Department of Mechatronics and Control Engineering
University of Engineering and Technology, Lahore Pakistan

LAB 2: Probability Simulation In R Studio

Aims and Objectives:

- Understand core probability concepts, rules, and basic simulation techniques in R.
- Strengthen R programming skills by implementing efficient probability simulations.
- Validate theoretical concepts through simulations, comparing results with analytical solutions.

Problem Set:

We will be focusing on 3 main problems in today's lab session, and how run these simulations on R studio.

Problem 1:

You can easily look up the probability of 5 card poker hands, e.g. https://en.wikipedia.org/wiki/Poker_probability
For this problem, let's consider hands with 6 cards.
Here are two types:
1. Two-pair: Two cards have one rank, two cards have another rank, and the remaining two cards have two different ranks. e.g. {2♡, 2♠, 5♡, 5♣, Q♢, K♢}
2. Three-of-a-kind: Three cards have one rank and the remaining three cards have three other ranks. e.g. {2♡, 2♠, 2♣, 5♣, 9♠, K♡} Calculate the probability of each type of hand. Which is more probable?

Question 1: Solution

```
# Simulate some 6 card poker hands

  cat('Problem 1\n')
  ntrials = 100
  size_hand = 6
  nranks = 13
  nsuits = 4
  deck = c(1:(nranks*nsuits))


  is_two_pair = function(hand) {
    hand = hand %% nranks # remove suit information
    ranks_in_hand = hand[!duplicated(hand)]
    ret = FALSE
    npair = 0
    for (r in ranks_in_hand) {
      u = sum(hand==r)
      if (u == 2) {
        npair = npair + 1
      }
      else if (u > 2) { # 3 or more of a kind
        npair = 0
        break
      }
    }
    if (npair == 2) {
      ret = TRUE
    }
    return(ret)
  }
```

```
  is_three_of_a_kind = function(hand) {
    hand = hand %% nranks # remove suit information
    ranks_in_hand = hand[!duplicated(hand)]
    ret = FALSE
    ntriples = 0
    for (r in ranks_in_hand) {
      u = sum(hand==r)
      if (u == 3) {
        ntriples = ntriples + 1
      }
      else if (u == 2 || u > 3) { # Found pair or more than 3 of a kind
        ntriples = 0
        break
      }
    }
    if (ntriples == 1) {
      ret = TRUE
    }
    return(ret)
  }


  # Simulate 2 pair
  cnt_2_pair = 0
  cnt_3_of_kind = 0
  for (j in 1:ntrials) {
    h = sample(deck, size_hand)
    if (is_two_pair(h)) {
      cnt_2_pair = cnt_2_pair + 1
    }
    if (is_three_of_a_kind(h)) {
      cnt_3_of_kind = cnt_3_of_kind + 1
    }
  }

  cat('Simulated prob of 2 pair =', cnt_2_pair/ntrials, '\n')
  cat('Simulated prob of 3 of a kind =', cnt_3_of_kind/ntrials, '\n')


  # Exact calculations for 6 card hands out of 52 card decks
  # Two pair
  numerator = choose(13,2)*choose(4,2)*choose(4,2)*choose(11,2)*choose(4,1)*choose(4,1)
  denominator = choose(52,6)
  cat('Exact two pair: numerator =', numerator, ', denominator =', denominator, ', prob
=', numerator/denominator, '\n')


  # Three of a kind
  numerator = choose(13,1)*choose(4,1)*choose(12,3)*choose(4,1)*choose(4,1)*choose(4,1)
  denominator = choose(52,6)
  cat('Exact three of a kind: numerator =', numerator, ', denominator =', denominator,
', prob =', numerator/denominator, '\n')
```

This R code focuses on simulating and calculating probabilities for specific poker hand scenarios—two pairs and three of a kind—using a 52-card deck. The **is_two_pair** and **is_three_of_a_kind** functions examine whether a given hand contains the respective combinations. The main simulation loop runs 100 trials, each time randomly selecting a 6-card hand from the deck and updating counters for the occurrences of two pairs and three of a kind. The simulated probabilities are then calculated by dividing the counts by the total number of trials.

Department of Mechatronics and Control Engineering
University of Engineering and Technology, Lahore Pakistan

This code serves as a practical demonstration of probability concepts in action, illustrating how simulations can provide empirical estimates that align with theoretical calculations based on combinatorics. It offers a hands-on approach to grasp the connection between probability theory and real-world scenarios, specifically in the context of poker hands and card games.

Problem 2:

In class we worked with non-transitive dice:

Blue: 3 3 3 3 3 6; Orange: 1 4 4 4 4 4; White: 2 2 2 5 5 5.

(a) Find the probability that white beats orange, the probability that orange beats blue and the probability that blue beats white. Can you line the dice up in order from best to worst? (Hint: this is why these are called 'non-transitive'.)

(b) Suppose you roll two white dice against two blue dice. What is the probability that the sum of the white dice is greater than the sum of the blue dice?

We will be calculating both the results for both unfair die and fair dies.

Question 2: Solution

```
# Simulated the nontransitive dice. This code is fun because it can compare two or
  three different dice over any number of rolls

cat('Problem 2\n')

db = c(3,3,3,3,3,6) # blue
do = c(1,4,4,4,4,4) # orange
dw = c(2,2,2,5,5,5) # white

playpair = function(die1, die2, nroll, ntrials) {
  wins1 = 0
  ties = 0
  wins2 = 0
  for (j in 1:ntrials) {
    a = sum(sample(die1, nroll, replace=T))
    b = sum(sample(die2, nroll, replace=T))
    if (a > b) {
      wins1 = wins1 + 1
    }
    else if (a == b) {
      ties = ties + 1
    }
    else {
      wins2 = wins2 + 1
    }
  }
  ret = c(wins1, wins2, ties)/ntrials
  return(ret)
}


playthree = function(die1, die2, die3, nroll, ntrials) {
  wins1 = 0
  wins2 = 0
  wins3 = 0
  draws = 0
  for (j in 1:ntrials) {
    a1 = sum(sample(die1, nroll, replace=T))
    a2 = sum(sample(die2, nroll, replace=T))
    a3 = sum(sample(die3, nroll, replace=T))

    if (a1 > a2 && a1 > a3) {
```

```
      wins1 = wins1 + 1
    }
    else if (a2 > a1 && a2 > a3) {
      wins2 = wins2 + 1
    }
    else if (a3 > a1 && a3 > a2) {
      wins3 = wins3 + 1
    }
    else {
      draws = draws + 1
    }
  }

  ret = c(wins1, wins2, wins3, draws)/ntrials
  return(ret)
}

cat("For Unfair Dies\n")
x = play_2die(do, dw, 1, 1000)
cat("Probability of Orange winning ", x[1], "\n")
cat("Probability of White winning ", x[2], "\n")
cat("Probability of draw ", x[3], "\n\n")

y = playall(do, dw, db, 1, 1000)
cat("Probability of Orange winning ", y[1], "\n")
cat("Probability of White winning ", y[2], "\n")
cat("Probability of Blue winning ", y[3], "\n")
cat("Probability of draw ", y[4], "\n\n")


cat("For Fair Dies\n")
dw = c(1:6)
db = c(1:6)
do = c(1:6)
x = play_2die(do, dw, 1, 1000)
cat("Probability of Orange winning ", x[1], "\n")
cat("Probability of White winning ", x[2], "\n")
cat("Probability of draw ", x[3], "\n\n")

y = playall(do, dw, db, 1, 1000)
cat("Probability of Orange winning ", y[1], "\n")
cat("Probability of White winning ", y[2], "\n")
cat("Probability of Blue winning ", y[3], "\n")
cat("Probability of draw ", y[4], "\n\n")
```

This R code simulates the outcomes of dice games involving nontransitive dice, which are dice designed such that one die is more likely to win against another, but the winning relationship is not transitive. The code defines three different dice: blue (db), orange (do), and white (dw). Two functions, **playpair** and **playthree**, simulate the outcomes when two or three dice are rolled, respectively. The results are then expressed as the probabilities of each die winning or a draw after a specified number of trials.

In the "Unfair Dice" section, the code compares the orange and white dice using the **playpair** function, calculating the probabilities of orange winning, white winning, or a draw. It then extends the comparison to all three dice (orange, white, and blue) using the **playthree** function. The "Fair Dice" section resets the dice to fair six-sided dice (1 to 6) and repeats the comparisons.

For instance, the code shows the probability of the orange die winning, the white die winning, and the probability of a draw for both unfair and fair dice scenarios. These simulations provide an engaging way to explore nontransitive relationships in dice games and understand the impact of dice configurations on the probabilities of different outcomes.

Department of Mechatronics and Control Engineering
University of Engineering and Technology, Lahore Pakistan

Problem 3:

Ignoring leap days, the days of the year can be numbered 1 to 365.
Assume that birthdays are equally likely to fall on any day of the year. Consider a group of n people, of which you are not a member. An element of the sample space S will be a sequence of n birthdays (one for each person).
(a) Define the probability function P for S. (This will depend on n.)

(b) Consider the following events: A: "someone in the group shares your birthday" B: "some two people in the group share a birthday" C: "some three people in the group share a birthday" Carefully describe the subset of S that corresponds to each event.

(c) Find an exact formula for $P(A)$. What is the smallest n such that $P(A) > 0.5$?

(d) Justify why n in part (c) is greater than 365 without doing any computation. (We are 2 looking for a short answer giving a heuristic sense of why this is so.)

(e) Use R simulation to estimate the smallest n for which $P(B) > 0.9$. For these simulations, let the number of trials be 10000. (You can reuse your code from Studio 1.)

For this value of n, repeat the simulation a few times to verify that it always gives similar results. Using 10000 trials you saw very little variation in the estimate of $P(B)$. Try this again using 30 trials and verify that the estimated probabilities are much more variable. On your pset, give the results 7 runs of 30 trials using the value of n you just found.

(f) Find an exact formula for $P(B)$.

(g) Use R simulation to estimate the smallest n for which $P(C) > 0.5$. Again use 10000 trials. You will find that two adjacent values of n are equally plausible based on simulations. You may pick either one for your answer.

Problems (a), (b), (c), (d), (f) will be done by hand on a piece of paper. We will use R simulation for problem **(e)** and **(g)** only.

Disclaimer: We will be using an external function for this simulation and the code for this given below. Save the code with the name **"mit18_05_s22_colmatches.r"** *and source it to effectively use. Incase you renamed this file, you will need to change the lines of code in second code that instance or call this file.*

mit18_05_s22_colmatches.r

```
colMatchesHelp = function() {
    cat("# ------------------------------------------------\n")
    cat("Syntax: colMatches(A, n)\n")
    cat("A is an array\n")
    cat("n = the number of matches in each column to check for \n")
    cat("For each column in array A, colMatches() checks if at least\n")
    cat("one entry is repeated n times\n")
    cat("\n")
    cat("Returns a vector of 0's and 1's: 1 means at least\n")
    cat("one entry was repeated at least n times.\n")
    cat("# ------------------------------------------------\n")

}

colMatches = function(A, size_match=2) {
    # See colMatchesHelp()

    #RED_FLAG: We assume A is a vector or 2 dimensional array
```

```
    #RED_FLAG: We don't check that size_match > 0
    A.dim = dim(A)
    if (is.null(A.dim)) {
        #assume A is a column vector
        nrows = length(A)
        ncols = 1
        Asrt= matrix(sort(A), nrow=nrows, ncol=ncols)
    }
    else {
        nrows = A.dim[1]
        ncols = A.dim[2]
        #apply() is an r-magic function. In this case it applies sort to each column. T
o apply to each row use apply(A, 1, sort)
        Asrt = apply(A, 2, sort)
    }
    if (size_match > nrows) {
        #Can't possibly have more matches than rows, return a vector of 0's
        b = rep(0, times=ncols)
    }
    else {
        #Sneaky way to look for runs of size_match in sorted columns
        x = Asrt[size_match:nrows,] == Asrt[1:(nrows-size_match+1),]
        if (ncols == 1) {
            b = 1.0*(sum(x) > 0)
        }
        else if (size_match == nrows) {
            b = as.vector(1.0*x)
        }
        else {
            b = as.vector(1.0*(apply(x, 2, sum) >0))
        }
    }
    return(b)
}
```

The R code defines two functions: **colMatchesHelp** and **colMatches**. The **colMatchesHelp** function serves as a help message, explaining the syntax and purpose of the **colMatches** function, which checks if at least one entry in each column of an array is repeated a specified number of times. The **colMatches** function takes an array **A** and an optional parameter **size_match** (default is 2), sorting each column and identifying runs of repeated entries. It returns a binary vector, where 1 indicates that at least one entry is repeated at least **size_match** times in the corresponding column. The code handles different array dimensions and edge cases, providing an efficient mechanism to detect repeated entries in columns based on user-defined criteria.

Question 3: Solution

```
do_problem_3e = T
do_problem_3g = T

#----------------------------

if (do_problem_3e) {

# colMatches is an 18.05 function. Its source file MIT18_05S22_colMatches.r needs to be
in the R working directory
# You can find the file MIT18_05S22_colMatches.r on our class website.
source('mit18_05_s22_colmatches.r')

  cat('------ Problem 3e ------\n')
# So we can run it multiple times we put the simulation in a function
  simulate_birthday_match2 = function(npeople, ntrials) {
    # Set up the parameters (if wanted, these could also be arguments to the function)
    ndays = 365
```

```
    size_match = 2
    year = 1:ndays
    # Run ntrials -one per column- using sample() and matrix()
    y = sample(year, npeople*ntrials, replace=TRUE)
    trials = matrix(y, nrow=npeople, ncol=ntrials)
    w = colMatches(trials, size_match)
    prob_B = mean(w)

    cat('npeople =', npeople, ', size_match =', size_match, 'ntrials =', ntrials, ', pr
obability of match =', prob_B, '\n')
  }

  simulate_birthday_match2(20, 10000)
  simulate_birthday_match2(30, 10000)
  simulate_birthday_match2(40, 10000)
  simulate_birthday_match2(40, 10000)
  simulate_birthday_match2(40, 10000)
  simulate_birthday_match2(40, 10000)
  simulate_birthday_match2(41, 10000)
  simulate_birthday_match2(41, 10000)
  simulate_birthday_match2(41, 10000)
  simulate_birthday_match2(41, 10000)
}


if (do_problem_3g) {

  cat('\n------ Problem 3g ------\n')

  # This is the same code as for do_problem_3, except size_match is set to 3
  source('mit18_05_s22_colmatches.r')

  # So we can run it multiple times we put the simulation in a function
  simulate_birthday_match3 = function(npeople, ntrials) {
    # Set up the parameters (if wanted, these could also be arguments to the function)
    ndays = 365
    size_match = 3
    year = 1:ndays
    # Run ntrials -one per column- using sample() and matrix()
    y = sample(year, npeople*ntrials, replace=TRUE)
    trials = matrix(y, nrow=npeople, ncol=ntrials)
    w = colMatches(trials, size_match)
    prob_B = mean(w)

    cat('npeople =', npeople, ', size_match =', size_match, 'ntrials =', ntrials, ', pr
obability of match =', prob_B, '\n')
  }



  simulate_birthday_match3(30, 10000)
  simulate_birthday_match3(40, 10000)
  simulate_birthday_match3(60, 10000)
  simulate_birthday_match3(80, 10000)
  simulate_birthday_match3(90, 10000)
  simulate_birthday_match3(85, 10000)
  simulate_birthday_match3(86, 10000)
  simulate_birthday_match3(87, 10000)
  simulate_birthday_match3(87, 10000)
  simulate_birthday_match3(87, 10000)
  simulate_birthday_match3(87, 10000)
  simulate_birthday_match3(88, 10000)
  simulate_birthday_match3(88, 10000)
  simulate_birthday_match3(88, 10000)
  simulate_birthday_match3(88, 10000)
}
```

Simulation Functions: The code defines two simulation functions, **simulate_birthday_match2** and **simulate_birthday_match3**. These functions are designed to simulate the birthday match problem with varying parameters, such as the number of people and the number of trials. They use the **colMatches** function, which is loaded from an external source, to calculate the probability of birthday matches.

Problem 3e Simulation: The code checks if **do_problem_3e** is set to **TRUE** and proceeds to simulate the birthday match problem for different scenarios. It prints the results, showcasing the probability of matches for various group sizes, like 20, 30, 40, and 41 people, over a fixed number of trials.

Problem 3g Simulation: Similarly, the code checks if **do_problem_3g** is set to **TRUE** and simulates the birthday match problem again, but this time with a fixed size of matches (3). It explores different group sizes, such as 30, 40, 60, 80, 90, 85, 86, 87, and 88 people, displaying the resulting probabilities of matches.

Teaching Emphasis: This code serves as a practical demonstration of probability concepts through the birthday match problem. It introduces the idea of simulations to estimate probabilities, and students can observe how changing parameters impacts the likelihood of birthday matches. The external **colMatches** function, which checks for repeated entries in columns, is crucial to the simulations. By varying the number of people and size of matches, students gain insights into the counterintuitive nature of the birthday problem and strengthen their understanding of probability in a hands-on manner.