University of Cape Town

Department of Statistical Sciences

STA5068Z

Machine Learning

Validation and Algorithms

Author:
Roger Bukuru

Student Number:
BKRROG001

November 10, 2024

Contents

1 Question 1

1.1 Experiment I

In this experiment, we test the performance of the drone navigation system with an initial random setup of 50 trees and a random starting position. The goal is to observe the drone's ability to find a valid escape path from the forest, avoiding trees.

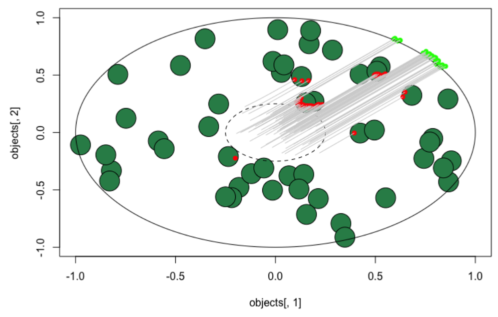

Figure 1: Escape Drone Initial Draw

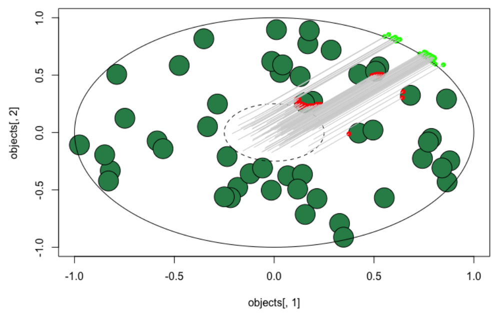

Figure 2: Escape Drone New Draw

In figures 1 and 2, we plotted the trace plots initial draw and a new draw of 50 trees. In terms of performance we computed the proportion of successful escapes in both cases and found that the initial setup had a better escape proportion of successful escapes of 50% compared to a 48% escape rate for the new draw.

With regards to the models ability to learn from the figures above, we observe that the model was able to find an escape route, however we note that it mostly relied on a single route besides there being multiple escape routes. This ability to escape while avoiding trees is promising but not perfect. The success rate shows the model has learned some level of avoidance, but there is room for further optimization as it appears that it has memorized a given route.

1.2 Experiment II

In this experiment, we updated the control function for the drone to improve its learning and adaptability.

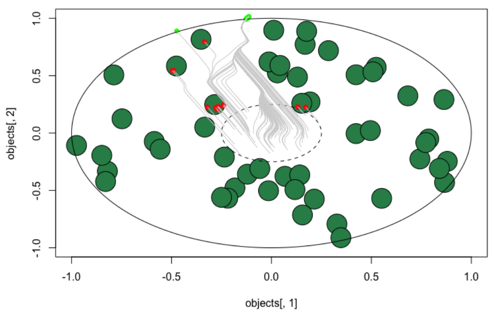

Figure 3: Escape Drone Initial Draw

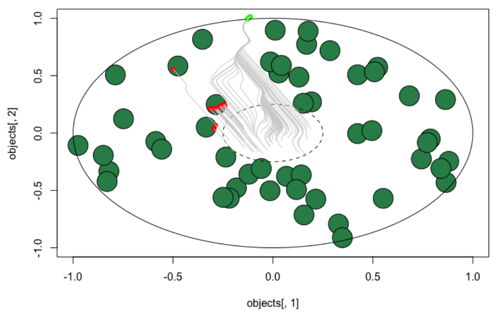

Figure 4: Escape Drone New Draw

In figures 3 and 4 we present the results when updating the control function. Reviewing the figures we observe that the model's ability to learn has improved, namely because we observe that the drones have explored multiple paths but moreover have found the most optimal path to escape the forest. As a result of this for the initial draw we observed a successful escape rate of 85% and on the second draw a successful escape rate of 77%. The success rates in this experiment demonstrates a marked improvement in generalization, indicating that the model had moved beyond memorization and could now better assess and respond to different tree layouts.

1.3 Experiment III

In this experiment, we further tested the drone navigation system by introducing random tree layouts and random starting positions for the drone.

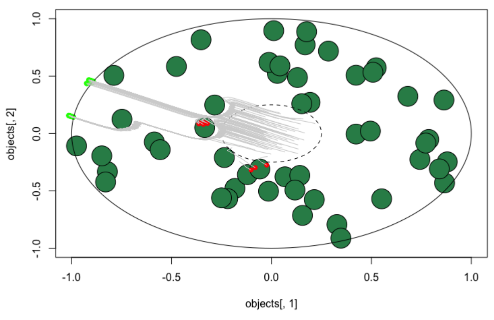

Figure 5: Escape Drone Initial Draw

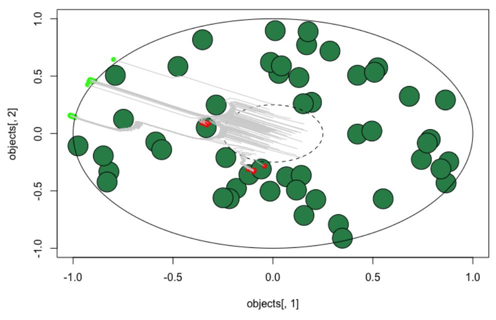

Figure 6: Escape Drone New Draw

For this setup, the model achieved a 90% escape success rate for the initial obstacle draw and an 88% success rate for the second draw. This indicates a strong improvement in it's ability to learn, with strong adaptability to varying configurations. In this experiment we explored random configurations as well as random starting positions for each game which provided a rigorous test of the model's ability to generalize its escape strategies. Given the high success rate across different configurations suggests that the model effectively leveraged the additional proximity input, allowing it to adapt dynamically and make more informed path decision, in the figures above we note something unique that we didn't observe before, which was that; the drones learnt how to go around trees and not simply straight underscoring the models improved ability to learn.

2 Question 2

2.1 Analysis of Tic-Tac-Toe with Modified Number of Moves

In this experiment, we analyzed the results of a modified Game Tree for Tic-Tac-Toe, where the game was played under two different conditions:

- The number of wins, draws, and losses in exactly 5 moves from an open board.
- The number of wins, draws, and losses in exactly 8 moves from an open board.

The results of the analysis are as follows:

- **Results for 5 Moves:**

  $$\text{Xwins} = 1440, \quad \text{Draws} = 0, \quad \text{Owins} = 0$$

  In this scenario, where the game was played to exactly 5 moves from an open board, all games resulted in a win for the starting player, **X**. There were no draws, and **O** (the second player) did not win any games.

- **Results for 8 Moves:**

  $$\text{Xwins} = 49392, \quad \text{Draws} = 23040, \quad \text{Owins} = 77904$$

  When the game was allowed to continue up to 8 moves, the results showed that **O** (the second player) won more frequently, with 77,904 wins compared to **X**'s 49,392 wins. Additionally, there were 23,040 draws.

From the results of the two scenarios, we can conclude the following:

- In the case of a game limited to exactly 5 moves, **X** (the first player) has a clear advantage and wins every time.
- However, when the game is extended to 8 moves, **O** (the second player) wins more often than **X**. Thus, if the game is played to allow at most 8 moves, it is better to be the second player.

This suggests that playing second gives a strategic advantage when the game allows for more moves, as **O** can better respond to **X**'s moves and potentially secure a win.

2.2 Monte Carlo Tree Search

In this section, we apply Monte Carlo Tree Search (MCTS) to estimate the relative frequencies of each outcome (X wins, O wins, and draws) for a given board state in a 3x3 Tic-Tac-Toe game. The board state used for the simulation is represented as:

$$m^T = [+1, +1, 0, 0, 0, 0, 0, 0, -1]$$

This board represents a game state where Player X (denoted by +1) has two moves, Player O (denoted by -1) has one move, and the remaining spots are empty.

The MCTS algorithm was implemented with each branch being pursued with a probability of $\alpha = 0.95$. A total of 1000 simulations were run to estimate the relative frequencies of the possible outcomes. The estimated frequencies for each outcome were:

$$\text{X wins} = 0.4106, \quad \text{Draws} = 0.0628, \quad \text{O wins} = 0.5266$$

Comparison with True Values

To compare these results with the true values, the true values for X wins, O wins, and draws were computed by enumerating the full game tree for the given board state. The true values are as follows:

$$\text{Total games} = 441, \quad \text{X wins} = 174, \quad \text{Draws} = 72, \quad \text{O wins} = 195$$

The relative frequencies of these outcomes based on the true values are:

$$\text{X wins} = \frac{174}{441} \approx 0.3945, \quad \text{Draws} = \frac{72}{441} \approx 0.1631, \quad \text{O wins} = \frac{195}{441} \approx 0.4424$$

Analysis of Results

The comparison between the MCTS estimates and the true values shows the following:

- X wins: The MCTS estimate of 0.4106 for X wins is fairly close to the true value of 0.3945. This indicates that the MCTS method is accurately estimating the likelihood of X winning the game in this scenario.
- Draws: The MCTS estimate of 0.0628 for draws is notably lower than the true value of 0.1631. This discrepancy suggests that MCTS is underestimating the likelihood of a draw, which is likely due to the randomness inherent in the simulation process.
- O wins: The MCTS estimate of 0.5266 for O wins is somewhat higher than the true value of 0.4424. This suggests that MCTS might be slightly overestimating O's chances of winning.

Overall, the MCTS method provides a good approximation of the relative frequencies of outcomes, but some deviations are present, especially in the estimation of draws and O wins. These differences can be attributed to the inherent randomness of the MCTS simulations and the fact that MCTS only explores a portion of the game tree rather than the entire space.

Analysis of MCTS Relative Frequencies for Different Alpha Values

We simulated 100 samples of the game tree using MCTS with two different exploration rates, $\alpha = 0.9$ and $\alpha = 0.7$, for the board state $m^T = [+1, +1, -1, 0, 0, 0, 0, 0, -1]$. The true relative frequencies for the outcomes are:

$$\text{Xwins} = 0.7260, \quad \text{Draws} = 0.1644, \quad \text{Owins} = 0.1096.$$

We depict the results of the relative frequencies of the outocmes with boxplots and superimpose the true values in figure 7 below.

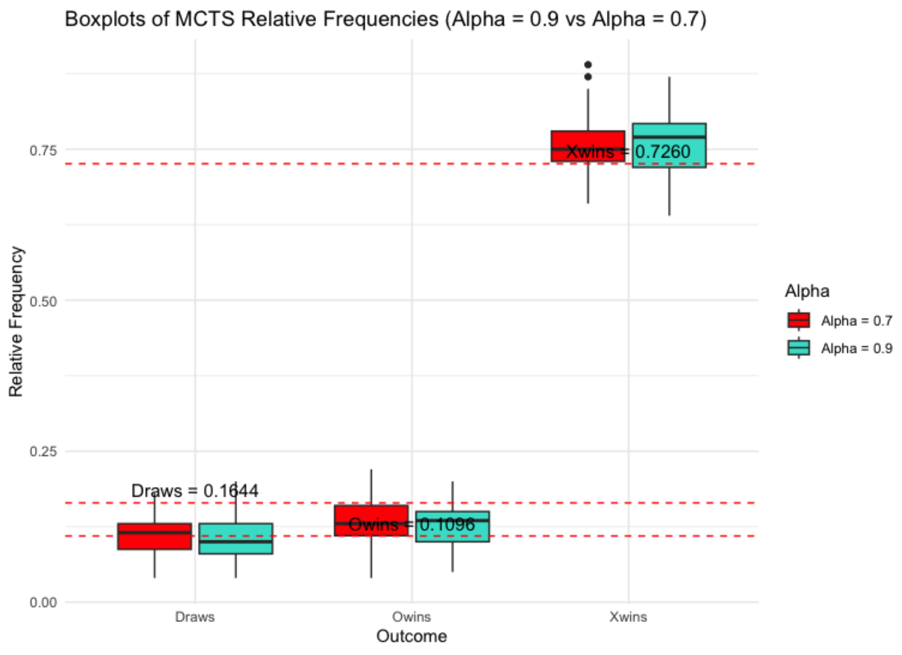

Figure 7: Relative Frequencies

We provide an analysis of our results below.

Alpha = 0.7 (Lower Exploration Rate)

When analyzing the results for $\alpha = 0.7$, the boxplots for **Xwins**, **Owins**, and **Draws** show relatively low variability, with **Xwins** closely aligned to the true value (0.7260) and **Owins** near the true value (0.1096). However, **Draws** are slightly underestimated but still within an acceptable range. The tight distribution indicates more stable and reliable estimates.

Alpha = 0.9 (Higher Exploration Rate)

For $\alpha = 0.9$, the boxplots exhibit higher variability. **Xwins** shows a wider spread around the true value (0.7260), indicating increased uncertainty in predictions. **Owins** is consistently overestimated, and the **Draws** frequency is poorly represented, with a significant underestimation around the average value. The larger spread suggests less reliable predictions compared to $\alpha = 0.7$.

Conclusion

- **Lower exploration** ($\alpha = 0.7$) on average leads to more accurate and consistent predictions, especially for **Xwins**.
- **Higher exploration** ($\alpha = 0.9$) on average results in greater variability and less reliable predictions, with **Owins** and **Draws** showing higher biases.

3 Question 3

3.1 Soft-Margin SVM

We implemented a custom **soft-margin SVM** function in R, named `my_svm()`, for binary classification. The function utilizes the `quadprog` package to solve the quadratic programming problem and includes a **cost parameter** to enforce the soft-margin constraint. For this implementation, we set the **cost parameter** to 1000 to balance between maximizing the margin and allowing for misclassifications.

Upon testing our model, we verified its correctness by comparing the number of **support vectors** with the results from the `svm()` function in the `e1071` package. Our implementation identified **28 support vectors**, which aligns with the results from the `e1071` package, confirming that the function works as expected.

The support vectors identified in the `PLA_dynamics.txt` dataset are visualized in the following plot, where the **red circles** highlight the support vectors used by the SVM algorithm for classification.

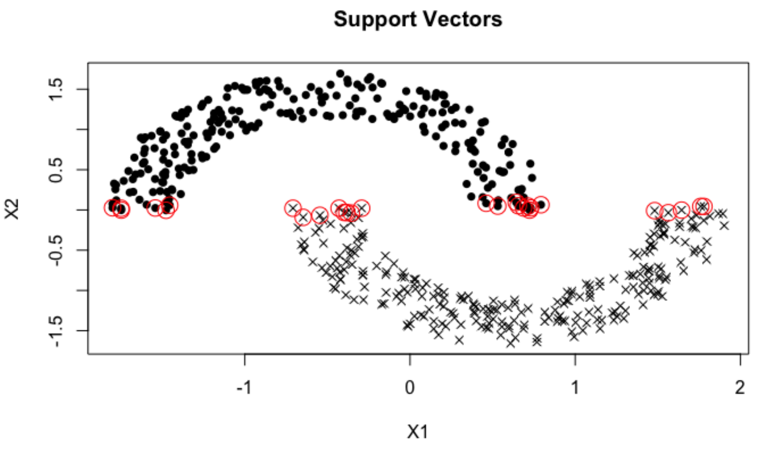

Figure 8: Support Vectors

This confirms that the **soft-margin SVM** classifier, with the appropriate cost parameter, effectively performs the classification task and correctly identifies the support vectors.

4 Appendix

4.1 Question 1

```r
rm(list = ls())
#=========================================================================
#                  1. Set up a game universe (Encoding)
#=========================================================================

draw_objects = function(J)
{
  r_o       =  runif(J,0.3,1)
  pi_vals   =  runif(J,-1,1)*pi
  o1        =  r_o*cos(pi_vals)     # x-coordinates of objects
  o2        =  r_o*sin(pi_vals)
  return(cbind(o1,o2))
}
draw_starts = function(N = 1)
{
  r_x      =  runif(N,0,0.25)
  pi_x     =  runif(N,-1,1)*pi
  xt       =  matrix(cbind(r_x*cos(pi_x),r_x*sin(pi_x)),nrow = N,byrow = TRUE)
}

set.seed(2024)
J       =  50
objects = draw_objects(J)
xt      = draw_starts()
delt    = 0.025             # How big are the steps you can take?
rad     = 0.05              # How close to an object before you crash?

# Just a function to plot the game:
plot_game = function(xt,objects,rad,cols = 1)
{
  plot(objects[,2]~objects[,1],type = 'n', ylim = c(-1,1), xlim = c(-1,1))
  symbols(objects[,1],objects[,2],circles = rep(rad,J),ylim = c(-1,1),
  xlim = c(-1,1),inches = FALSE,add = TRUE,bg = 'seagreen')
  points(xt[,2]~xt[,1], pch = 16, cex = 1,col = cols)
  pi_v = seq(-1,1,1/100)*pi
  y_edge = sin(pi_v)
  x_edge = cos(pi_v)
  lines(y_edge~x_edge)
  lines(c(0.25*y_edge)~c(0.25*x_edge),lty = 2)
}

plot_game(xt,objects,rad,'black')
#=========================================================================
#                   2. Evaluating the game state
```

```r
#==========================================================================

game_status=function(xt, objects, rad) {
  # Initialize the status and minimum distance vectors
  status    = rep(0, dim(xt)[1])
  min_dists = rep(0, dim(xt)[1])

  # Number of objects (trees)
  J = dim(objects)[1]

  # Create a matrix of ones for broadcasting in distance calculations
  ones = matrix(1, J, 1)

  # Calculate minimum distance to objects for each drone position
  for (i in 1:dim(xt)[1]) {
    min_dists[i] = min(sqrt(rowSums((objects - ones %*% xt[i,])^2)))
  }

  # Update status: 1 for escape, -1 for crash, 0 otherwise
  status = 1 * (sqrt(rowSums(xt^2)) > 1) - 1 * (min_dists < rad)

  # Compute 1 / (d_min - r) for each drone
  proximity_term = ifelse(min_dists > rad, 1 / (min_dists - rad), 100)


  # Return the status and minimum distances
  ret = list(status = status, dists = min_dists, proximity = proximity_term)
  return(ret)
}


# Just randomly move pieces for now:
#control = function(xt,theta)
#{
#  N_games = dim(xt)[1]
#  return(cbind(runif(N_games,-1,1),1))
#}

play = function(x0, delt, objects, rad, theta, plt = FALSE, trace = FALSE) {
  k = 0                       # Initialize step counter
  xt = x0                     # Set the initial positions of drones
  trajectories = NULL

  # Check the initial game status
  res_status = game_status(xt, objects, rad)
  status = res_status$status
  terminal = status != 0      # Indicates if a game has ended
  # Initialize trajectories storage if tracing is enabled
  if (trace) {
    trajectories = array(dim = c(dim(xt)[1], dim(xt)[2], 101))
    trajectories[,,1] = xt
  }

  # Plot the initial game setup if plotting is enabled
  if (plt) { plot_game(xt, objects, rad, 'black') }

  while ((any(status == 0)) & (k < 100)) {
    k = k + 1

    # Use the control function to get movement direction
    ct = control(xt, theta)
    xt = xt + ct * delt * cbind(1 - terminal, 1 - terminal)  # Update positions

    # Update the game status after movement
    res_status = game_status(xt, objects, rad)
    status = res_status$status
    terminal = status != 0  # Update terminal status

    # Record trajectory if tracing is enabled
    if (trace) { trajectories[,,k] = xt }
  }

  # Plot final state if plotting is enabled
  if (plt){ plot_game(xt, objects, rad, c('red', 'black', 'green')[status + 2])}

  return(list(k = k, status = status, xt = xt, trajectories = trajectories))
}



#==========================================================================
#                   3. Control (Giving a Model Agency.)
#==========================================================================

model = function(X, theta, nodes) {
  N = dim(X)[1]
  p = dim(X)[2]
  q = 2
  dims = c(p, nodes, q)

  # Populate weight and bias matrices
  index = 1:(dims[1] * dims[2])
  W1 = matrix(theta[index], dims[1], dims[2])
  index = max(index) + 1:(dims[2] * dims[3])
  W2 = matrix(theta[index], dims[2], dims[3])
  index = max(index) + 1:(dims[3] * dims[4])
  W3 = matrix(theta[index], dims[3], dims[4])

  index = max(index) + 1:(dims[2])
  b1 = matrix(theta[index], dims[2], 1)
  index = max(index) + 1:(dims[3])
  b2 = matrix(theta[index], dims[3], 1)
  index = max(index) + 1:(dims[4])
  b3 = matrix(theta[index], dims[4], 1)

  ones = matrix(1, 1, N)
  a0 = t(X)

  # Forward pass through network layers
  a1 = tanh(t(W1) %*% a0 + b1 %*% ones)
  a2 = tanh(t(W2) %*% a1 + b2 %*% ones)
  a3 = tanh(t(W3) %*% a2 + b3 %*% ones)

  # Return the output
  return(list(a3 = t(a3)))
}

p     = 3
q     = 2
nodes = 5
npars = p*nodes+nodes*nodes+nodes*q+nodes+nodes+q
npars
theta_rand = runif(npars,-1,1)

control = function(xt, theta) {

  res_status = game_status(xt, objects, rad)
  proximity_term = res_status$proximity
  X = cbind(xt, proximity_term)
  res_model = model(X, theta, rep(nodes,2))
  return(res_model$a3)  # The control output as movement directions

}

#==========================================================================
#                  4. Objectives and Fitness
#==========================================================================

play_a_game = function(theta) {
  objects <<- draw_objects(J)
  # Draw a random starting position
  xt = draw_starts(1)
  # Play the game
```

```r
  res = play(xt, delt, objects, rad, theta, plt = TRUE, trace = FALSE)
  # Return the final status
  score = res$status == 1  # Successful escape if status is 1
  return(score)
}

play_a_game(theta_rand)


#==========================================================================
#                    5. Evolutionary Learning
#==========================================================================

library('GA')

# Objective function for GA optimization
obj = play_a_game

# Run the GA
GA <- ga(type = 'real-valued',
         fitness = obj,
         lower = rep(-10, npars),
         upper = rep(+10, npars),
         popSize = 100,
         maxiter = 200,
         keepBest = TRUE)

# Plot the GA results
plot(GA)

# Get the optimized parameters
theta_hat = GA@solution[1,]

set.seed(2024)
# Draw 100 starting positions
xt_try = draw_starts(100)

# Run the game with optimized parameters
res_final = play(xt_try, delt, objects, rad, theta_hat, plt = TRUE, trace = TRUE)

# Calculate success rate
success_rate = mean(res_final$status == 1)

# Print the success rate
cat("Proportion of successful escapes:", success_rate, "\n")

# Plot trajectories
for (i in 1:dim(xt_try)[1]) {
  lines(res_final$trajectories[i,2,] ~ res_final$trajectories[i,1,],
  col = 'lightgrey')
}

####### New Draw Analysis #######
set.seed(2024)
J = 50
objects = draw_objects(J)

# Draw 100 random starting positions
xt_try = draw_starts(100)

theta_hat = GA@solution[1,]

plot_game(xt_try, objects, rad, 'black')

# Run the simulation for 100 games with trace enabled
res_final = play(xt_try, delt, objects, rad, theta_hat, plt = TRUE, trace = TRUE)

# Calculate and report the success rate (proportion of successful escapes)
success_rate = mean(res_final$status == 1)
cat("Proportion of successful escapes with new trees:", success_rate, "\n")

# Draw trace plots for each trajectory
for (i in 1:dim(xt_try)[1]) {
  lines(res_final$trajectories[i,2,] ~ res_final$trajectories[i,1,],
  col = 'lightgrey')
}
```

4.2 Question 2

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

#===============================================================================
# Populate the state matrix
#===============================================================================
S = matrix(0,9,8)
# Row Sums:
S[1:3,1] = 1
S[4:6,2] = 1
S[7:9,3] = 1
# Col Sums:
S[c(1:3)*3-2,4] = 1
S[c(1:3)*3-1,5] = 1
S[c(1:3)*3-0,6] = 1
```

```
# Diag Sums
S[c(1,5,9),7] = 1
S[c(3,5,7),8] = 1
S

#==============================================================================
# Evaluate the Game State rho(m,S)
#==============================================================================
rho = function(m,S)
{
  m = matrix(m,ncol = 1)
  player1  = any(t(m)%*%S==-3) # X
  player2  = any(t(m)%*%S==+3) # O
  m_1      = (m==-1) #m_
  m_2      = (m==+1) #m+
  tie      = sum((t(m_1)%*%S>0)*(t(m_2)%*%S>0))==8
  winner   = c(-1,0,1)[c(player1,tie,player2)]
  terminal = player1|tie|player2
  ret      = list(terminal = terminal,winner = winner)
  return(ret)
}

m = as.matrix(c(1,1,-1,0,0,0,0,0,-1))
matrix(m,3,3,byrow = TRUE)
rho(m,S)

m = as.matrix(c(1,1,-1,0,0,-1,0,0,-1))
matrix(m,3,3,byrow = TRUE)
rho(m,S)

#==============================================================================
# game_tree(m,k) returns a vector of -1, 0, and +1s delineating all terminal
# game states.
#==============================================================================
game_tree = function(m, k, n_moves = 0, max_moves = 8)
{
  g = c()
  game_state = rho(m, S)

  if (game_state$terminal || n_moves >= max_moves) {
    g = c(g, game_state$winner)
    return(g)
  } else {
    Index = which(m == 0)  # Find empty cells to make a move
    for (i in 1:length(Index)) {
      x = m
      x[Index[i]] = k
      g = c(g, game_tree(x, -k, n_moves + 1, max_moves))
    }
    return(g)
  }
}

m = as.matrix(c(0,0,0,0,0,0,0,0,0))  # Open Board
#m = as.matrix(c(1, 1, 0, 0, 0, 0, 0, 0, -1)) #2b
#m = as.matrix(c(1, 1, -1, 0, 0, 0, 0, 0, -1)) #2c
max_moves = 8
result    = game_tree(m, -1, 0, max_moves)
n_g       = length(result)
# Count results for wins and draws
Xwins = sum(result == -1)  # Count X wins
Draws = sum(result == 0)   # Count draws
Owins = sum(result == 1)   # Count O wins
c(n_g,Xwins, Draws, Owins)

#===============================================================================
# Write a function that draws the game tree.
#===============================================================================
draw_game_tree = function(m,k,x0 = 0,d = 0, lims = c(-1,1))
{
  g = c()
  game_state = rho(m,S)
  if(d == 0) # Initialise plot region
  {
    plot(1,1,type = 'n',xlim = lims, ylim = c(0,sum(m==0)+2),axes = FALSE)
    axis(2)
  }
  if(game_state$terminal)
  {
    text(x0,d,labels = c('X','-','O')[game_state$winner+2],cex = 0.5) # Placing terminal nodes
    g = c(g,game_state$winner)
    return(g)
  }else{
    Index = which(m == 0)

    x_seq = seq(lims[1],lims[2],length = length(Index)+1)
    x_mids = x_seq[-1]-0.5*diff(x_seq)
    for(i in 1:length(Index))
    {
      segments(x0,d,x_mids,d+1,col = 'grey')
      x = m
      x[Index[i]]=k
      g = c(g,draw_game_tree(x,-1*k,x_mids[i],d+1,lims = c(x_seq[i],x_seq[i+1])))
    }
    return(g)
  }
```

```r
}

#m = as.matrix(c(1,1,0,0,0,0,0,0,-1))
m = as.matrix(c(0,0,0,0,0,0,0,0,0))
matrix(m,3,3,byrow = TRUE)
res = draw_game_tree(m,-1)
n_g  =         length(res)
Xwins = sum(res==-1)
Draws = sum(res== 0)
Owins = sum(res==+1)
c(n_g,Xwins,Draws,Owins)



#================================================================================
# MCTS Function
#================================================================================

mcts = function(m, k, alpha = 0.95, n_simulations = 100) {
  outcomes = matrix(0, nrow = n_simulations, ncol = 3)
  colnames(outcomes) = c("Xwins", "Draws", "Owins")

  for (sim in 1:n_simulations) {
    current_board = m
    current_player = k

    # Run a simulation of the game with random moves based on the given probability alpha
    while (TRUE) {
      game_state = rho(current_board, S)

      # Check if the game has reached a terminal state
      if (game_state$terminal) {
        if (game_state$winner == -1) {  # X wins
          outcomes[sim, "Xwins"] = 1
        } else if (game_state$winner == 1) {  # O wins
          outcomes[sim, "Owins"] = 1
        } else {  # Draw
          outcomes[sim, "Draws"] = 1
        }
        break
      }

      # Get empty spots
      empty_spots = which(current_board == 0)

      # If there are empty spots, choose the next move with probability alpha
      if (length(empty_spots) > 0) {
        if (runif(1) < alpha) {
          # Explore: Choose a random move
          move = sample(empty_spots, 1)
        } else {
          # Exploit: Choose a deterministic move (for simplicity, we still randomly choose here)
          move = sample(empty_spots, 1)
        }

        # Make the move
        current_board[move] = current_player
        current_player = -current_player  # Switch player
      }
    }
  }

  # Return the relative frequencies of each outcome
  outcome_frequencies = colMeans(outcomes)
  return(outcome_frequencies)
}

#===============================================================================
# Simulate the MCTS with the given board state
#===============================================================================
m = as.matrix(c(1, 1, 0, 0, 0, 0, 0, 0, -1))  # 2b
alpha = 0.95
n_simulations = 1000

# Run MCTS simulation
relative_frequencies = mcts(m, -1, alpha, n_simulations)

relative_frequencies


library(ggplot2)

# Define the board state and set parameters
m = as.matrix(c(1, 1, -1, 0, 0, 0, 0, 0, -1))  # 2c
n_simulations = 100
n_games = 100

# Run simulations for alpha = 0.9
relative_frequencies_alpha_09 = replicate(n_games,
mcts(m, -1, 0.9, n_simulations = n_simulations))

# Run simulations for alpha = 0.7
relative_frequencies_alpha_07 = replicate(n_games,
mcts(m, -1, 0.7, n_simulations = n_simulations))

alpha_09_df = data.frame(t(relative_frequencies_alpha_09))
```

```
alpha_07_df = data.frame(t(relative_frequencies_alpha_07))

# True values for comparison
true_values = c(Xwins = 53/73, Draws = 12/73, Owins = 8/73)

df = data.frame(
  Xwins = c(alpha_09_df$Xwins, alpha_07_df$Xwins),
  Draws = c(alpha_09_df$Draws, alpha_07_df$Draws),
  Owins = c(alpha_09_df$Owins, alpha_07_df$Owins),
  alpha = factor(rep(c("Alpha = 0.9", "Alpha = 0.7"), each = n_simulations))
)

# Reshape the data
df_long = reshape(df,
                  varying = c("Xwins", "Draws", "Owins"),
                  v.names = "value",
                  timevar = "outcome",
                  times = c("Xwins", "Draws", "Owins"),
                  direction = "long")

# Labels for true values
true_labels = data.frame(
  outcome = c("Xwins", "Draws", "Owins"),
  true_value = c(true_values[1], true_values[2], true_values[3]),
  label = c("Xwins = 0.7260", "Draws = 0.1644", "Owins = 0.1096")
)

# Draw the boxplots
ggplot(df_long, aes(x = outcome, y = value, fill = as.factor(alpha))) +
  geom_boxplot() +
  scale_fill_manual(values = c("Alpha = 0.7" = "red",
  "Alpha = 0.9" = "turquoise")) +
  geom_hline(data = true_labels, aes(yintercept = true_value),
  color = "red", linetype = "dashed") +
  geom_text(data = true_labels, aes(x = outcome, y = true_value, label = label),
            color = "black", vjust = -0.5, size = 4) +
  labs(title = "Boxplots of MCTS Relative Frequencies
  (Alpha = 0.9 vs Alpha = 0.7)",
       x = "Outcome",
       y = "Relative Frequency",
       fill = "Alpha") +
  theme_minimal()
```

4.3 Question 3

```r
rm(list = ls())
library(quadprog)
library(tidyverse)
library(e1071)

my_svm <- function(Y, X, gm = 1, cf = 1, dg = 2, cost = 10000) {

  # Define the polynomial kernel function
  KK <- function(X1, X2) {
    return((cf + gm * t(X1) %*% X2)^dg)
  }


  N <- length(Y)

  # Compute the Gram matrix (kernel matrix)
  DD <- matrix(0, N, N)
  for (i in 1:N) {
    for (j in 1:N) {
      DD[i, j] <- Y[i] * Y[j] * KK(X[i, ], X[j, ])
    }
  }

  # Add a small epsilon to the diagonal for numerical stability
  eps <- 0.041
  DD <- DD + eps * diag(N)

  # Modify constraints for soft-margin
  Amat <- cbind(Y, diag(N), diag(N) * -1)# Additional constraints to limit a <= cost
  bvec <- c(0, rep(0, N), rep(-cost, N))    # Add upper bound of cost

  d <- matrix(rep(1, N), N, 1)

  res <- solve.QP(Dmat = DD, dvec = d, Amat = Amat, bvec = bvec,
  meq = 1, factorized = FALSE)

  # Lagrange multipliers (a.k.a. alpha values)
  a <- pmin(res$solution, cost)  # Cap alphas at cost

  # Retrieve the support vectors based on non-zero alpha values
  support_vector_indices <- which(a > 1e-3)
  support_vectors <- X[support_vector_indices, ]
  support_labels <- Y[support_vector_indices]
  support_alphas <- a[support_vector_indices]

  # Calculate the intercept
  intercept <- mean(support_labels - rowSums(sapply(1:length(support_alphas)
  , function(i) {
    support_alphas[i] * support_labels[i] * KK(support_vectors[i, ], t(X))
  })))

  return(list(
    alphas = a,
    intercept = intercept,
    support_vectors = support_vectors,
    support_labels = support_labels
  ))
}


dat <- read.table("PLA Dynamics.txt")

# Standardize the features
X <- scale(cbind(dat$X1, dat$X2))
Y <- dat$Y

# Fit the model
model <- my_svm(Y = Y, X = X, gm = 1, cf = 1, dg = 2, cost = 10000)


plot(X[, 2] ~ X[, 1], pch = c(4, 16)[(Y + 1) / 2 + 1],
xlab = "X1", ylab = "X2", main="Support Vectors")
points(model$support_vectors, pch = 1, col = "red", cex = 2)

# Compare with the e1071 SVM
svm_model <- svm(Y ~ ., data = data.frame(Y = as.factor(Y), X1 = X[, 1],
X2 = X[, 2]),
                 kernel = "polynomial", degree = 2, gamma = 1,
                 coef0 = 1, cost = 10000, scale = FALSE)

# SVMs for each model
cat("Number of support vectors (my_svm):",
length(model$support_vectors) / ncol(X), "\n")
cat("Number of support vectors (e1071 svm):", length(svm_model$SV), "\n")
```