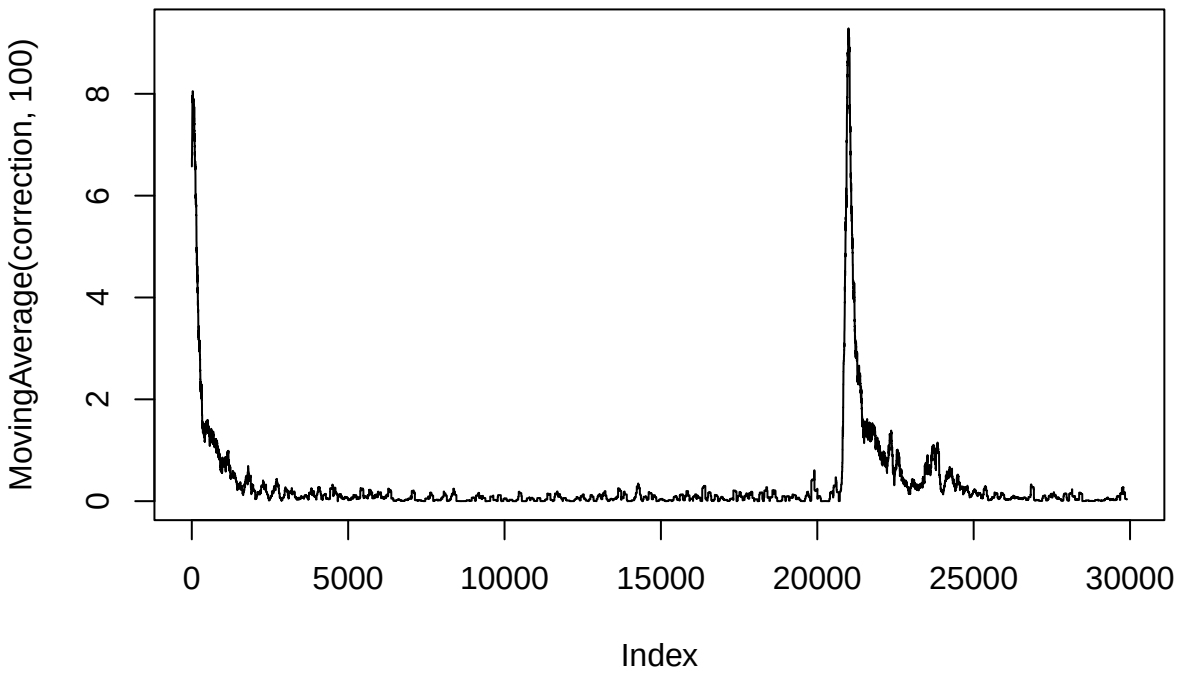

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5


MovingAverage(reward, 100)
2.5
3.0
3.5
4.0
4.5
5.0
0
5000
10000
15000
20000
25000
30000
Index

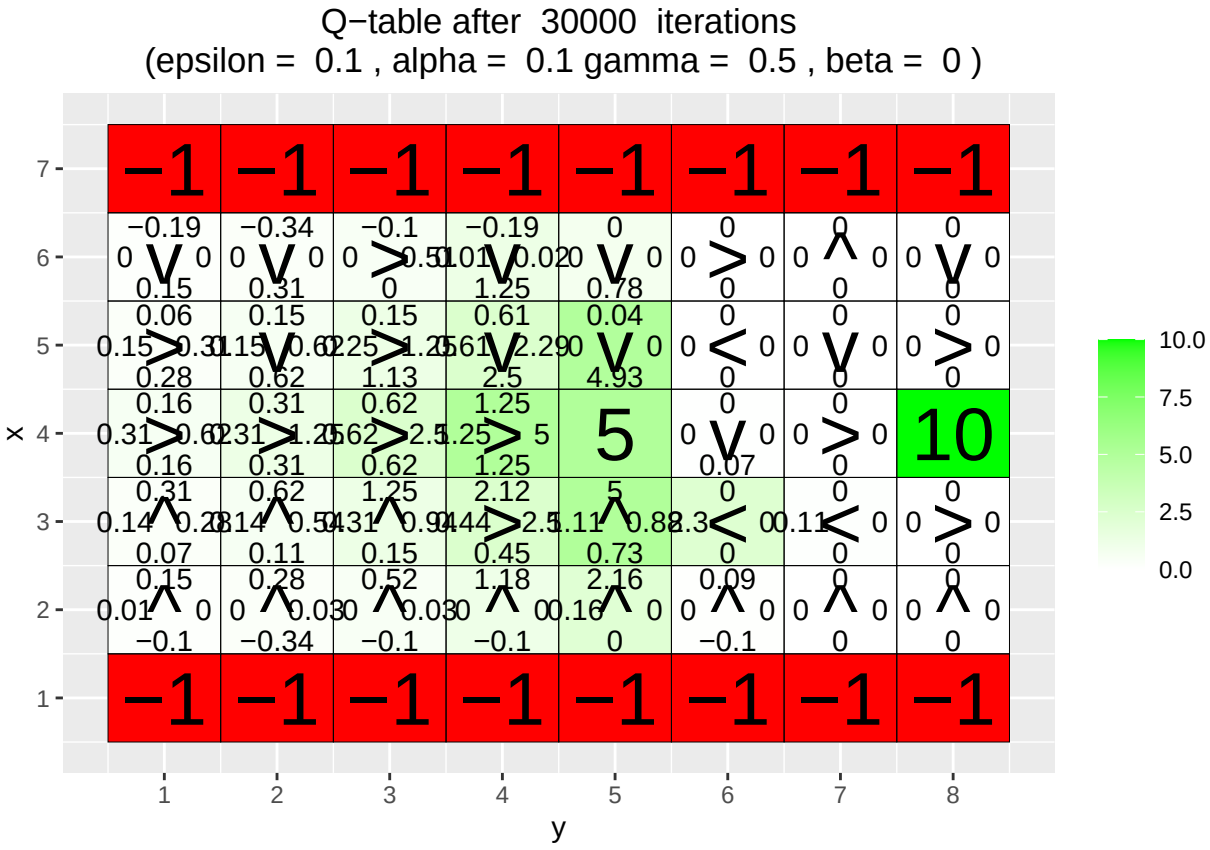

MovingAverage(correction, 100)
0
1
2
3
4
5
0
5000
10000
15000
20000
25000
30000
Index

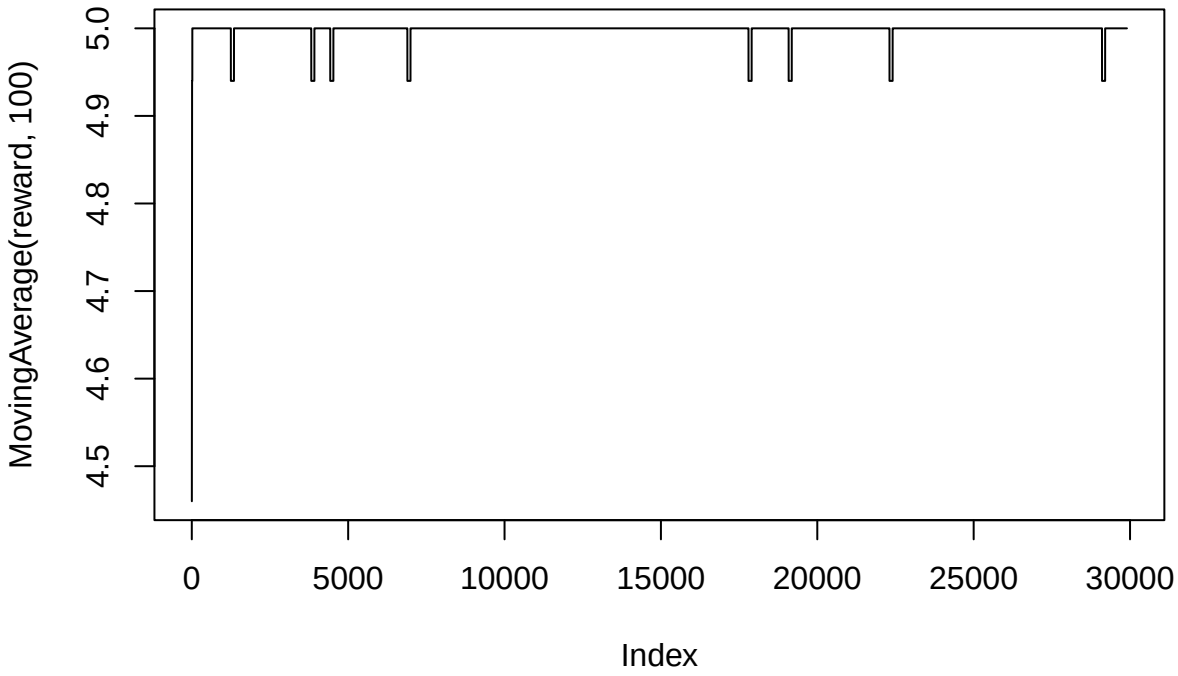

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10
9
8
7
6
5

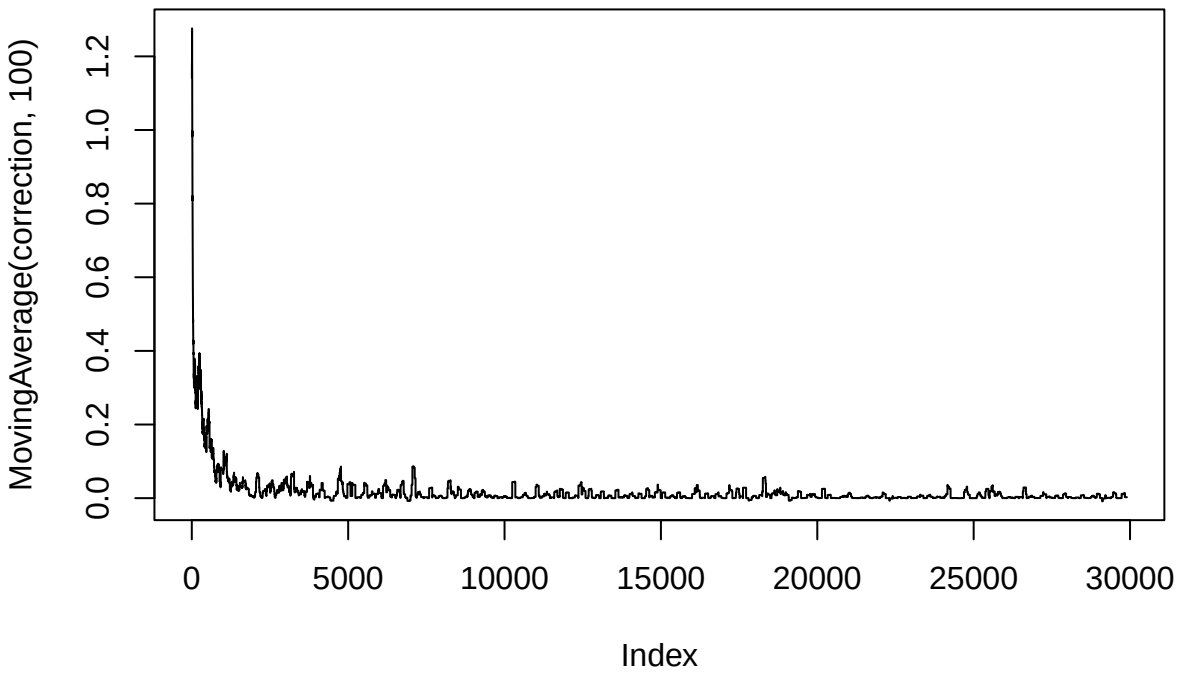

MovingAverage(reward, 100)
4
5
6
7
8
0
5000
10000
15000
20000
25000
30000
Index

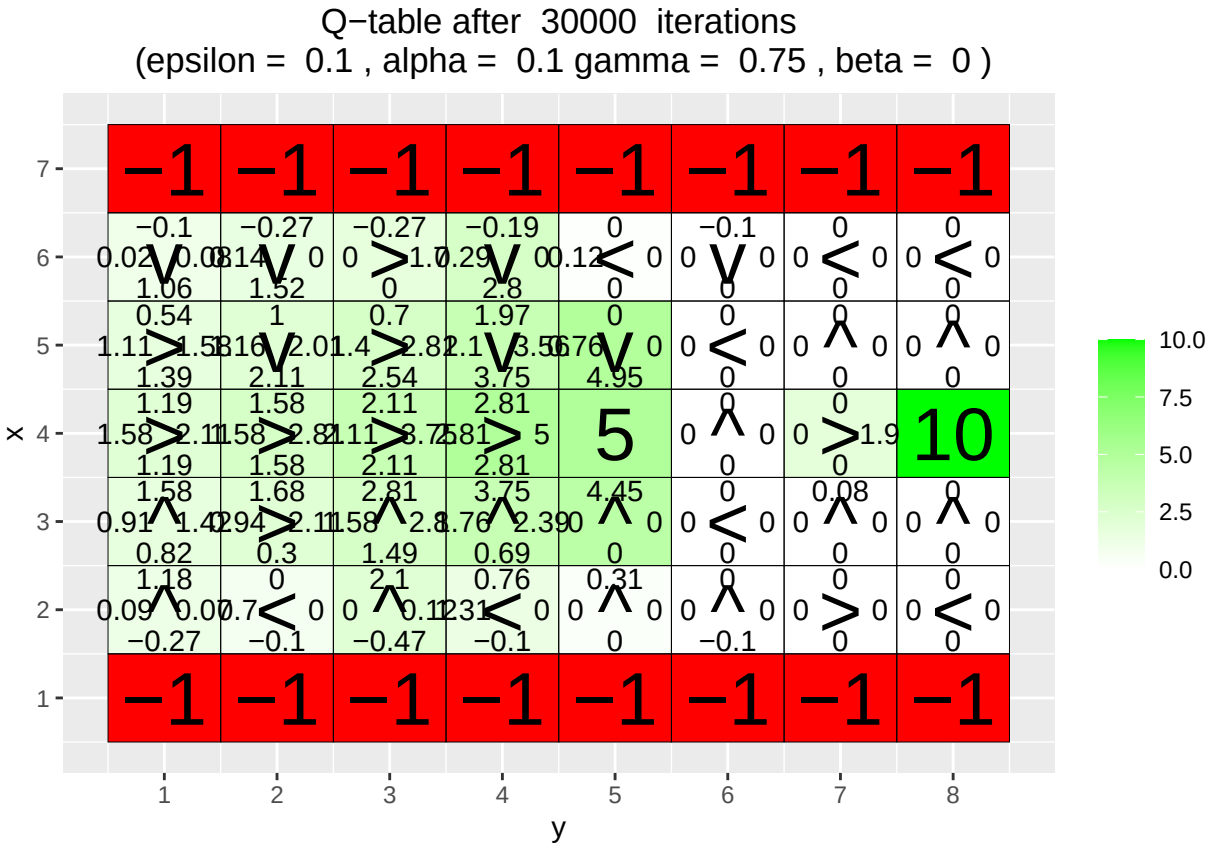


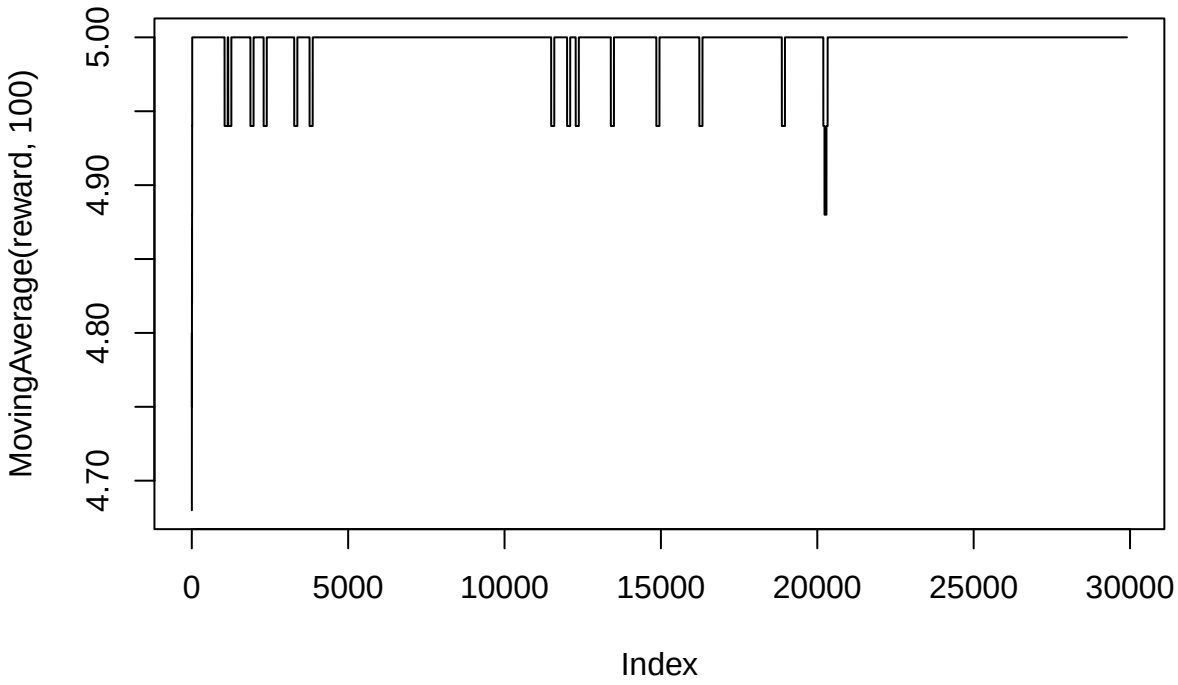

```
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    # This is epsilon 0.1
    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")
}
```

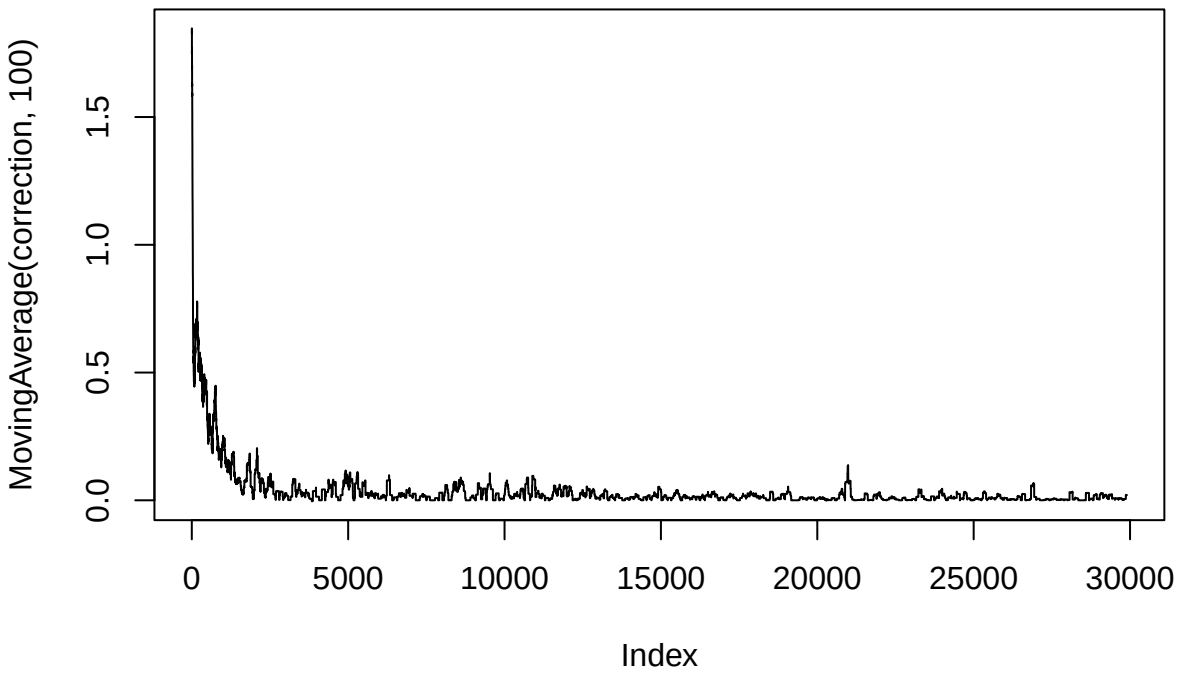

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0

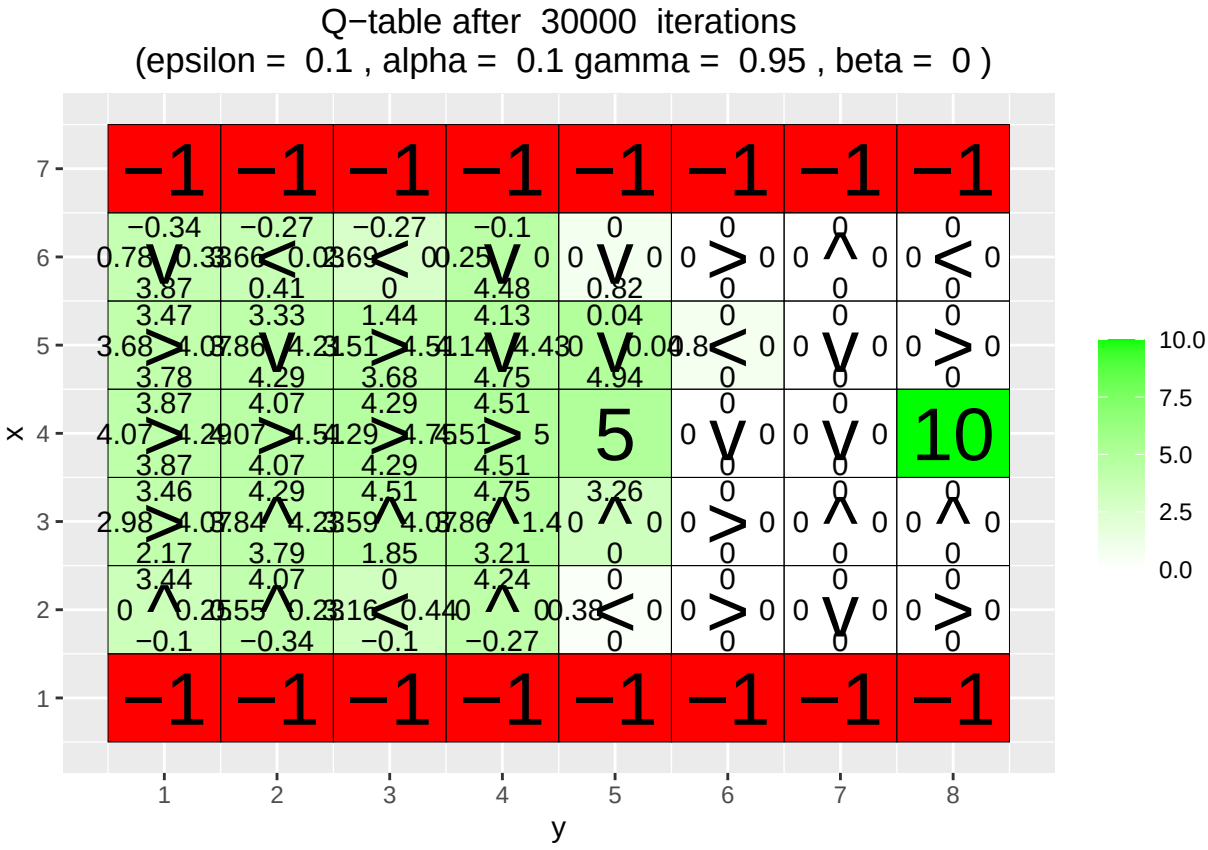

MovingAverage(reward, 100)
4.5
4.6
4.7
4.8
4.9
5.0
0
5000
10000
15000
20000
25000
30000
Index

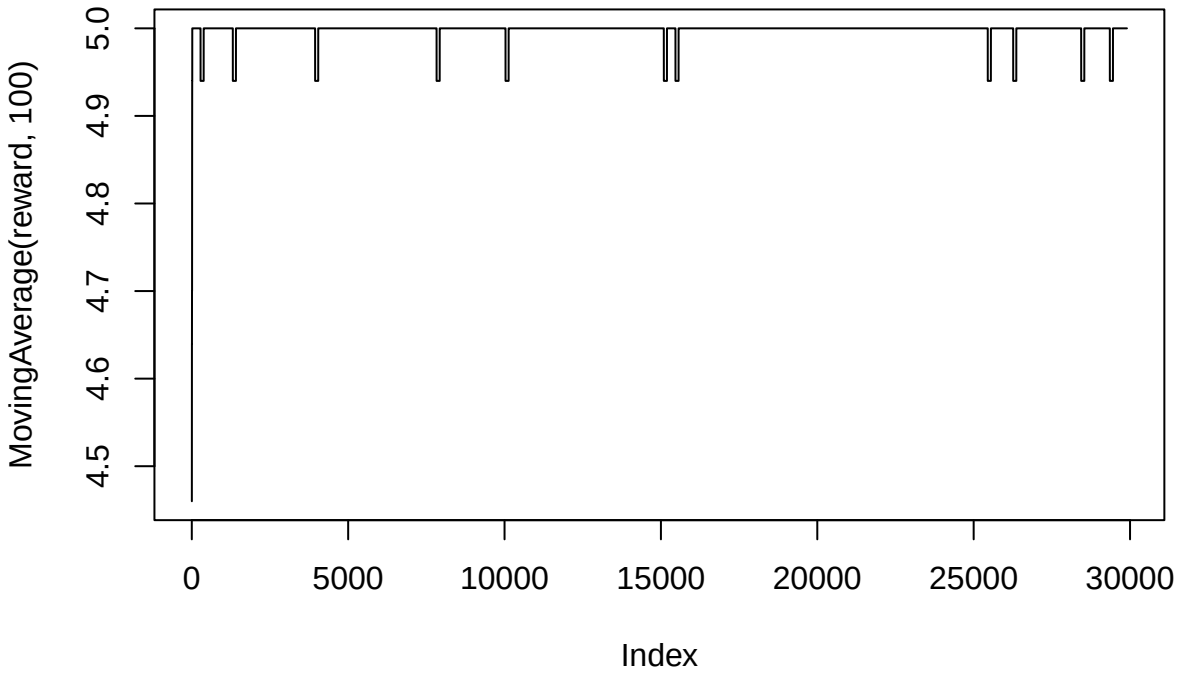

MovingAverage(correction, 100)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
5000
10000
15000
20000
25000
30000
Index

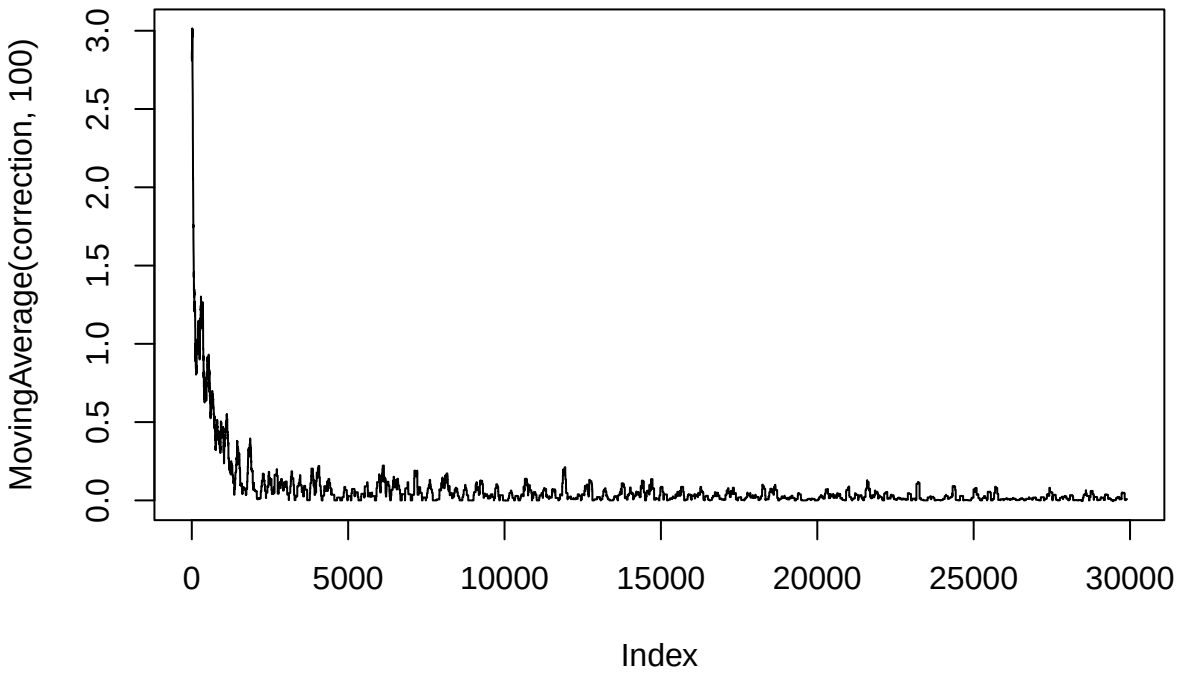

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
4.70
4.80
4.90
5.00
0
5000
10000
15000
20000
25000
30000
Index


MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
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)
4.5
4.6
4.7
4.8
4.9
5.0
0
5000
10000
15000
20000
25000
30000
Index



Your task is to investigate how the epsilon and gamma parameters affect the learned policy by running

My observations are:

epsilon = 0.5

gamma = 0.5: It chooses 5 quite often since the discount factor gamma is low meaning that it prioritizes future rewards low

gamma = 0.75: Most often chooses 10 but sometimes 5

gamma = 0.95: results in that it learns to avoid 5 reward and go for 10 reward

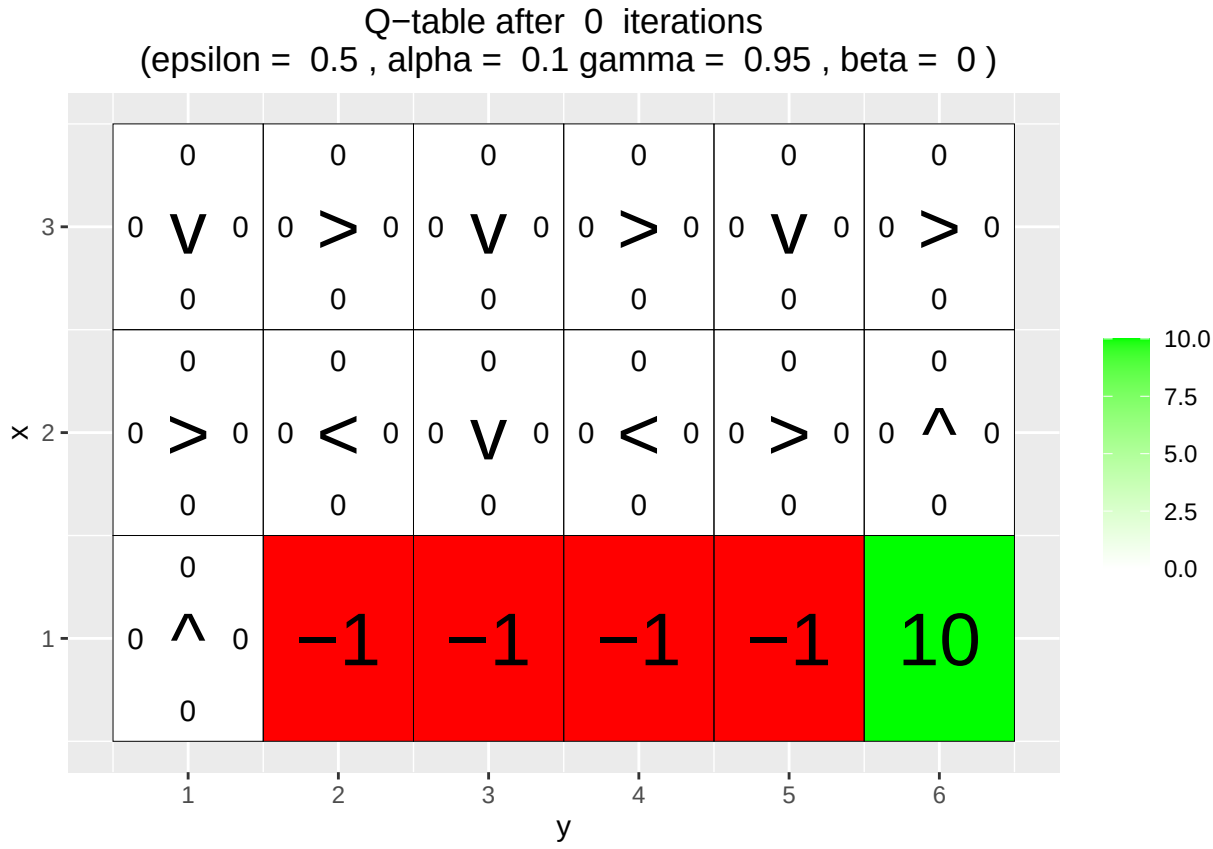
With epsilon = 0.1 the agent doesn't explore as much and subsequently doesn't find the 10 reward. It only finds 5 reward.

2.4 Environment C

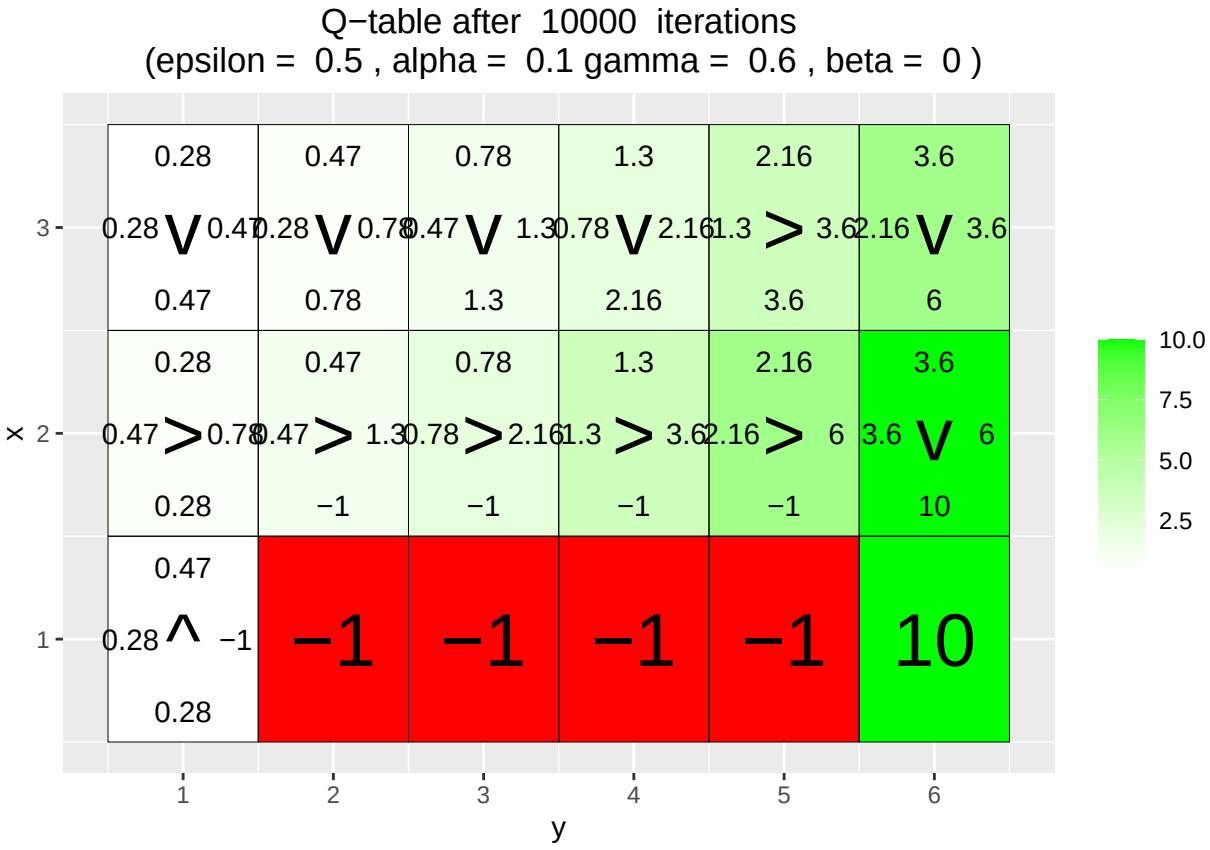
Explore how the beta parameter affects the learned policy in a 3×6 environment by running 10,000 episodes of Q-learning with different beta values and analyzing the outcomes.

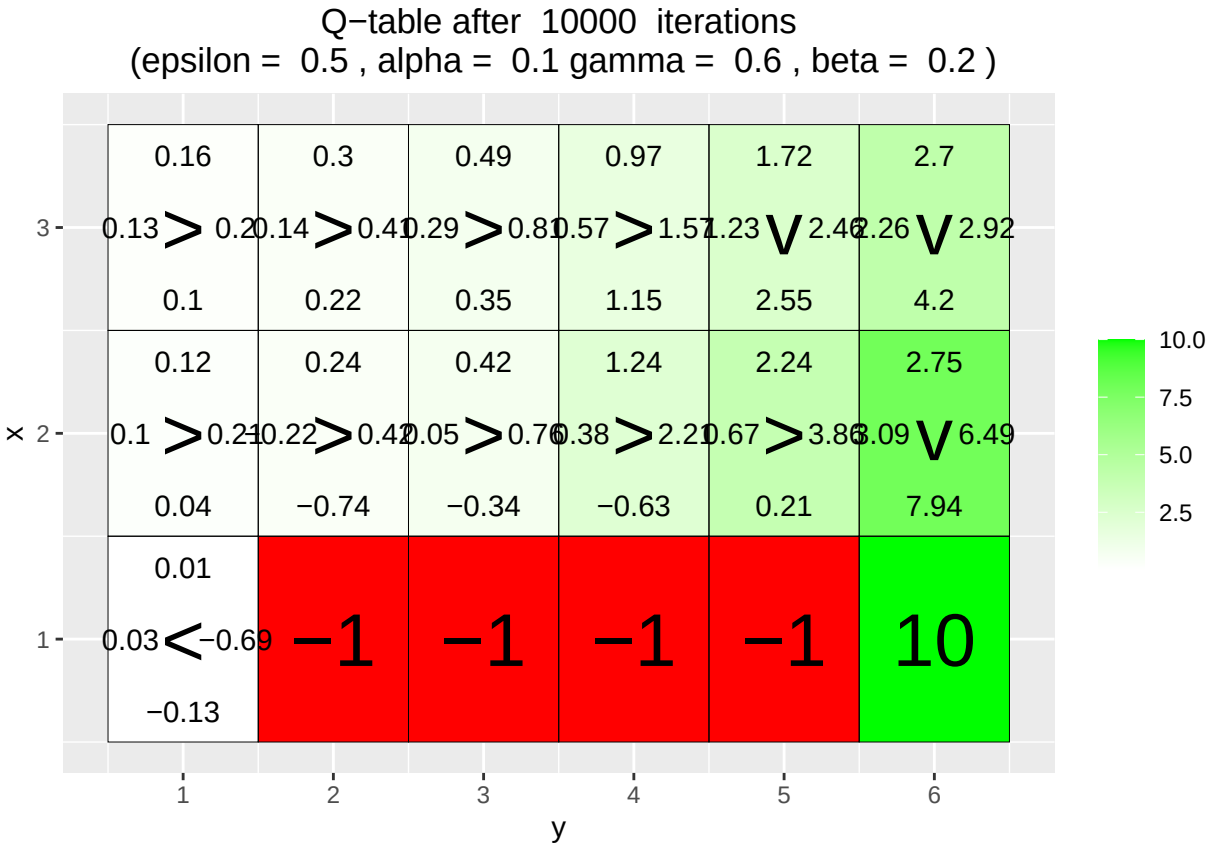
```
H <- 3
W <- 6

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

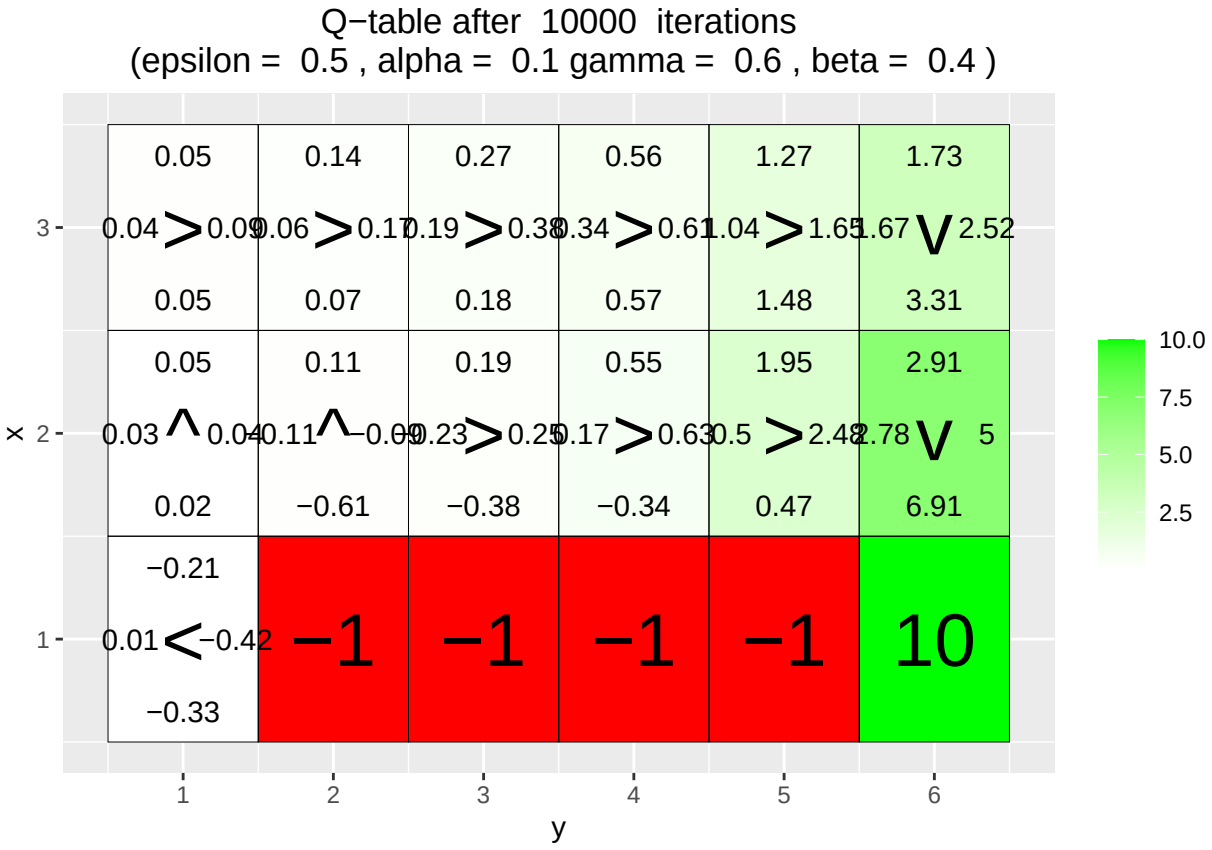
```
q_table <- array(0,dim = c(H,W,4))

vis_environment()
```

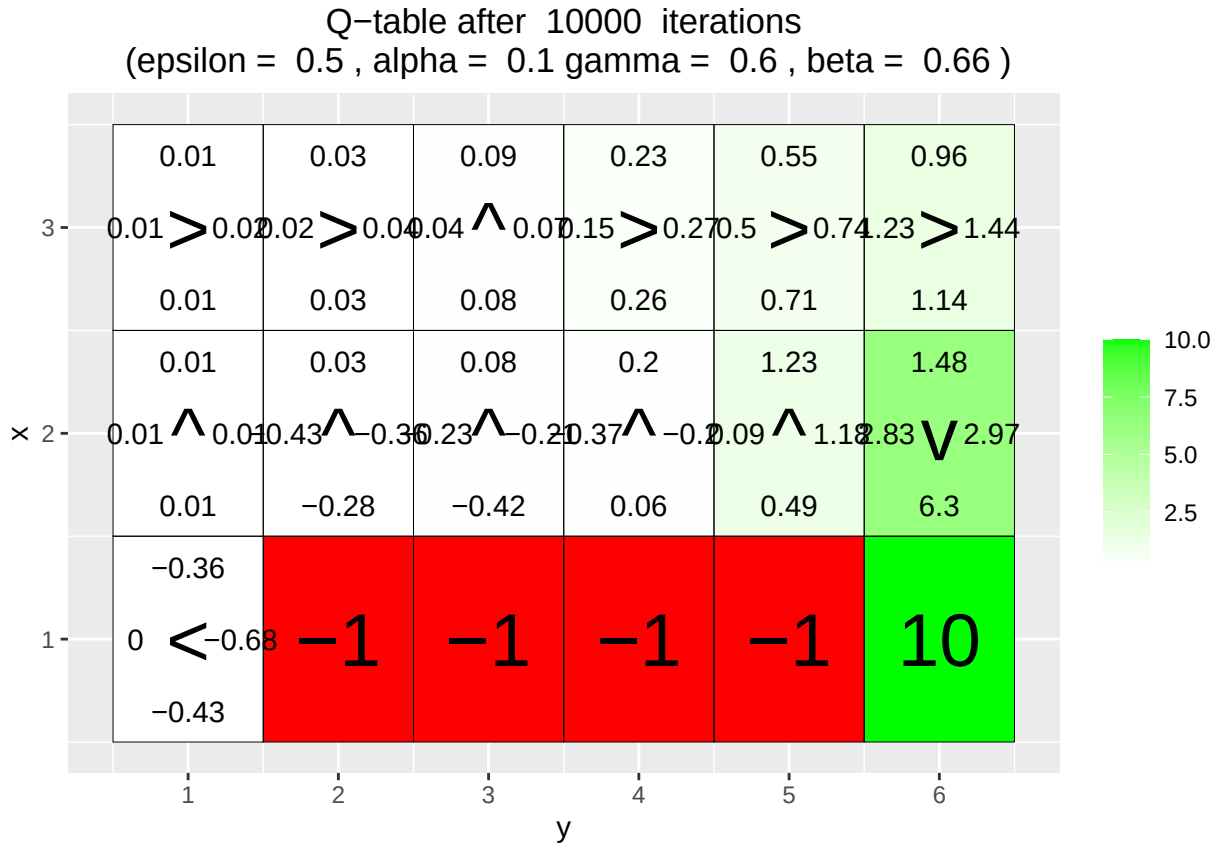



```
for(j in c(0,0.2,0.4,0.66)){
  q_table <- array(0,dim = c(H,W,4))

  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
0.28 V 0.47
0.47
0.47
0.28 V 0.78
0.78
0.78
0.47 V 1.3
1.3
1.3
0.78 V 2.16
2.16
2.16
1.3 > 3.6
3.6
3.6
2.16 V 3.6
6
0.28
0.47 > 0.78
0.28
0.47
0.47 > 1.3
−1
0.78
0.78 > 2.16
−1
1.3
1.3 > 3.6
−1
2.16
2.16 > 6
−1
3.6
3.6 V 6
10
0.47
0.28 ^ −1
0.28
−1
−1
−1
−1
10
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)
0.16
0.13 > 0.2
0.1
0.3
0.14 > 0.41
0.22
0.49
0.29 > 0.81
0.35
0.97
0.57 > 1.57
1.15
1.72
1.23 V 2.46
2.55
2.7
2.26 V 2.92
4.2
0.12
0.1 > 0.21
0.04
0.24
0.22 > 0.42
−0.74
0.42
0.05 > 0.76
−0.34
1.24
0.38 > 2.21
−0.63
2.24
0.67 > 3.86
0.21
2.75
3.09 V 6.49
7.94
0.01
0.03 < −0.69
−0.13
−1
−1
−1
−1
10
x
y
1
2
3
4
5
6
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
0.14
0.27
0.56
1.27
1.73
0.04 > 0.09
0.06 > 0.17
0.19 > 0.38
0.34 > 0.61
1.04 > 1.65
1.67 V 2.52
0.05
0.07
0.18
0.57
1.48
3.31
0.05
0.11
0.19
0.55
1.95
2.91
0.03 ^ 0.04
0.11 ^ −0.09
0.23 > 0.25
0.17 > 0.63
0.5 > 2.48
2.78 V 5
0.02
−0.61
−0.38
−0.34
0.47
6.91
−0.21
0.01 < −0.42
−1
−1
−1
−1
10
−0.33
x
y
1
2
3
4
5
6
10.0
7.5
5.0
2.5

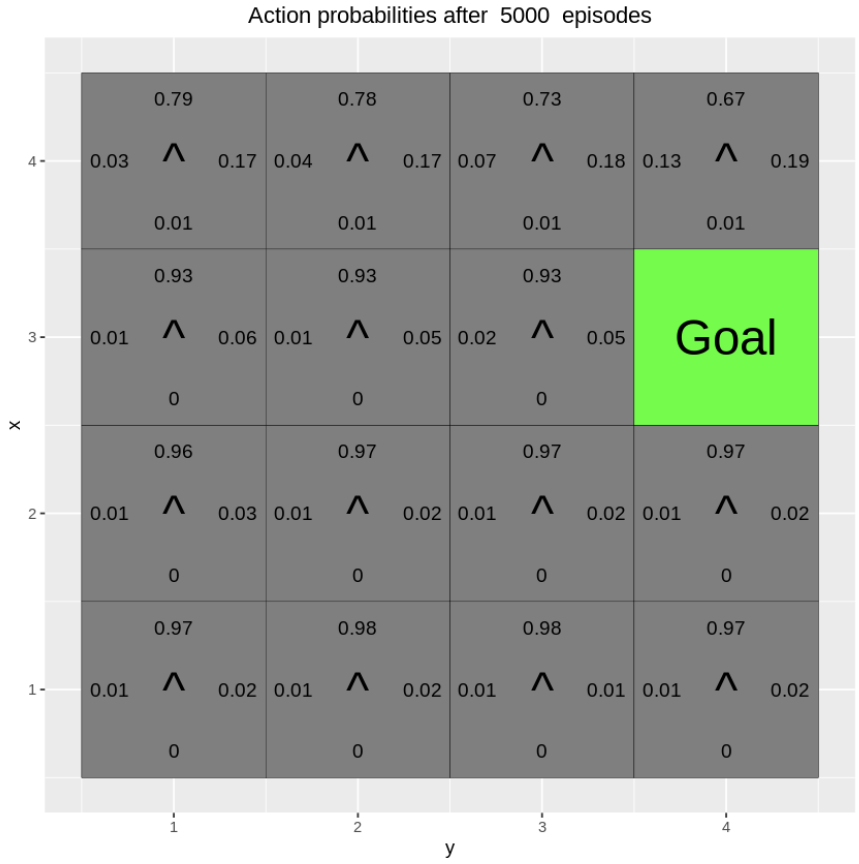


Here, Beta determines the probability that the agent will slip. A higher beta results in the agent avoiding the -1 wall with greater margin since it might slip.

2.5 REINFORCE

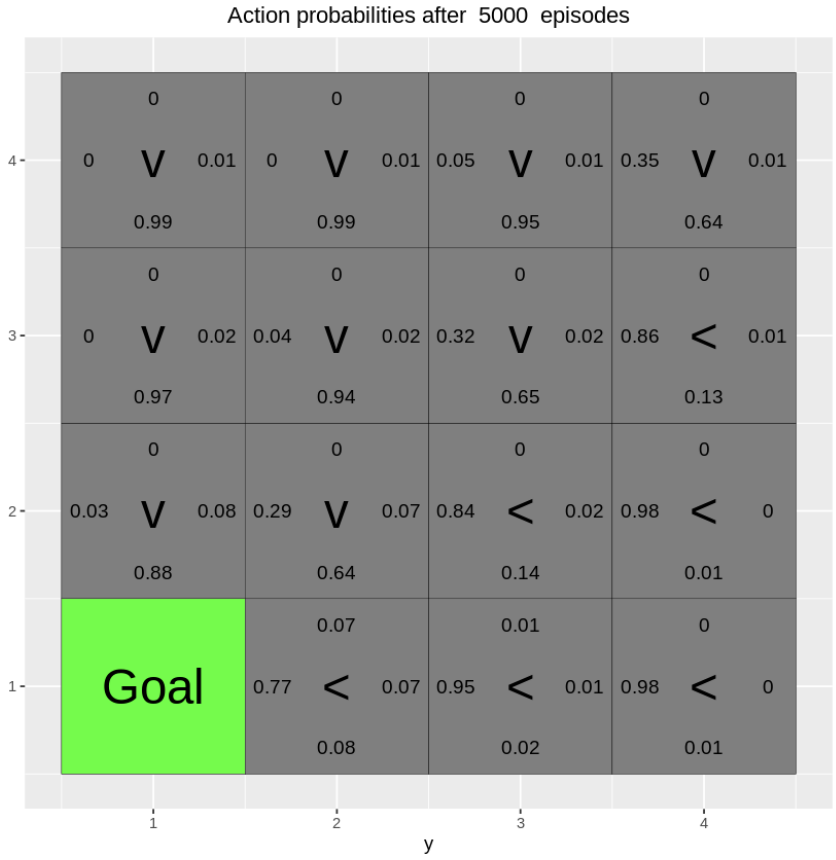
Study the provided REINFORCE algorithm for a 4×4 grid-world, where the agent must learn to navigate to a random goal position, and analyze the code and results without needing to run it.

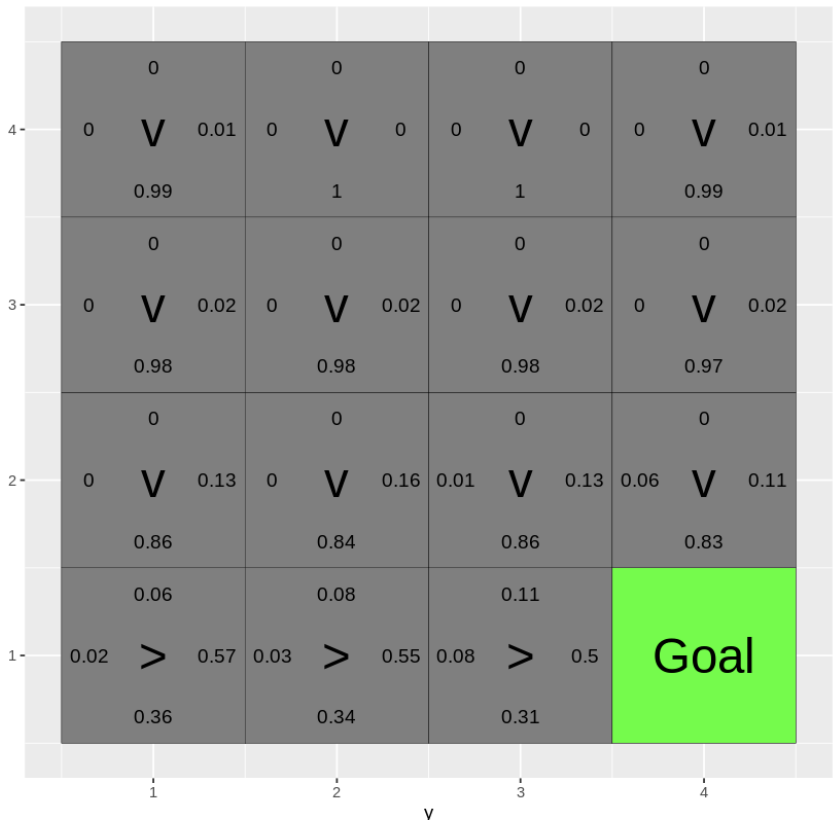
Plot 1 and 2


Action probabilities after 5000 episodes
0.79
0.03
^
0.17
0.01
0.78
0.04
^
0.17
0.01
0.73
0.07
^
0.18
0.01
0.67
0.13
^
0.19
0.01
0.93
0.01
^
0.06
0
0.93
0.01
^
0.05
0
0.93
0.02
^
0.05
0
Goal
0.96
0.01
^
0.03
0
0.97
0.01
^
0.02
0
0.97
0.01
^
0.02
0
0.97
0.01
^
0.02
0
0.97
0.01
^
0.02
0
0.98
0.01
^
0.02
0
0.98
0.01
^
0.01
0
0.97
0.01
^
0.02
0
x
y
1
2
3
4


Action probabilities after 5000 episodes
0.21
0.26
>
0.49
0.04
0.18
0.35
>
0.43
0.03
0.13
0.5
<
0.34
0.03
0.08
0.63
<
0.27
0.02
0.63
0.08
^
0.26
0.03
0.59
0.12
^
0.26
0.03
0.51
0.2
^
0.26
0.03
0.37
0.33
^
0.26
0.03
0.86
0.03
^
0.09
0.01
0.86
0.04
^
0.09
0.02
Goal
0.75
0.1
^
0.12
0.02
0.91
0.02
^
0.06
0.01
0.92
0.03
^
0.05
0.01
0.91
0.03
^
0.05
0.01
0.88
0.04
^
0.06
0.02
x
y
1
2
3
4

Plot 3 and 4


Action probabilities after 5000 episodes
0 0 0 0
0 V 0.01 0 V 0.01 0.05 V 0.01 0.35 V 0.01
0.99 0.99 0.95 0.64
0 0 0 0
0 V 0.02 0.04 V 0.02 0.32 V 0.02 0.86 < 0.01
0.97 0.94 0.65 0.13
0 0 0 0
0.03 V 0.08 0.29 V 0.07 0.84 < 0.02 0.98 < 0
0.88 0.64 0.14 0.01
0.07 0.01 0
Goal 0.77 < 0.07 0.95 < 0.01 0.98 < 0
0.08 0.02 0.01
4
3
2
1
1 2 3 4
y


Action probabilities after 5000 episodes
0 0 0 0
0 V 0.01 0 V 0 0 V 0 0 V 0.01
0.99 1 1 0.99
0 0 0 0
0 V 0.02 0 V 0.02 0 V 0.02 0 V 0.02
0.98 0.98 0.98 0.97
0 0 0 0
0 V 0.13 0 V 0.16 0.01 V 0.13 0.06 V 0.11
0.86 0.84 0.86 0.83
0.06 0.08 0.11
0.02 > 0.57 0.03 > 0.55 0.08 > 0.5 Goal
0.36 0.34 0.31
4
3
2
1
1 2 3 4
y

2.6 Environment D

Train a REINFORCE agent on eight goal positions and validate it on the remaining eight positions, then answer questions on whether the agent learned a good policy and whether Q-learning could solve the task.

Has the agent learned a good policy? Why / Why not ?

Yes, the agent has learned relatively good policy (plot 3 and 4). The policy generalizes well during validation since the training goals were spread across the grid, allowing the agent to learn a flexible policy.

Could you have used the Q-learning algorithm to solve this task?

Yes, although REINFORCE is much more suitable. Q-learning could solve this task by incorporating the goal coordinates into the state representation. However, Q-learning might struggle with generalizing as effectively as REINFORCE.

2.7 Environment E

Analyze the agent's performance when trained with goals from the top row and validated with goals from lower rows, and compare the results with environment D, explaining any differences.

Has the agent learned a good policy? Why / Why not ?

No, the agent has not learned a fully generalized policy (plot 1 and 2). It has overfitted to the training where the goals were located in the top row making the policy arrows pointing upward.

If the results obtained for environments D and E differ, explain why.

The difference arises from the diversity of the training goals. In Environment D, the training goals were spread across the grid, resulting in better generalization. In Environment E, the training goals were restricted to the top row, leading to overfitting and poor generalization to new goal positions during validation.

Lab4

2.1 Implement GP Regression

Libraries

```
library(kernlab)
```

```
##
## Attaching package: 'kernlab'

## The following object is masked from 'package:ggplot2':
##
##     alpha
```

```
library(AtmRay)
rm(list =ls())
```

Task 1: Write Function for Posterior GP Simulation Implement a function named `posteriorGP` according to algorithm 2.1 in the book to simulate from the posterior distribution using the squared exponential kernel.

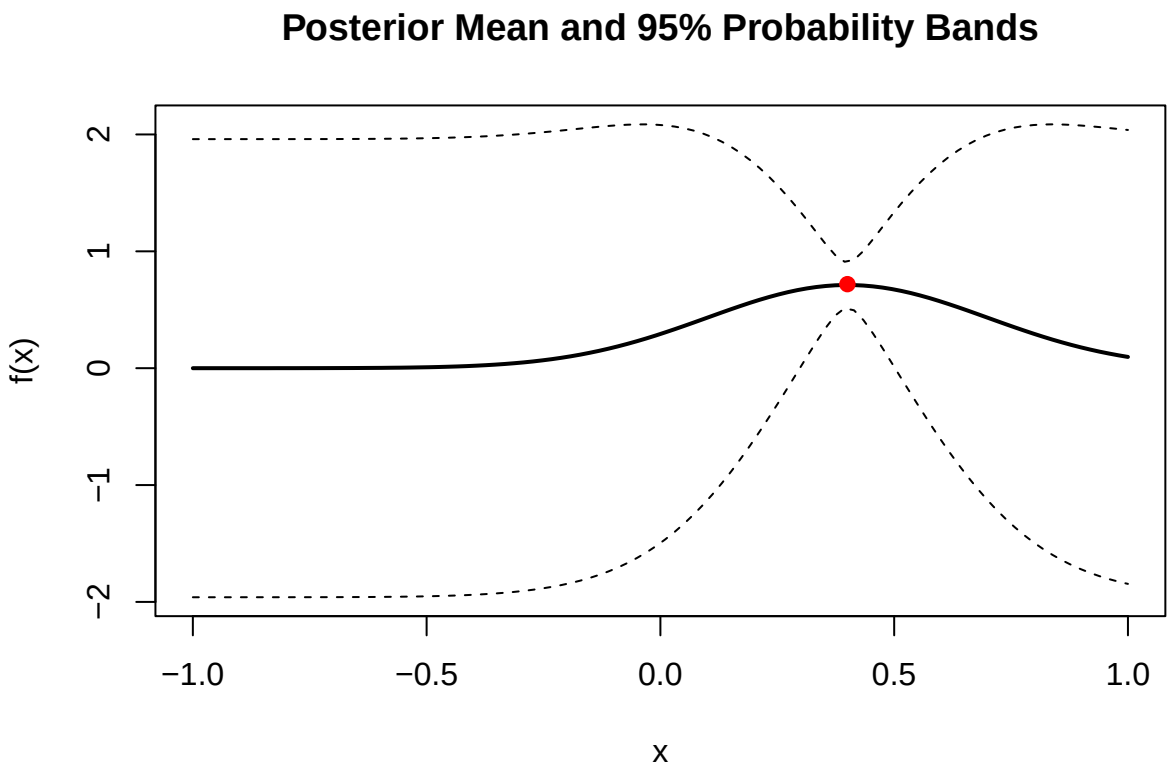
```
# Squared Exponential Kernel Function
SquaredExpKernel <- function(x1,x2,sigmaF=1,ell=3){
  n1 <- length(x1)
  n2 <- length(x2)
  K <- matrix(NA,n1,n2)
  for (i in 1:n2){
    K[,i] <- sigmaF^2*exp(-0.5*( (x1-x2[i])/ell)^2 )
  }
  return(K)
}

# Posterior GP Function
posteriorGP <- function(X, y, XStar, sigmaNoise, k, ...) {
  n = length(X)
  K <- k(X, X, ...)  # Compute the covariance matrix
  kStar <- k(X, XStar, ...) # Compute covariance

  # Step 2 in algo
  #---------------------
  K_y <- K + sigmaNoise^2 * diag(length(X)) # Add noise variance to diagonal
  L <- t(chol(K_y))   # Compute Cholesky decomposition, to get lower triangular L we take t()
  alpha <- solve(t(L), solve(L, y))   # Solve for alpha
  #----------------------

  # Step 4 in algo
  #---------------------
  fStar_mean <- t(kStar) %*% alpha   # Compute posterior mean
  v <- solve(L, kStar)   # Compute v = solve(L, kStar)
  #--------------------

  # Step 6 in algo
  #--------------------
  V_fStar <- k(XStar, XStar, ...) - t(v) %*% v # pred variance (cov matrix)
  #--------------------

log_marg_likelihood = -(1/2)*t(y)%*%alpha - sum(log(diag(L))) - (n/2)*log(2*pi)

  return(list(mean = fStar_mean, variance = V_fStar, log_likelihood = log_marg_likelihood))
}
```

Task 2: Update Posterior with Single Observation Let prior hyperparameters be sigmaf = 1 and l = 0.3, update with (x, y) = (0.4, 0.719), and plot the posterior mean with 95% bands.

```
# Plotting Function
plotGP <- function(XStar, res, X_train, y_train, title) {

  # Extract posterior mean and variance
  pos_mean <- res$mean
  pos_var <- diag(res$variance)
```

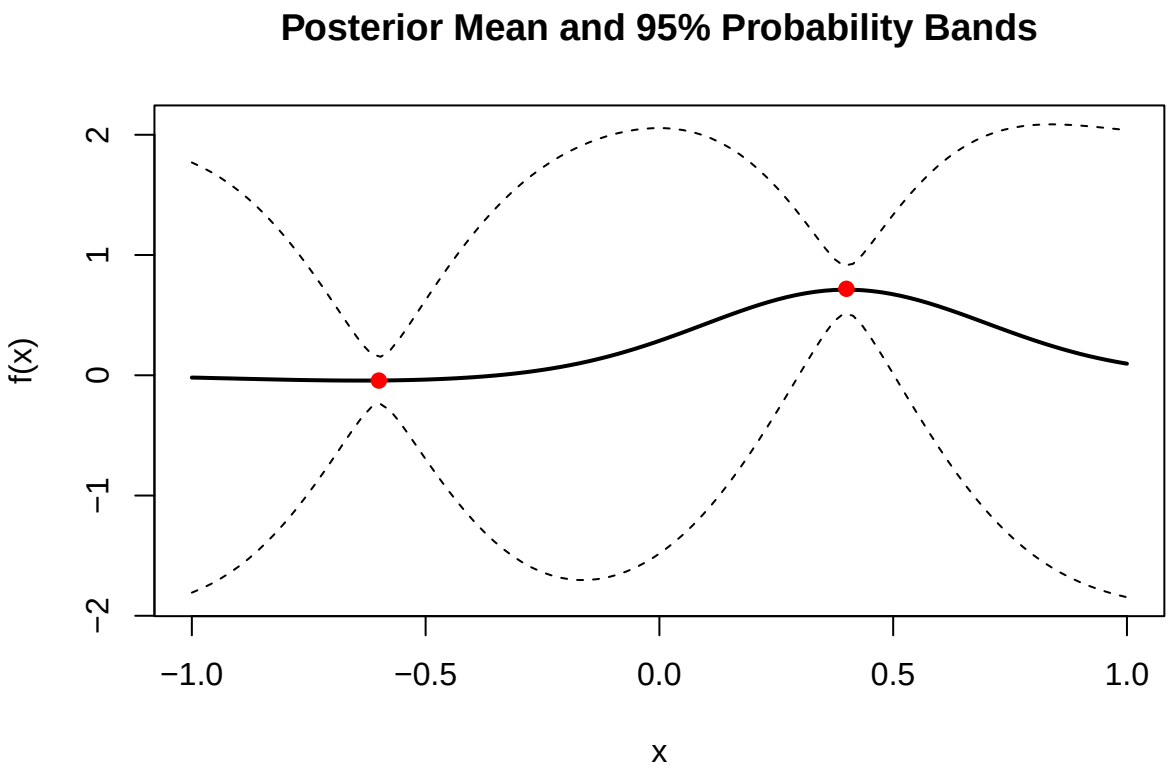
```
  # Compute 95% confidence intervals
  lower_bound <- pos_mean - 1.96 * sqrt(pos_var)
  upper_bound <- pos_mean + 1.96 * sqrt(pos_var)

  # Plot the posterior mean and 95% probability bands
  plot(XStar, pos_mean, type = "l", lwd = 2,
    ylim = range(c(lower_bound, upper_bound, y_train)),
    ylab = "f(x)", xlab = "x", main = title)

  # Add the confidence intervals
  lines(XStar, lower_bound, lty = 2)
  lines(XStar, upper_bound, lty = 2)
  # Plot the training data points
  points(X_train, y_train, pch = 19, col = "red")
}
```

```
# Single observation
X <- c(0.4)
y <- c(0.719)

# Test inputs over the interval [-1, 1]
XStar <- seq(-1, 1, length.out = 100)

# Hyperparameters
sigmaF <- 1        # sigma_f
ell <- 0.3         # length-scale l
sigmaNoise <- 0.1  # sigma_n

# Call posteriorGP
res <- posteriorGP(X, y, XStar, sigmaNoise, k = SquaredExpKernel, sigmaF = sigmaF, ell = ell)

plotGP(XStar, res, X, y, "Posterior Mean and 95% Probability Bands")
```



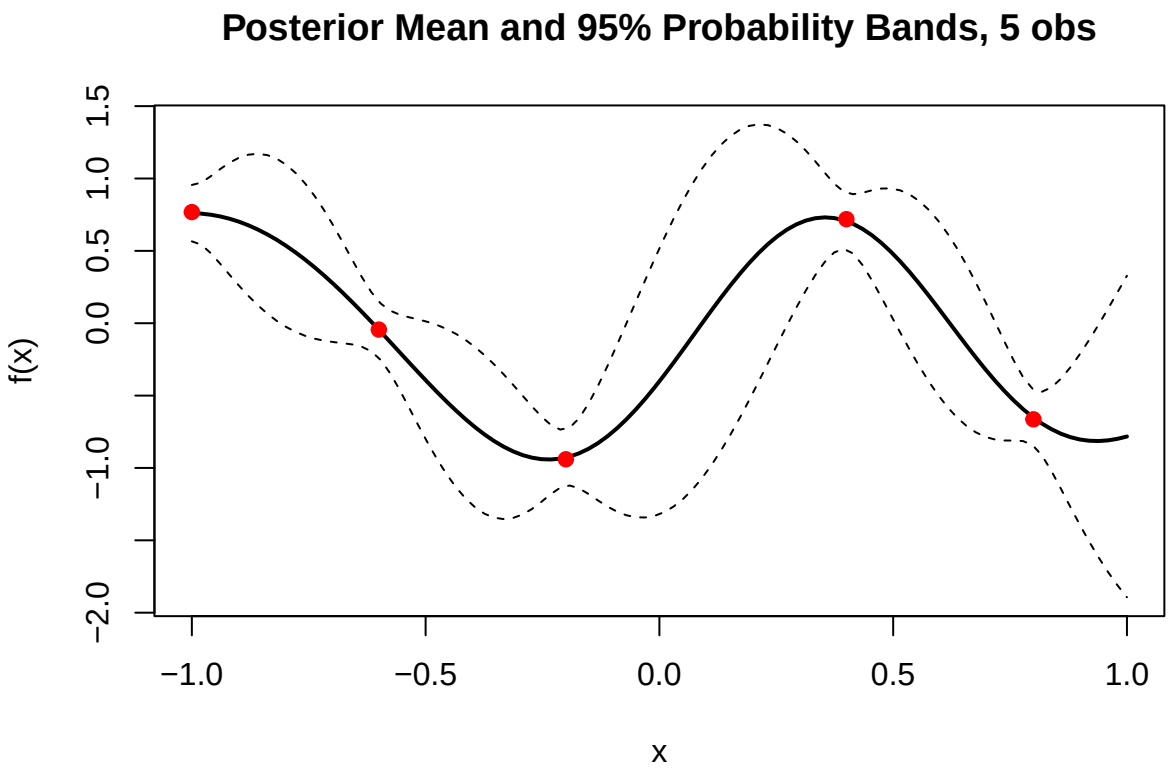
Task 3: Update with Another Observation Update posterior from Task 2 with (x, y) = (-0.6, -0.044), plot the posterior mean, and include 95% probability bands.

```
# Updated training data
X <- c(0.4, -0.6)
y <- c(0.719, -0.044)

# Same XStar and hyperparameters as in task2
res <- posteriorGP(X, y, XStar, sigmaNoise, k = SquaredExpKernel, sigmaF = sigmaF, ell = ell)

plotGP(XStar, res, X, y, "Posterior Mean and 95% Probability Bands")
```



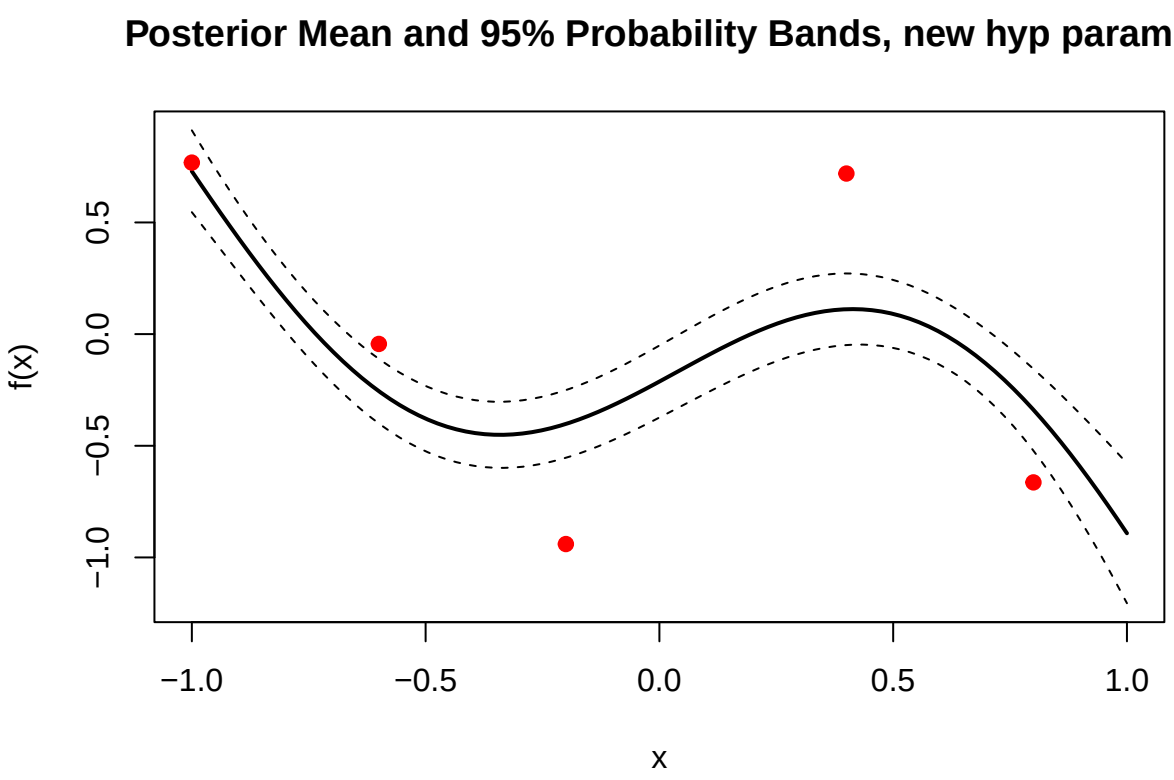
Task 4: Compute Posterior with All Observations Compute the posterior distribution of f using all five data points and plot the posterior mean with 95% bands.

```
# Step 4: Posterior with all five observations
X <- c(-1.0, -0.6, -0.2, 0.4, 0.8)
y <- c(0.768, -0.044, -0.940, 0.719, -0.664)

res <- posteriorGP(X, y, XStar, sigmaNoise, k = SquaredExpKernel, sigmaF = sigmaF, ell = ell)
plotGP(XStar, res, X, y, "Posterior Mean and 95% Probability Bands, 5 obs")
```



Task 5: Compare Hyperparameter Settings Repeat Task 4 using different hyperparameters: sigmaf = 1 and l = 1, compare the results.

```
# Hyperparameters
sigmaF <- 1     # sigma_f
ell <- 1        # length-scale l

res <- posteriorGP(X, y, XStar, sigmaNoise, k = SquaredExpKernel, sigmaF = sigmaF, ell = ell)
plotGP(XStar, res, X, y, "Posterior Mean and 95% Probability Bands, new hyp params")
```



When increasing ell from 0.3 to 1 we get more smoothness which is expected.

2.2 GP Regression with kernlab

Task 1: Define Kernel and Compute Covariance Matrix Define own squared exponential kernel and use the `kernelMatrix` function to compute the covariance matrix for given vectors.

```
# Preparing the data
data = read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/TempTullinge.cs

time = seq(1,2190, 5)
day = seq(1,365, 5)

data_sampled = data[time,]
temps = data_sampled$temp
```

```
# if doing it without class (covariance)
# SEKernel <- function(x1,x2,params = c(0,0)){
#   n1 <- length(x1)
#   n2 <- length(x2)
#   K <- matrix(NA,n1,n2)
#   for (i in 1:n2){
#     K[,i] <- (params[1]^2)*exp(-0.5*( (x1-x2[i])/params[2])^2 )
#   }
#   return(K)
```

```r
# }
#
# SEKernel(c(2, 185, 365), c(2, 185, 365), c(20,90))
# The further apart two points x values, the less correlated their f values
# time series data is not idd, peaking into fututre, use rolling window

# Squared Exponential Kernel Function
SEKernel = function(ell, sigmaF) {
  calc_K = function (X, XStar) {
    K = matrix(NA, length(X), length(XStar))
    for (i in 1:length(X)) {
      K[, i] = sigmaF ^ 2 * exp(-0.5 * ((X - XStar[i]) / ell) ^ 2)
    }
    return(K)
  }
  class(calc_K) = 'kernel' # Return as class kernel
  return (calc_K)
}

kernel = SEKernel(1,1)
# eval in points 1,2
kernel(1,2)
```

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

```r
X = c(1,3,4)
XStar = c(2,3,4)
# gives matrix for X and XStar
kernelMatrix(kernel, X, XStar) #K(X,XStar)
```

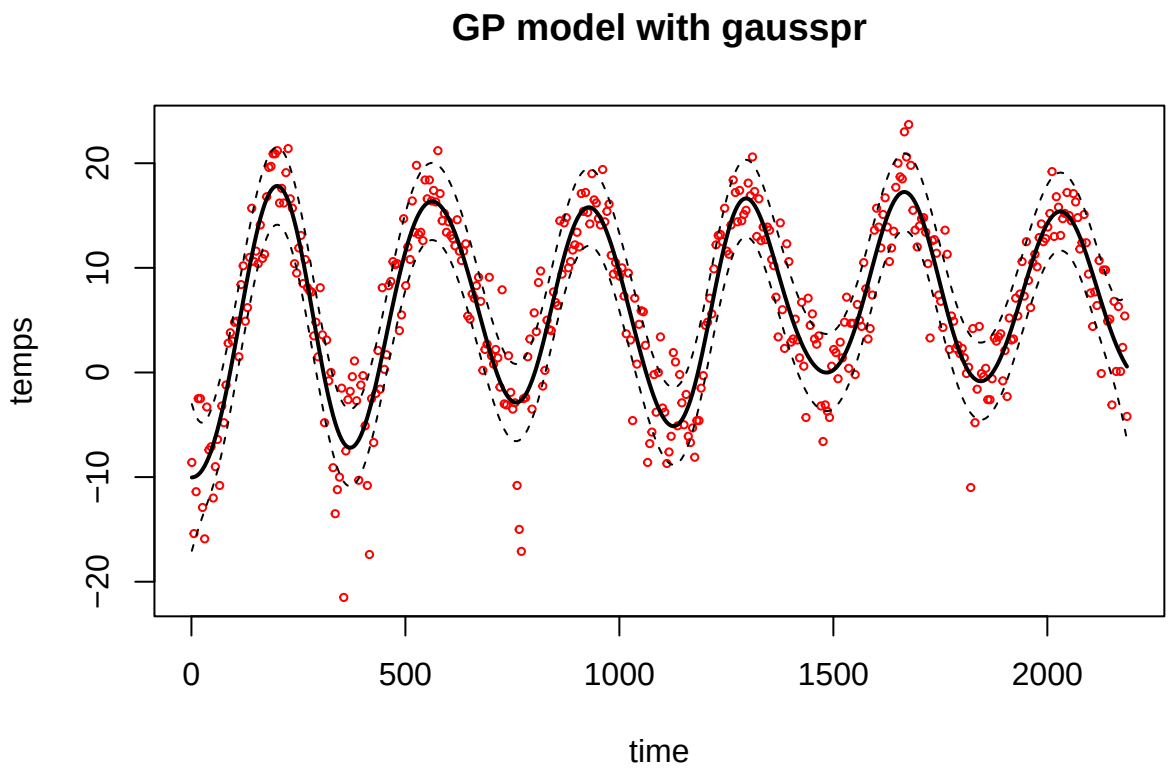
```
## An object of class "kernelMatrix"
##           [,1]      [,2]      [,3]
## [1,] 0.6065307 0.1353353 0.0111090
## [2,] 0.6065307 1.0000000 0.6065307
## [3,] 0.1353353 0.6065307 1.0000000
```

Task 2: Estimate GP Model with gausspr Use the gausspr function with sigmaf = 20 and l = 100, estimate the Gaussian process regression model, and plot the posterior mean.

```r
# new plot function
GP_plot_new = function(time_mean_pred, upper, lower, title){
plot(time, temps, pch = 1, cex = 0.5, col = "red", main = title)
lines(time, time_mean_pred, lwd = 2)
lines(time, upper,lty = 2)
lines(time, lower ,lty = 2)
}
```

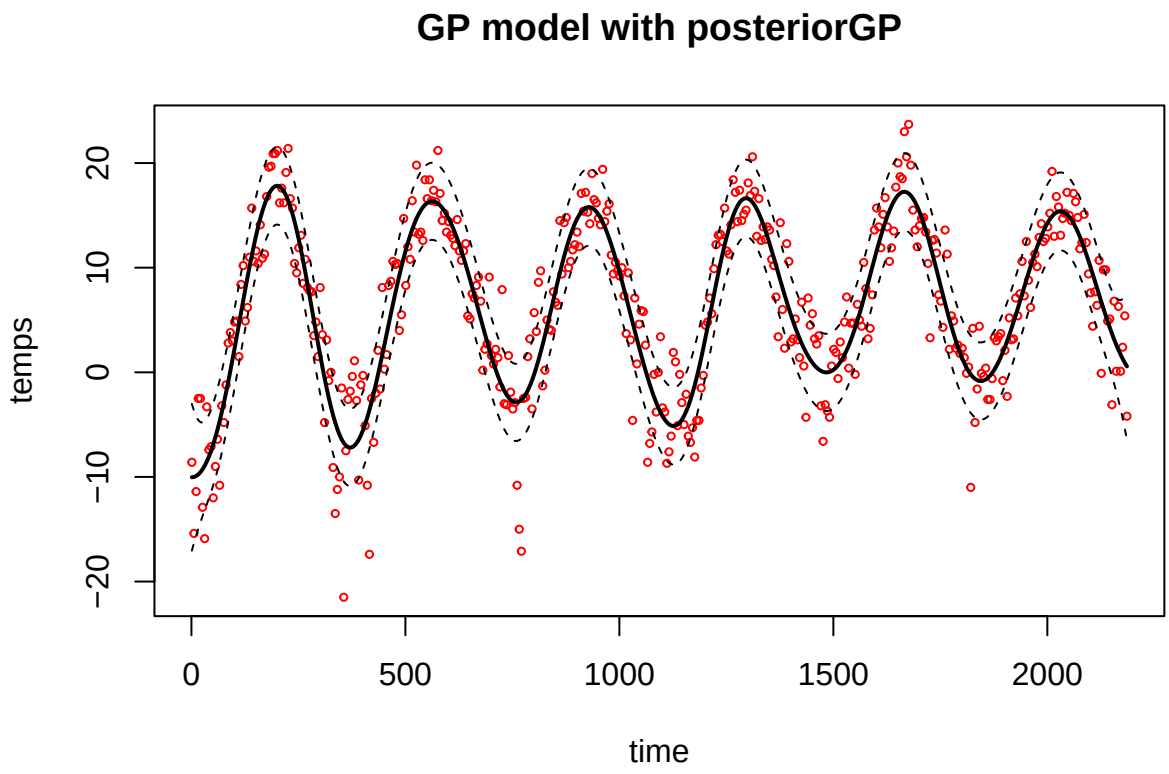
```r
quad_model = lm(temps~time + I(time^2), data = data_sampled)

fit = gausspr(time, temps, kernel = SEKernel(ell = 100, sigmaF = 20), var = var(quad_model$residuals),
```

```r
time_mean_pred <- predict(fit, time)

upper = time_mean_pred+1.96*predict(fit,time, type="sdeviation")
lower = time_mean_pred-1.96*predict(fit,time, type="sdeviation")
#Plot
GP_plot_new(time_mean_pred, upper, lower, "GP model with gausspr")
```



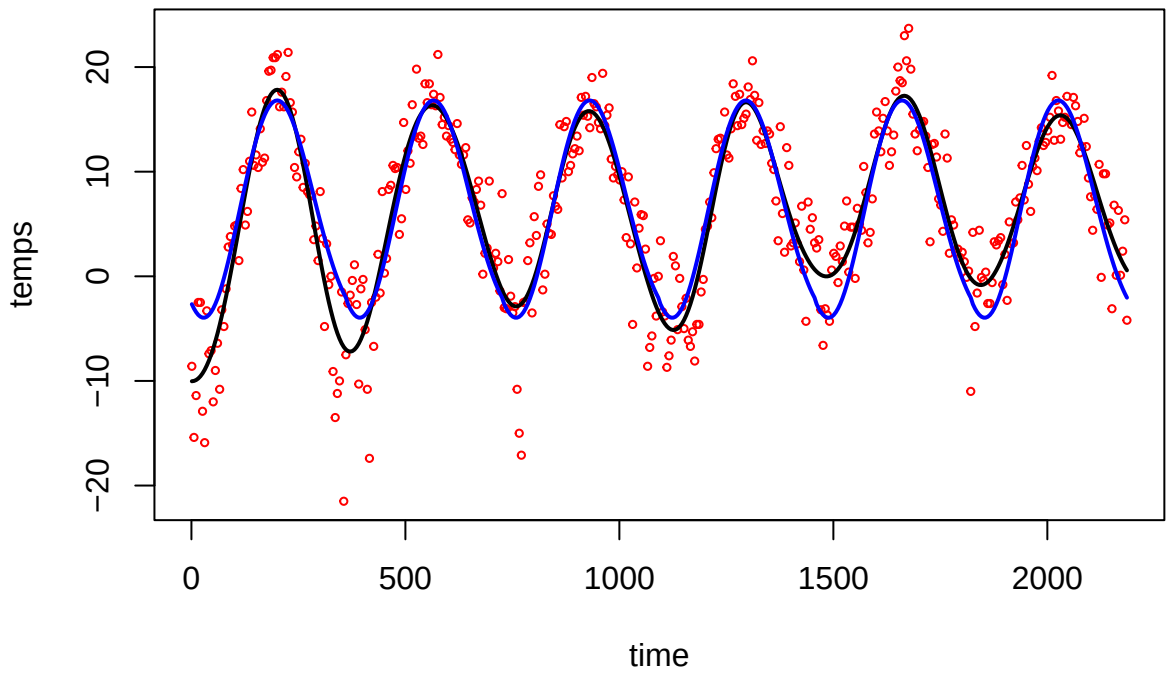
Task 3: Implement Algorithm 2.1 for GP Regression Use Algorithm 2.1 from Rasmussen and Williams' book (posteriorGP) to compute the posterior mean and variance.

```r
sigmaNoise = sqrt(var(quad_model$residuals))
res = posteriorGP(time, temps, time, sigmaNoise, k = SEKernel(ell = 100, sigmaF = 20))

  upper = res$mean + 1.96 * sqrt(diag(res$variance))
  lower = res$mean - 1.96 * sqrt(diag(res$variance))

#Plot
GP_plot_new(res$mean, upper, lower, "GP model with posteriorGP")
```



Task 4: Estimate Model with day Variable Estimate the model using `gausspr` with the day variable, superimpose the posterior mean on the previous model. Compare the results of both models. What are the pros and cons of each model?

```
# constructing day vector for 6 years
days = rep(day, 6)

quad_model_day = lm(temps~days + I(days^2), data = data_sampled)

fit_day = gausspr(days, temps, kernel = SEKernel(ell = 100, sigmaF = 20), var = var(quad_model_day$resi

day_mean_pred <- predict(fit_day, days)

upper2 = day_mean_pred+1.96*predict(fit_day,days, type="sdeviation")
lower2 = day_mean_pred-1.96*predict(fit_day,days, type="sdeviation")

plot(time, temps, pch = 1, cex = 0.5, col = "red")
lines(time, time_mean_pred, lwd = 2)
#lines(time, upper,lty = 2)
#lines(time, lower ,lty = 2)
lines(time, day_mean_pred, lwd = 2, col = "blue")
```



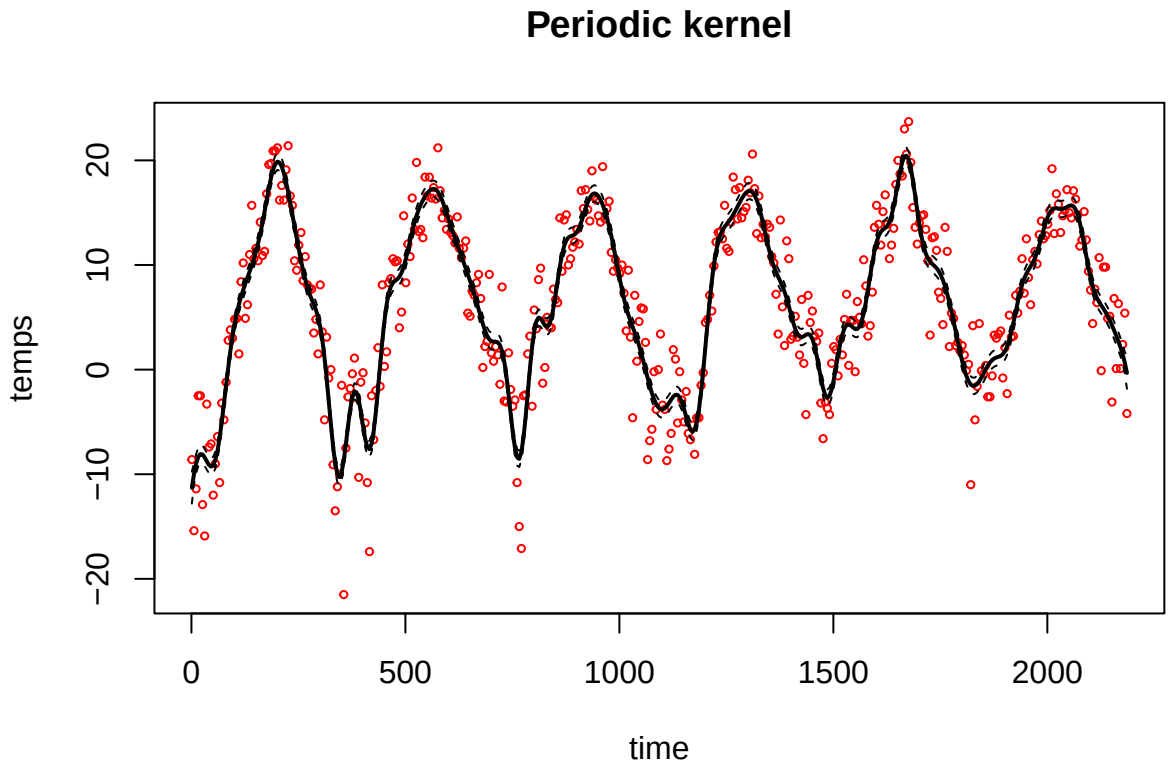

```
# bands for days
#lines(time, upper2,lty = 2, col = "green")
#lines(time, lower2,lty = 2, col = "green")
```

A: Quite similar performance. f(day) seems to have the same height for every year (which is logic since it repeat itself) while f(time) almost has a slight up-going trend, varying more from year to year.

Task 5: Implement Locally Periodic Kernel Implement the extended squared exponential kernel with a periodic kernel and compare the results.

```
# Periodic Kernel Function
PeriodicKernel <- function(sigmaF, ell1, ell2, d){

  calc_K = function (X, XStar) {
    temp1 = exp(-(2*sin(pi*abs(X - XStar)/d)^2)/ell1^2)
    temp2 = exp(- (0.5*abs(X - XStar)^2)/ell2^2)
    K = matrix(NA, length(X), length(XStar))
    for (i in 1:length(X)) {
      K[, i] = (sigmaF^2)*temp1*temp2
    }
    return(K)
  }
  class(calc_K) = 'kernel' # Return as class kernel
  return (calc_K)
}
```

```r
fit_periodic = gausspr(time, temps, kernel = PeriodicKernel(ell1 = 1, ell2 = 100, sigmaF = 20, d = 365)

time_mean_periodic_pred = predict(fit_periodic, time)

upper = time_mean_periodic_pred+1.96*predict(fit_periodic, time, type="sdeviation")
lower = time_mean_periodic_pred-1.96*predict(fit_periodic, time, type="sdeviation")

GP_plot_new(time_mean_periodic_pred, upper, lower, "Periodic kernel")
```



Q: Compare the fit to the previous two models (with Sigmaf = 20 and l = 100). Discuss the results.

A: The periodic GP fits the data more tightly with much narrower confidence bands compared to previous models. This could indicate over-fitting but at the same time we don't have any test data so we cant decide which generalizes better.

2.3 GP Classification with kernlab

Task 1: Fit GP Classification Model Use the `kernlab` package to fit a Gaussian process classification model for fraud, and plot contours of prediction probabilities.

```r
data <- read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/banknoteFraud
names(data) <- c("varWave","skewWave","kurtWave","entropyWave","fraud")
data[,5] <- as.factor(data[,5])

set.seed(111)
```

```r
SelectTraining <- sample(1:dim(data)[1], size = 1000, replace = FALSE)
train_data = data[SelectTraining, ]
test_data = data[-SelectTraining, ]

GPfit <- gausspr(fraud ~ varWave + skewWave, kernel = "rbfdot", data=train_data)
```

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

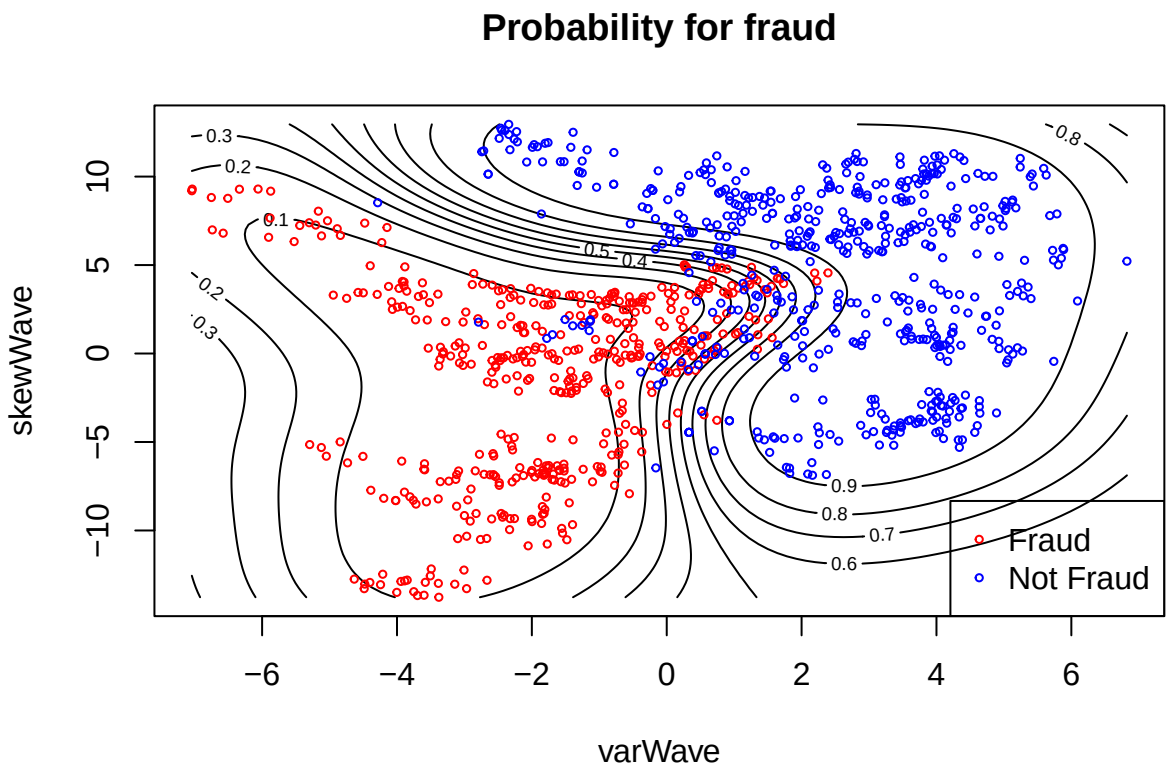
```r
pred = predict(GPfit, train_data)
cm = table(pred, train_data$fraud)
cm
```

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

```r
accuracy <- sum(diag(cm)) / sum(cm)
cat("Accuracy on the training set:", accuracy*100, "%\n")
```

```
## Accuracy on the training set: 94.1 %
```

```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(GPfit, gridPoints, type="probabilities")

# Plotting for Prob(fraud)
contour(x1,x2,matrix(probPreds[,1],100,byrow = TRUE), 10, xlab = "varWave", ylab = "skewWave", main = '
points(train_data[train_data[,5]==1,1],train_data[train_data[,5]==1,2],col="red", cex = 0.5)
points(train_data[train_data[,5]==0,1],train_data[train_data[,5]==0,2],col="blue", cex = 0.5)
legend("bottomright", legend = c("Fraud", "Not Fraud"), col = c("red", "blue"), pch = 1, pt.cex = 0.5)
```



Task 2: Make Predictions for Test Set Make predictions for the test set and compute the accuracy.

```
pred_test = predict(GPfit, test_data)
cm = table(pred_test, test_data$fraud)
cm
```

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

```
accuracy <- sum(diag(cm)) / sum(cm)
cat("Accuracy on the training set using two covariates:", round(accuracy*100, 1), "%\n")
```

```
## Accuracy on the training set using two covariates: 92.5 %
```

Task 3: Train Model with All Covariates Train a model using all four covariates and compare the accuracy to the model with only two covariates.

```
GPfit <- gausspr(fraud ~., kernel = "rbfdot", data=train_data)
```

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

```r
pred_test_all = predict(GPfit, test_data)
cm = table(pred_test_all, test_data$fraud)
cm
```

```
##
## pred_test_all   0   1
##             0 216   0
##             1   2 154
```

```r
accuracy <- sum(diag(cm)) / sum(cm)
cat("Accuracy on the training set using all covariates:", round(accuracy*100, 1), "%\n")
```

```
## Accuracy on the training set using all covariates: 99.5 %
```