Tenta Oct 2020

ID: 14671

10/27/2020

{ setup, include=FALSE} knitr::opts_chunk$set(echo = TRUE, message = FALSE, warning = FALSE, out.width='.49\\linewidth', fig.width=5, fig.height=5,fig.show='hold',fig.align='center', results='asis')

1 - Graphical models

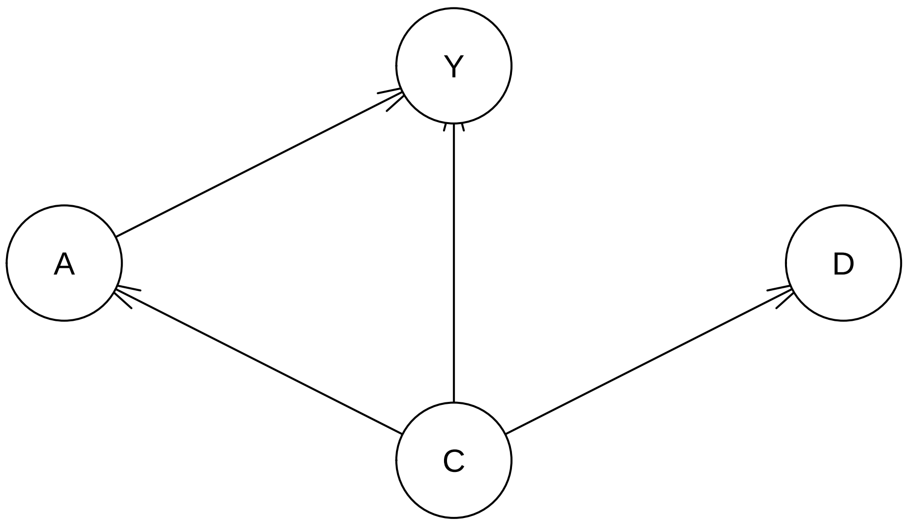
```
library(bnlearn)
library(gRain)
```

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

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

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

```
bn = model2network(string = "[C][A|C][D|C][Y|A:C]")
plot(bn)
```



```r
# 2
# Functions to check monotonicity
# Non decreasing
non_decreasing = function(grain, nodes, node_q ){
  evid = setEvidence(grain, nodes = nodes, states = c("1", "1"))
  prob_y = querygrain(evid, nodes = node_q)

  evid = setEvidence(grain, nodes = nodes, states = c("1", "0"))
  prob_y_0 = querygrain(evid, nodes = node_q)
  non_dec_c_1 = prob_y$Y[2]>=prob_y_0$Y[2]

  evid = setEvidence(grain, nodes = nodes, states = c("0", "1"))
  prob_y = querygrain(evid, nodes = node_q)

  evid = setEvidence(grain, nodes = nodes, states = c("0", "0"))
  prob_y_0 = querygrain(evid, nodes = node_q)
  non_dec_c_2 = prob_y$Y[2]>=prob_y_0$Y[2]
  non_dec_c = non_dec_c_1 & non_dec_c_2
  return(non_dec_c)
}

# Non increasing
non_increasing = function(grain, nodes, node_q ){
  evid = setEvidence(grain, nodes = nodes, states = c("1", "1"))
  prob_y = querygrain(evid, nodes = node_q)
  evid = setEvidence(grain, nodes = nodes, states = c("1", "0"))
  prob_y_0 = querygrain(evid, nodes = node_q)
  non_inc_1 = prob_y$Y[2]<=prob_y_0$Y[2]

  evid = setEvidence(grain, nodes = nodes, states = c("0", "1"))
  prob_y = querygrain(evid, nodes = node_q)

  evid = setEvidence(grain, nodes = nodes, states = c("0", "0"))
  prob_y_0 = querygrain(evid, nodes = node_q)
  non_inc_2 = prob_y$Y[2]<=prob_y_0$Y[2]
  non_inc = non_inc_1 & non_inc_2
  return(non_inc)
}

monotone = function(grain, nodes, node_q){
  non_dec = non_decreasing(grain, nodes, node_q)
  non_inc = non_increasing(grain, nodes, node_q)
  return(non_dec|non_inc)
}

count_i = 0
count_ii = 0
for (i in 1:1000) {

  # Random param for network
  bn = model2network(string = "[C][A|C][D|C][Y|A:C]")
  draw = runif(1)
  cptC = matrix(c(draw, 1-draw), ncol = 2, dimnames = list(NULL, c("0", "1")))

  draw = runif(2)
  cptA = c(draw[1], 1-draw[1], draw[2], 1-draw[2])
  dim(cptA) = c(2, 2)
  dimnames(cptA) = list("A" = c("0", "1"), "C" =  c("0", "1"))

  draw = runif(2)
  cptD = c(draw[1], 1-draw[1], draw[2], 1-draw[2])
  dim(cptD) = c(2, 2)
  dimnames(cptD) = list("D" = c("0", "1"), "C" =  c("0", "1"))

  draw = runif(4)
  cptY = c(draw[1], 1-draw[1], draw[2], 1-draw[2], draw[3], 1-draw[3], draw[4], 1-draw[4])
  dim(cptY) = c(2, 2, 2)
  dimnames(cptY) = list("Y" = c("0", "1"), "A" =  c("0", "1"), "C" =  c("0", "1"))

  bn_fit = custom.fit(bn, dist =list(C = cptC, A = cptA, D = cptD, Y = cptY) )
  grain = as.grain(bn_fit)

  # Monotone in C?
  nodes = c("A", "C")
  node_q = c("Y")
  mon_c = monotone(grain, nodes, node_q)
  mon_c
  # Monotone in D?
  nodes = c("A", "D")
  node_q = c("Y")
  mon_d = monotone(grain, nodes, node_q)
  mon_d
  # Check i
  if (mon_c == TRUE & mon_d == FALSE) {
    count_i = count_i+1
  }
  # check ii
  if (mon_d ==TRUE & mon_c == FALSE) {
    count_ii = count_ii+1
  }
}

cat("Count of i: " , count_i)
```

```
## Count of i:  0
```

```
cat("Count of ii: ", count_ii)
```

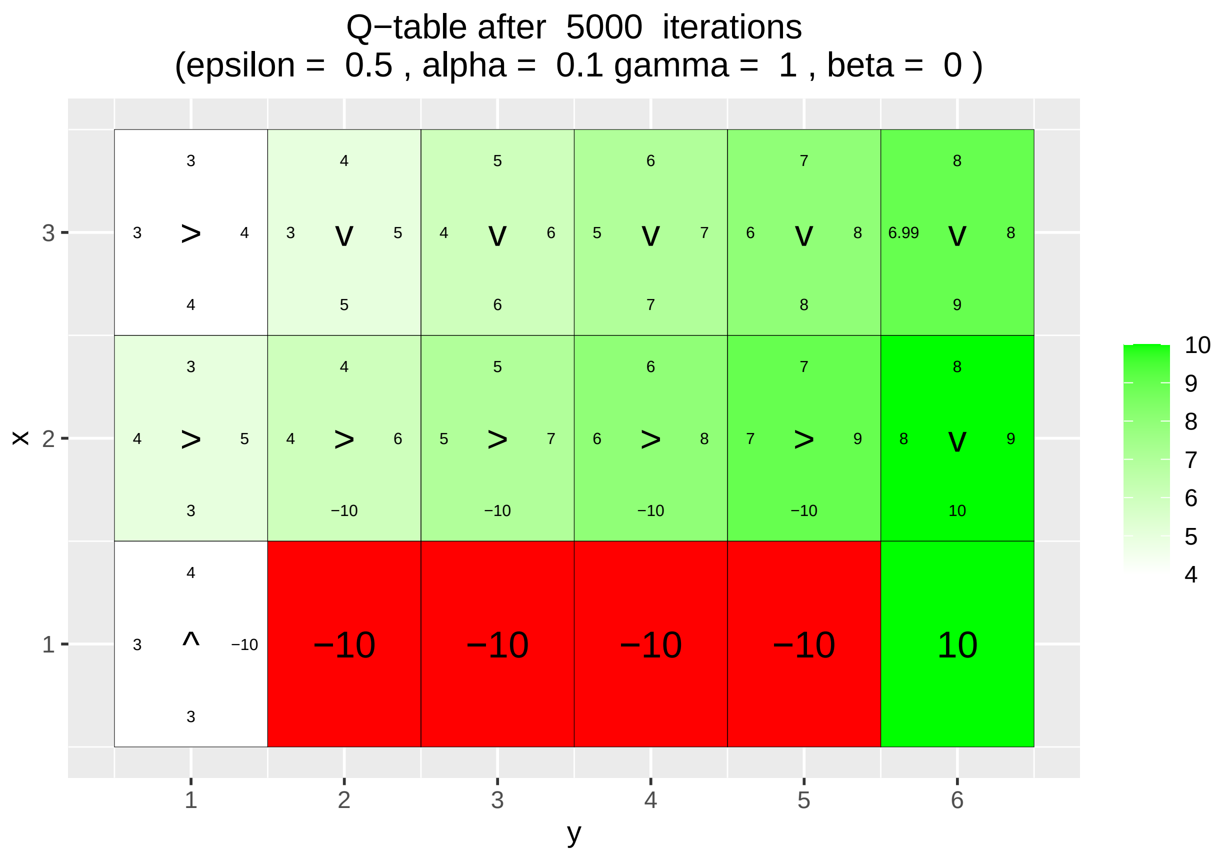
```
## Count of ii:  0
```

2 - Reinforcemnt learning

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

# install.packages("ggplot2")
# install.packages("vctrs")
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] == -1,q_table[x,y,1],NA),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == -1,q_table[x,y,2],NA),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == -1,q_table[x,y,3],NA),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == -1,q_table[x,y,4],NA),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y)
    ifelse(reward_map[x,y] == -1,arrows[GreedyPolicy(x,y)],reward_map[x,y]),df$x,df$y)
  df$val5 <- as.vector(foo)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == -1,max(q_table[x,y,]),
                                     ifelse(reward_map[x,y]<0,NA,reward_map[x,y])),df$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 = 2,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 2,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 2,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 2,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 5) +
          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){
  return(which.max(rank(q_table[x,y,], ties.method = "random")))
}

EpsilonGreedyPolicy <- function(x, y, epsilon){
  if(runif(1)>epsilon){
    direction = GreedyPolicy(x,y)
  }else{
    direction = sample(1:4,1)
  }
  return(direction)
}

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){
  current_state = start_state
  episode_reward = 0
  repeat{
    # Follow policy, execute action, get reward.
    action = EpsilonGreedyPolicy(current_state[1], current_state[2], epsilon)
    new_state = transition_model(current_state[1], current_state[2], action, beta)
    reward = reward_map[new_state[1], new_state[2]]
```

```r
    # Q-table update.
    q_action_value = q_table[current_state[1], current_state[2], action]
    next_exp_r = max(q_table[new_state[1], new_state[2],])

    correction = gamma*next_exp_r - q_action_value
    q_table[current_state[1], current_state[2],action] <<- q_action_value + alpha*(reward + correction)
    episode_reward = episode_reward+reward
    current_state=new_state
    if(reward!=-1)
      # End episode.
      return (episode_reward)

  }

}
new_q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                           beta = 0){
  current_state = start_state
  next_action = -1
  episode_reward = 0
  repeat{
    # Follow policy, execute action, get reward.
    if (next_action == -1) {
      action = EpsilonGreedyPolicy(current_state[1], current_state[2], epsilon)
    }else{
      action = next_action
    }
    new_state = transition_model(current_state[1], current_state[2], action, beta)
    reward = reward_map[new_state[1], new_state[2]]

    # Q-table update.
    next_action = EpsilonGreedyPolicy(new_state[1], new_state[2], epsilon)
    q_action_value = q_table[current_state[1], current_state[2], action]
    next_exp_r = q_table[new_state[1], new_state[2], next_action]

    correction = reward + gamma*next_exp_r - q_action_value
    q_table[current_state[1], current_state[2],action] <<- q_action_value + alpha*(correction)
    episode_reward = episode_reward+reward
    current_state=new_state
    if(reward!=-1)
      # End episode.
      return (episode_reward)
  }
}
MovingAverage <- function(x, n){

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

  return (rsum)
}
#################################################
# Environment C (the effect of beta).
```

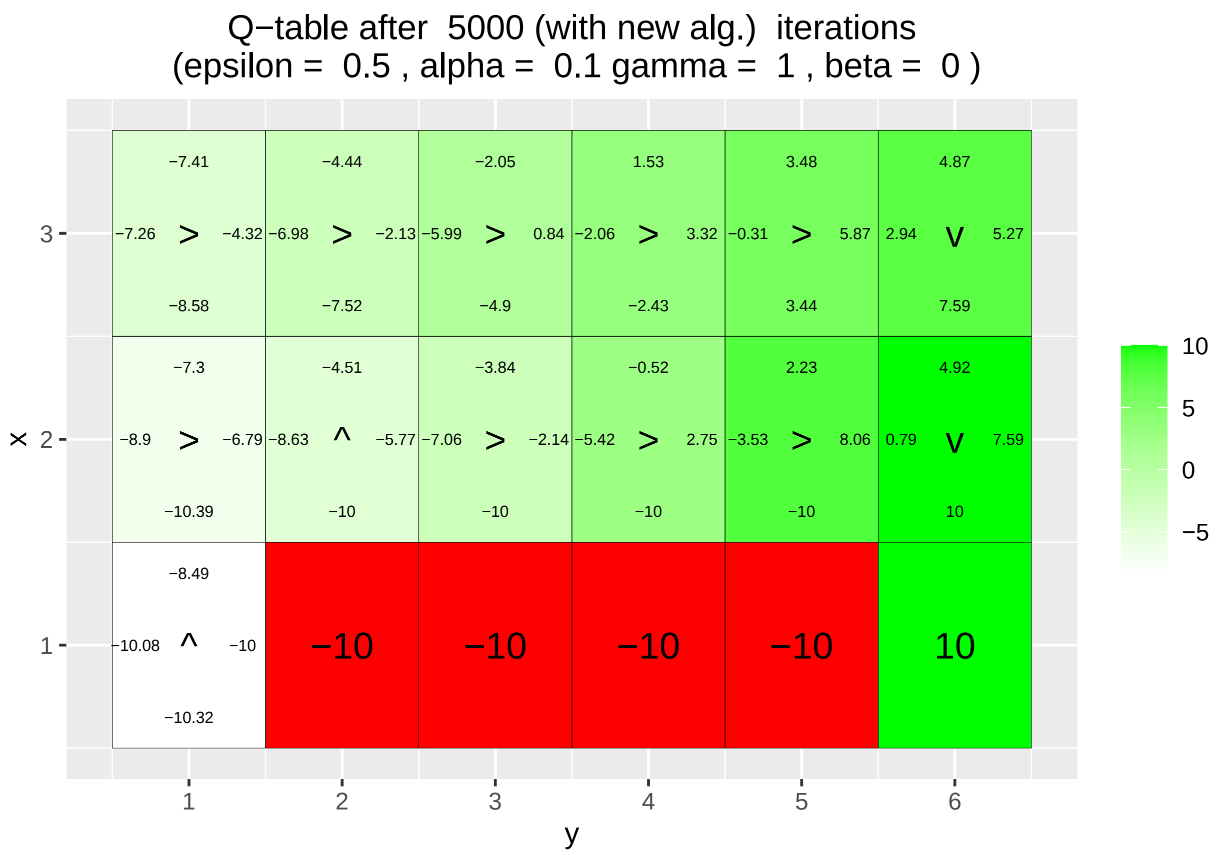
```
##################################################
H <- 3
W <- 6

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

# old q_learn
q_table <- array(0,dim = c(H,W,4))
episode_reward_old = rep(0, 5000)
for(i in 1:5000){
  foo <- q_learning(gamma = 1, beta = 0, start_state = c(1,1))
  episode_reward_old[i] = foo
}
vis_environment(5000, gamma = 1, beta = 0)
```



```
old_q_table = q_table
```

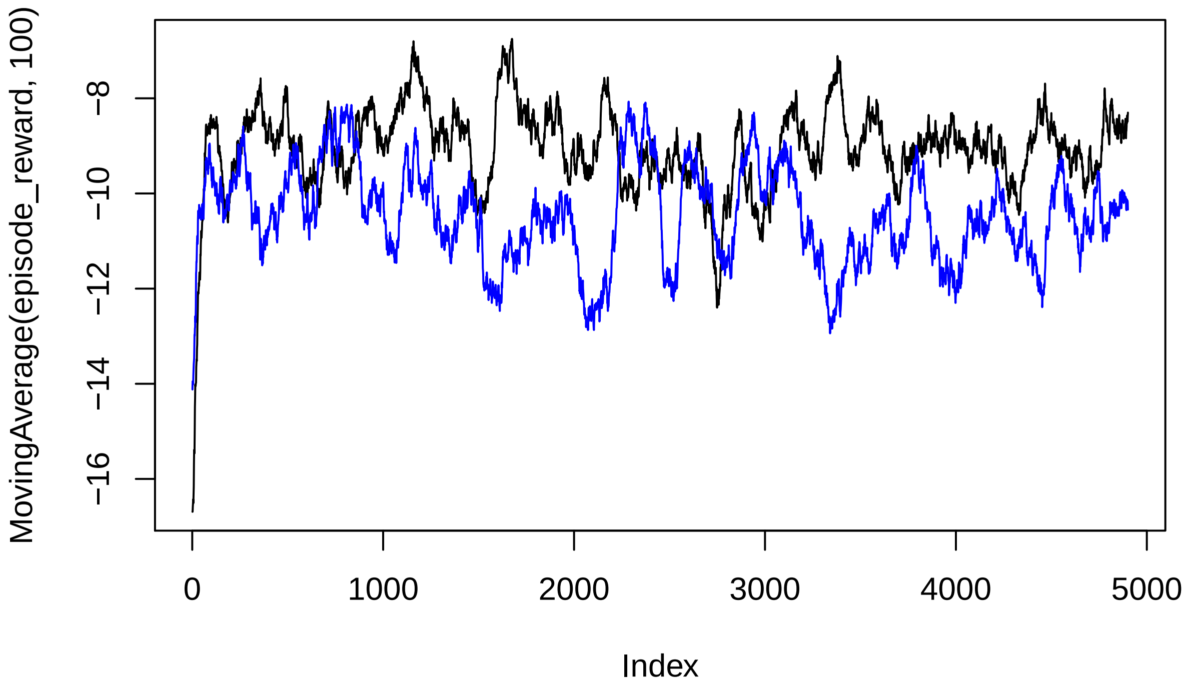
```
# New algoritm
q_table <- array(0,dim = c(H,W,4))
episode_reward = rep(0, 5000)
for(i in 1:5000){
  foo <- new_q_learning(gamma = 1, beta = 0, start_state = c(1,1))
  episode_reward[i] = foo
```

```r
}
vis_environment("5000 (with new alg.)", gamma = 1, beta = 0)
```



```r
new_q_table = q_table
```

```r
plot(MovingAverage(episode_reward,100),type = "l")
points(MovingAverage(episode_reward_old,100),type = "l", col = "Blue")
```

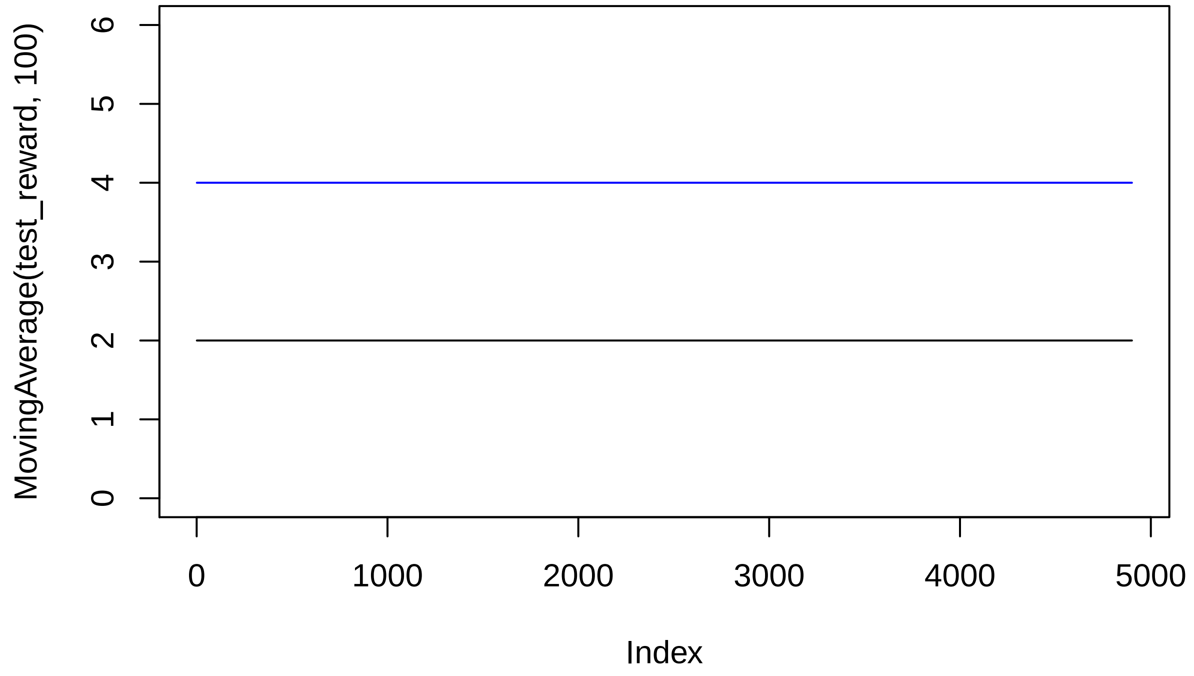


The new algorithm performes better, generating a higher reward over the later episodes. This is because it takes epsilon into accont, we can se that the new algorithm preffers to move away from the clif because if the agent stays along the cliff, there is always a risk of taking a random action(prob=epsilon) and then end up at the -10. This is not the case for the old q_learning.

```
# run 5000 test episodes
test_q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                            beta = 0){
  current_state = start_state
  episode_reward = 0
  repeat{
    # Follow policy, execute action, get reward.
    action = GreedyPolicy(current_state[1], current_state[2])
    new_state = transition_model(current_state[1], current_state[2], action, beta)
    reward = reward_map[new_state[1], new_state[2]]

    episode_reward = episode_reward+reward
    current_state=new_state
    if(reward!=-1)
      # End episode.
      return (episode_reward)
  }
}

# OLD Test
q_table = old_q_table
test_reward_old = rep(0, 5000)
```

```r
for(i in 1:5000){
  foo <- test_q_learning(gamma = 1, beta = 0, start_state = c(1,1))
  test_reward_old[i] = foo
}

# NEW Test
q_table = new_q_table
test_reward = rep(0, 5000)
for(i in 1:5000){
  foo <- test_q_learning(gamma = 1, beta = 0, start_state = c(1,1))
  test_reward[i] = foo
}
```

```r
plot(MovingAverage(test_reward,100),type = "l", ylim=c(0,6))
points(MovingAverage(test_reward_old,100),type = "l", col = "Blue")
```



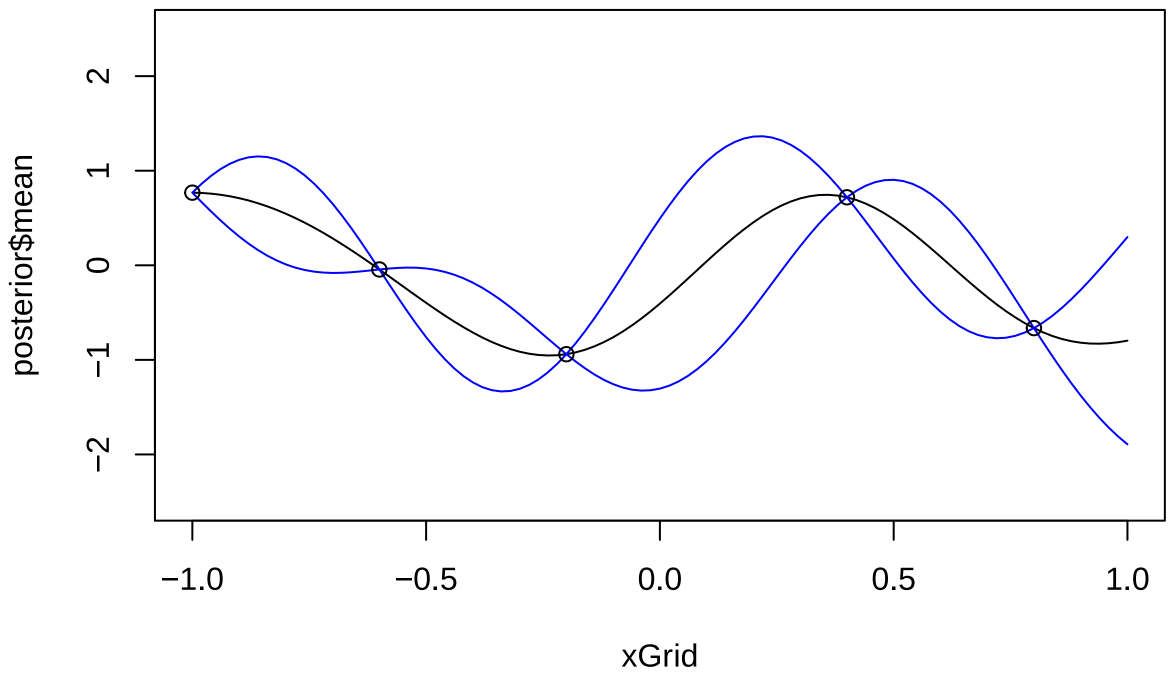
The old q_learning perfroms better in the test phase, that is becasue we do not use epsilon when “deploying” the learned policy. Instead of sometimes taking an action at random(with epsilon), we now allow the agent to always take an action based on the optimal policy.

3 - Gaussian Processes

```r
library(kernlab)
```

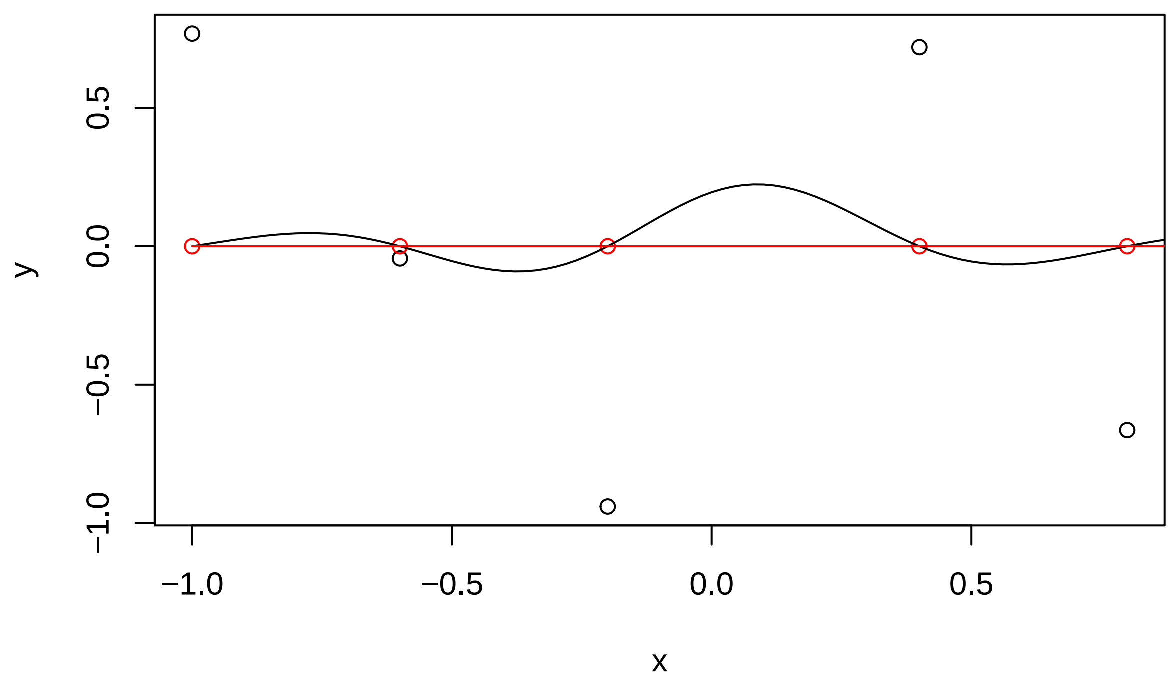
```
##
## Attaching package: 'kernlab'

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

```
# Computing the whole covariance matrix K from the kernel.
cov_function = function(X, Xstar){
  return(kernelMatrix(kernel = SEkernel, x=X, y=Xstar))
}

# Algorithm 2.1 sunns in n^3/6 instead of O(n^3). GP can be estimated faster, O(n) with eg. KISS-GP
posteriorGP = function(X_input, y_targets, k_cov_function, sigmaNoise=1, XStar){
  A = k_cov_function(X_input, X_input)
  A = A + diag(length(X_input))*sigmaNoise^2
  L = t(chol(A)) # chol Returns t(L)
  L_y = solve(L, y_targets)
  alpha = solve(t(L), L_y)

  k_star = k_cov_function(X_input, XStar)
  f_star = t(k_star)%*%alpha

  v = solve(L,k_star)
  V_f_star = k_cov_function(XStar, XStar) - t(v)%*%v
  return(list("mean"=f_star, "cov" = V_f_star))
}

sigma_f = 1
ell = 0.3
SEkernel <- rbfdot(sigma = 1/(2*ell^2)) # Reparametrize the rbfdot (which is the SE kernel) in kernlab.
sigma_n = 0

xGrid <- seq(-1,1,length=101) # x_star
x = c(-1.0, -0.6, -0.2, 0.4, 0.8)
y = c(0.768, -0.044, -0.940, 0.719, -0.664)
posterior = posteriorGP(x, y, cov_function, sigma_n, xGrid)

plot(xGrid, posterior$mean, ylim = c(-2.5,2.5), type="l") # posterior mean
std_dev = sqrt((diag(posterior$cov)))
points(xGrid, posterior$mean + 1.96*std_dev, col="blue", type="l")
points(xGrid, posterior$mean - 1.96*std_dev, col="blue", type="l")
points(x,y)
```



1. **Plot of posterior cov at x´**

```
# plot cov for x´=0
plot(x,y)
points(x,rep(0,5), col = "red")
lines(xGrid, posterior$cov[,50]) # Middle of xGrid is x´=0
lines(xGrid, rep(0, 101), col = "red")
```



2. Why is the posterior covariance zero at the training points?

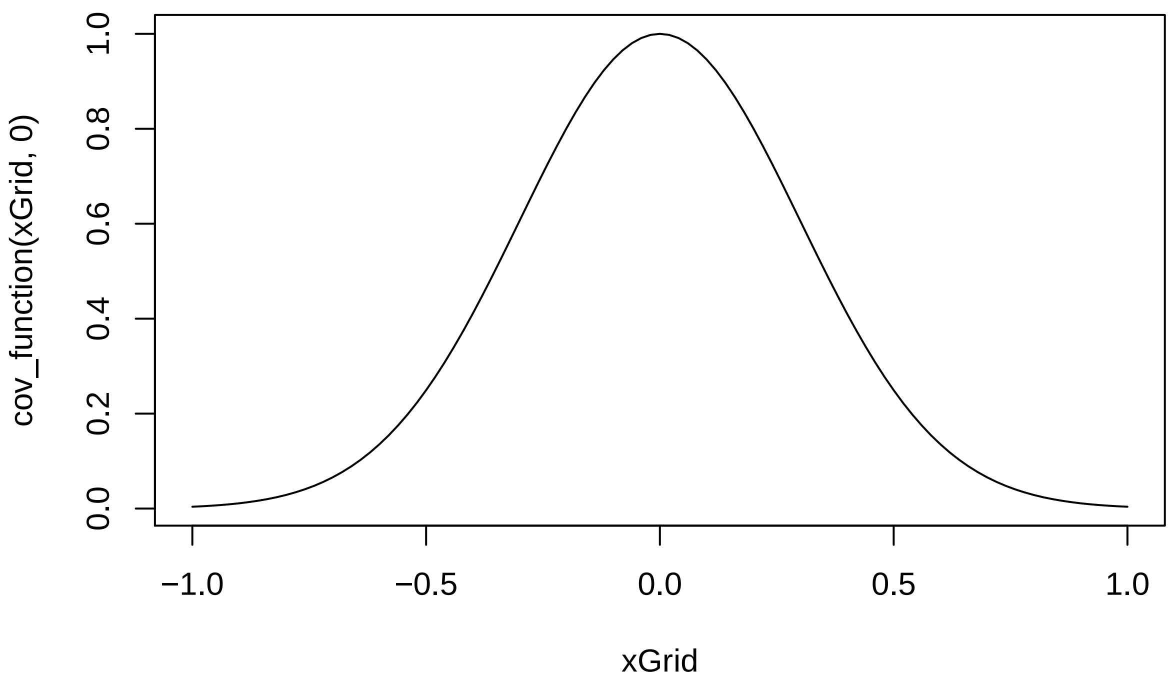
Covariance is 0 at the trainingpoints because if sigma_n is 0 then we have no noise in our data and we are certain that the observed y value is the actual f value for that x. From the model where "y ~ f + sigma_n"

3. Why does the posterior covariance not decrease as a function of the distance between x and xstar?

The posterior covariance describes the deviation from posterior mean. The posterior covariance matrix is calculated from the prior covariance function as: "k(x,0) - k(0,x)k(x,x)k(x,0)" therfore it also depend on the observed data points, x, and those points acts like a "reset" of the covariance function, as the covariance approaches 0 around the observed x-values.

4. Prior covariance

```
plot(xGrid, cov_function(xGrid, 0), type = "l")
```



## 3 part 2 - Gaussian Processes

```
# Algorithm 2.1 sunns in n^3/6 instead of O(n^3). GP can be estimated faster, O(n) with eg. KISS-GP
posteriorGP = function(X_input, y_targets, k_cov_function, sigmaNoise=1, XStar){
  A = k_cov_function(X_input, X_input)
  A = A + diag(length(X_input))*sigmaNoise^2
  L = t(chol(A)) # chol Returns t(L)
  L_y = solve(L, y_targets)
  alpha = solve(t(L), L_y)

  #k_star = k_cov_function(X_input, XStar)
  #f_star = t(k_star)%*%alpha

  #v = solve(L,k_star)
  #V_f_star = k_cov_function(XStar, XStar) - t(v)%*%v
  n = length(y_targets)
  log_mar = -0.5 * (t(y_targets)%*%alpha) - sum(diag(L))-(n/2)*log(2*pi)
  return(list("log" = log_mar) ) #list("mean"=f_star, "cov" = V_f_star, "log" = log_mar))
}


temprature = read.csv("https://github.com/STIMALiU/AdvMLCourse/raw/master/GaussianProcess/Code/TempTull
time = (1:2190)
nr_year = 2190/365
day = rep((1:365), nr_year)

reduce_data = function(array , nth){
```

```r
  array[seq(1, length(array), nth)]
}

temp = reduce_data(temprature$temp,5)
time = reduce_data(time, 5)
day = reduce_data(day, 5)

set_se_kernel = function(ell, sigma_f){
  se_kernel = function(x , y){
    r_square = sum((x-y)*(x-y)) # Euclidian distance^2
    return(sigma_f^2*exp(-r_square/(2*ell^2)))
  }
  class(se_kernel) <- "kernel"
  return(se_kernel)
}

LM = function(sigmaNoise, X, y, k,sigmaf_ell){
  #set kernel
  kernel = k(sigmaf_ell[2], sigmaf_ell[1])

  k_cov = function(x, x_star){
    return(kernelMatrix(kernel = kernel, x=x, y=x_star))
  }

  post = posteriorGP(X, y, k_cov, sigmaNoise, X)
  return(post$log)
}

sigmaf = 20
ell = 0.2

# using Optim
res = optim(par = 0.1, fn = LM, X=scale(time),y=scale(temp),k=set_se_kernel,sigmaf_ell=c(sigmaf, ell), 


cat("Resulting sigma_n: ", res$par)
```

```
## Resulting sigma_n:  0.5287827
```

```r
cat("With log marginal liklehood: ", res$value)
```

```
## With log marginal liklehood:  -853.5634
```

```r
# TEST if it seem reasanable
LM(0.1, X=scale(time),y=scale(temp),k=set_se_kernel,sigmaf_ell=c(sigmaf, ell))
```

```
##           [,1]
## [1,] -4289.269
```

```r
LM(0.5, X=scale(time),y=scale(temp),k=set_se_kernel,sigmaf_ell=c(sigmaf, ell))
```

```
##            [,1]
## [1,] -854.8534
```

```
# End TEST
```