uc3m | Universidad Carlos III de Madrid

Master Degree in Statistics for Data Science

2024-2025

Resampling and Simulation

Worksheet 3

Marcos Álvarez Martín
Simon Schmetz

Juan Miguel Marín Diazaraque

Madrid - Puerta de Toledo, February 2025

EXERCISE 1: RANDOM PERMUTATIONS

Alice, Bob, Charly, and Dave share an apartment that has only one bathroom. They have decided that every morning from 7 to 8 each one of them can use the bathroom for 15 minutes. The bath turns (1st turn 7 to 7:15, 2nd turn 7:15 to 7:30, 3rd 7:30 to 7:45, 4th turn 7.45 to 8) must be completely random, with the single restriction that Alice must have one of the first turns since she must leave home at 7:50. The have decided to proceed as follows: they select (at random) who takes the 1st turn, then they select who takes the 2nd turn if Alice is not taking any of them, she takes the 3rd and whoever has not yet used the bathroom at 7:45 takes the fourth. If Alice has taken one of the two first turns, they select that random who takes the 3rd turn, and the other flatmate takes the 4th turn.

(a) Is that method fair in the sense that all alowed permutations have the same probability?, why?

This method is **not fair** in the sense that not all allowed permutations have the same probability. The reasons are the following ones:

Understanding the Rules and Constraints

First of all, Alice must take one of the first three turns because she needs to leave by 7:50, so the selection process is:

- **1st turn:** Chosen randomly among the four.
- **2nd turn:** Chosen randomly among the remaining three.
- **3rd turn:**
  - If Alice has not taken the first two turns, she must take the 3rd turn.
  - Otherwise, the 3rd turn is chosen randomly among the two remaining flatmates.
- **4th turn:** Taken by whoever is left.

Now we will count the allowed permutations.

Counting Allowed Permutations

Since Alice must be one of the first three, the allowed permutations are:

- Alice in the 1st turn: $3! = 6$ ways for the others to take the remaining turns.
- Alice in the 2nd turn: $3! = 6$ ways for the others.
- Alice in the 3rd turn: $2! = 2$ ways for the others (since the 4th is fixed).

Then, the total allowed permutations: $6 + 6 + 2 = 14$

Now what we are going to do is to calculate the probabilities for each case:

Analyzing Probabilities

- Alice in the 1st turn:
  - Probability = $\frac{1}{4}$ (chosen randomly among four).
  - Remaining people have $3! = 6$ possible orders, each with probability $\frac{1}{6}$.
  - Probability for each permutation: $\frac{1}{4} \times \frac{1}{6} = \frac{1}{24}$.
- Alice in the 2nd turn:
  - Probability: $\frac{1}{4} \times \frac{1}{3} = \frac{1}{12}$ (since Alice was not chosen for the first turn, then selected for the second).
  - Remaining 2 people have $2! = 2$ possible orders.
  - Probability for each permutation: $\frac{1}{12} \times \frac{1}{2} = \frac{1}{24}$.
- Alice in the 3rd turn:
  - This happens if Alice is not chosen for the 1st or 2nd turn.
  - Probability: $\frac{3}{4} \times \frac{2}{3} = \frac{1}{2}$ (since 3 options left for the 1st, then 2 for the 2nd).
  - The remaining 2 people have only 1 possible order (4th fixed).
  - Probability for each permutation: $\frac{1}{2}$.

To answer to the question that is why this procedure is not fair, we can say that it is not fair because the probabilities for the permutations are not equal:

- Permutations where Alice is 1st or 2nd have a probability of $\frac{1}{24}$.
- Permutations where Alice is 3rd have a much higher probability.

So, in conclusion, we can say that this method is not fair because the probabilities are not uniforma across all allowed permutations. Specifically, Alice's position in the sequence affects the distribution of probabilities for the remaining participants. A truly fair method would ensure that each of the allowed permutations has an equal probability of $\frac{1}{14}$.

(b) Build a fair algorithm to distribute the turns (with the given restriction

To build a fair algorithm that distributes the turns while respecting the restriction that Alice must take one of the first three turns, we can proceed as follows:

The basis is that:

- Alice must take one of the first three turns.
- All allowed permutations must have the same probability.

We think that the best approach is the following:

- Generate all allowed permutations: List all permutations where Alice is in one of the first three positions.
- Select randomly from allowed permutations: Choose one permutation uniformly at random.

Now, we can reproduce this problem with simulation in R and Python:

R

CODE 1. Random Permutations in R

```r
library(combinat)
set.seed(100428853)  # For reproducibility

# List of flatmates
flatmates <- c("Alice", "Bob", "Charly", "Dave")

# Function to get all allowed permutations
get_allowed_permutations <- function() {
  # Initialize list to store allowed permutations
  allowed_perms <- list()

  # Alice in 1st position
  others <- setdiff(flatmates, "Alice")
  perms <- permn(others)
  allowed_perms <- c(allowed_perms, lapply(perms, function(x) c("
      Alice", x)))

  # Alice in 2nd position
  perms <- permn(others)
  allowed_perms <- c(allowed_perms, lapply(perms, function(x) c(x
      [1], "Alice", x[2:3])))

  # Alice in 3rd position
```

```r
  perms <- permn(others[1:2])  # Only 2 permutations left for the other two
  allowed_perms <- c(allowed_perms, lapply(perms, function(x) c(x, "Alice", others[3])))

  return(allowed_perms)
}

# Get all allowed permutations
allowed_perms <- get_allowed_permutations()

# Select one permutation at random
selected_perm <- sample(allowed_perms, 1)[[1]]

# Display the selected permutation with corresponding times
times <- c("7:00-7:15", "7:15-7:30", "7:30-7:45", "7:45-8:00")
schedule <- data.frame(Time = times, Person = selected_perm)

print(schedule)
```

```
       Time Person
1 7:00-7:15   Dave
2 7:15-7:30  Alice
3 7:30-7:45    Bob
4 7:45-8:00 Charly
```

Python

CODE 2. Random Permutations in Python

```python
import random
from itertools import permutations

# Set the seed for reproducibility
random.seed(100428853)

# List of flatmates
flatmates = ["Alice", "Bob", "Charly", "Dave"]

# Function to get all allowed permutations
def get_allowed_permutations():
    allowed_perms = []

    # Alice in the first position
    others = [f for f in flatmates if f != "Alice"]
    perms = list(permutations(others))
    allowed_perms.extend([("Alice",) + p for p in perms])
```

```python

    # Alice in the second position
    allowed_perms.extend([(p[0], "Alice", p[1], p[2]) for p in perms
        ])

    # Alice in the third position
    allowed_perms.extend([(p[0], p[1], "Alice", p[2]) for p in perms
        ])

    return allowed_perms

# Get all allowed permutations
allowed_perms = get_allowed_permutations()

# Randomly select one permutation
selected_perm = random.choice(allowed_perms)

# Assign times to each person
times = ["7:00-7:15", "7:15-7:30", "7:30-7:45", "7:45-8:00"]
schedule = list(zip(times, selected_perm))

# Print the schedule
print("Bathroom Schedule:")
for time, person in schedule:
    print(f"{time} - {person}")
```

```
Bathroom Schedule:
7:00-7:15 - Alice
7:15-7:30 - Charly
7:30-7:45 - Dave
7:45-8:00 - Bob
```

The results from the simulation are correct due to it satisfies the constraint, but the orders are different.

EXERCISE 2: COMPOUND POISSON PROCESS

The number of purchase at a webpage follows a non homogeneous Poisson process with intensity function $\lambda(x) = x$ during the first 10 days and $\lambda(x) = 10$ purchases per day ever after.

(a) Simulate 10000 processes until 100 purchases are executed. What is the average time until those 100 purchases?

In this statement we have to simulate a non homogeneus Poisson process, with the next description:

The purchase rate is time-dependent:

- For the first 10 days: $\lambda(x) = x$.
- After 10 days: $\lambda(x) = 10$.

What we have to do is to sumulate 10000 processes until 100 purchases are executed and calculate the average time to reach 100 purchases.

This approach will be followed:

1. For $x \leq 10$: The event rate is increasing linearly, so the waiting times will vary accordingly.
2. For $x > 10$: The rate becomes constant, so the waiting times follow an exponential distribution with mean 1/10.
3. Simulate the process 10000 times, record the time taken for 100 purchases in each simulation.
4. Calculate the average time across all simulations.

We will use the inverse transform sampling method for the non homogeneous rate.

R

CODE 3. Compound Poisson Process in R

```
set.seed(100428853)  # For reproducibility

# Number of simulations and target purchases
num_simulations <- 10000
```

```r
target_purchases <- 100

# Function to simulate one process
simulate_process <- function() {
  purchases <- 0
  time <- 0

  while (purchases < target_purchases) {
    if (time <= 10) {
      #   (t) = t for t     10 -> Inverse CDF method for waiting time
      u <- runif(1)
      waiting_time <- -time + sqrt(time^2 + 2 * u)
    } else {
      #   (t) = 10 for t > 10 -> Exponential waiting time
      waiting_time <- rexp(1, rate = 10)
    }

    time <- time + waiting_time
    purchases <- purchases + 1
  }

  return(time)
}

# Simulate the process for 10000 times
times <- replicate(num_simulations, simulate_process())

# Calculate the average time
average_time <- mean(times)

cat("Average time to reach 100 purchases:", round(average_time, 2), "days\n")
```

```
Average time to reach 100 purchases: 10.1 days
```

Python

CODE 4. Compund Poisson Process in Python

```python
import numpy as np

# Set seed for reproducibility
np.random.seed(100428853)

# Number of simulations and target number of purchases
num_simulations = 10000
target_purchases = 100
```

```python

# Function to simulate one process
def simulate_process():
    purchases = 0
    time = 0

    while purchases < target_purchases:
        if time <= 10:
            #   (t) = t for t     10, waiting time from inverse CDF method
            u = np.random.uniform()
            waiting_time = (-time + np.sqrt(time**2 + 2*u))  # Solving F(T) = U for T
        else:
            #   (t) = 10 for t > 10, waiting time from exponential distribution
            waiting_time = np.random.exponential(1/10)

        time += waiting_time
        purchases += 1

    return time

# Simulate the process for 10000 times
times = [simulate_process() for _ in range(num_simulations)]

# Calculate the average time
average_time = np.mean(times)

print(f"Average time to reach 100 purchases: {average_time:.2f} days")
```

```
Average time to reach 100 purchases: 10.09 days
```

The explanation of the code is as follows:

- For $x \leq 10$, since the rate is $\lambda(x) = x$, the CDF is $F(x) = \frac{X^2}{2}$. We solve $U = F(X)$ to get the waiting time.
- For $x > 10$, the waiting time follows an exponential distribution with mean $\frac{1}{10}$.
- Loop until 100 purchases for each simulation, updating the time acordingly.
- Store the total time for each simulation and then compute the average.

So, basically the results are the same and on average, it takes 10.1 days for the webpage to reach 100 purchases.

(b) Three different products are hold, one for 200 euros, another for 300 euros, and the third one for 500 euros. Half of the purchases correspond to the first product, 30% to the second, and the remaining to the third. What is the expected time until they sell products for a total value of 25000 euros?

In this statement we have to simulate the sales process and calculate the expected time until the total sales reach 25000€.

The plan is the next:

1. Simulate sales for 10000 runs until the total reach 25000 euros.
2. For each sale:
   - Determine the product sold using the given probabilities.
   - Add the product price to the cummulative total.
   - Count the number of sales and time taken.
3. Calculate the average time over all simulations.

Now, we are going to do the exercise in R and Python and see what happens.

R

CODE 5. Compound Poisson Process part (b) in R

```

set.seed(100428853)  # Set seed for reproducibility

# Simulation parameters
num_simulations <- 10000
target_sales <- 25000
product_prices <- c(200, 300, 500)
product_probs <- c(0.5, 0.3, 0.2)

# Store times and number of sales to reach the target for each
    simulation
times_to_target <- numeric(num_simulations)
sales_to_target <- numeric(num_simulations)

# Simulation loop
for (i in 1:num_simulations) {
  total_sales <- 0
  time <- 0
  num_sales <- 0

  while (total_sales < target_sales) {
```

```
    # Calculate the purchase rate
    rate <- if (time <= 10) time else 10

    # Calculate waiting time based on the rate
    if (time <= 10) {
      # Nonhomogeneous rate:   (t) = t, use inverse CDF method
      u <- runif(1)
      waiting_time <- (-time + sqrt(time^2 + 2 * u))
    } else {
      # Homogeneous rate:   (t) = 10, use exponential distribution
      waiting_time <- rexp(1, rate = 10)
    }

    # Update time
    time <- time + waiting_time

    # Simulate the sale
    product <- sample(product_prices, 1, prob = product_probs)
    total_sales <- total_sales + product
    num_sales <- num_sales + 1
  }

  # Store results
  times_to_target[i] <- time
  sales_to_target[i] <- num_sales
}

# Calculate the expected time and number of sales
expected_time <- mean(times_to_target)
expected_sales <- mean(sales_to_target)

cat("Expected time until 25000 euros in sales:", round(expected_time, 2), "days\n")
cat("Expected number of sales to reach 25000 euros:", round(expected_sales, 2), "\n")
```

```
Expected time until 25000 euros in sales: 9.3 days
Expected number of sales to reach 25000 euros: 86.62
```

Python

CODE 6. Compound Poisson Process part (b) in Python

```
import numpy as np

# Set seed for reproducibility
np.random.seed(100428853)

```

```python
# Simulation parameters
num_simulations = 10000
target_sales = 25000
product_prices = [200, 300, 500]
product_probs = [0.5, 0.3, 0.2]

# Store times and number of sales to reach the target for each simulation
times_to_target = []
sales_to_target = []

for _ in range(num_simulations):
    total_sales = 0
    time = 0
    num_sales = 0

    while total_sales < target_sales:
        # Calculate the purchase rate
        rate = time if time <= 10 else 10

        # Calculate waiting time based on the rate
        if time <= 10:
            # Nonhomogeneous rate:  (t) = t, use inverse CDF method
            u = np.random.uniform()
            waiting_time = (-time + np.sqrt(time**2 + 2*u))
        else:
            # Homogeneous rate:  (t) = 10, use exponential distribution
            waiting_time = np.random.exponential(1/10)

        # Update time
        time += waiting_time

        # Simulate the sale
        product = np.random.choice(product_prices, p=product_probs)
        total_sales += product
        num_sales += 1

    # Store results
    times_to_target.append(time)
    sales_to_target.append(num_sales)

# Calculate the expected time and number of sales
expected_time = np.mean(times_to_target)
expected_sales = np.mean(sales_to_target)

print(f"Expected time until 25000 euros in sales: {expected_time:.2f} days")
print(f"Expected number of sales to reach 25000 euros: {expected_sales:.2f}")
```

```
Expected time until 25000 euros in sales: 9.30 days
Expected number of sales to reach 25000 euros: 86.64
```

And finally, we can conclude that, on average, it will take aproximately 87 sales and 9.3 days to reach a total of 25000€ given the distribution of products above.

The difference between the result in Python and in R is very small.

EXERCISE 3: PRICING EUROPEAN OPTIONS

European options may only be exercised at expiration. That is, if we buy today a European call option with strike price $k = 100$ and maturity $t_m = 1$ year, in one year (at maturity) we have the right to buy the asset for the fixed strike price $k = 100$, which we will do if the asset price at that precise moment is greater than k. If the price of the asset at that moment is below k, we will not exercise the option. Suppose we want to price a European call option with the initial asset price $S(0) = 100$, strike price $k = 100$, risk free rate $r = 0.02$, volatility $\sigma = 0.25$, and maturity $t_m = 1$ year. The risk neutral pricing process is:$S(t) = S(0)\exp\left((r - \sigma^2/2)t + \sigma Z\sqrt{t}\right)$ where Z ~ N(0,1). Use MC = 10000 simulations to price the option and give a 95% CI on the price. The option will only be exercised if the asset's price at maturity is greater than 100, so its payoff will be max $S(t_m) - k, 0$, while the price is the expected payoff. Observe that the price is to be paid today, so it must be given in today's price of money, and we can use the risk-free rate to determine today's price of 100 monetary units in one year, which is 100$\exp(-r)$ = 98.01987.

The goal is to simulate the price of the underlying asset at maturity (one year) using the given formula for the asset price evolution. Based on the simulated final asset prices, you compute the payoff for the call option (which is the maximum of the asset price minus the strike price or zero) and average these payoffs over 10000 simulations. The price of the option is the discounted expected payoff, where the discount factor is based on the risk-free rate.

Now we can solve this exercise in R and Python as with the previous exercises.

R

CODE 7. Pricing European options in R

```

# Parameters
S0 <- 100  # Initial asset price
K <- 100   # Strike price
r <- 0.02  # Risk-free rate
sigma <- 0.25  # Volatility
T <- 1  # Time to maturity (1 year)
num_simulations <- 10000  # Number of Monte Carlo simulations

# Simulate asset prices at maturity
payoffs <- numeric(num_simulations)
for (i in 1:num_simulations) {
  Z <- rnorm(1)  # Standard normal random variable
```

```r
  ST <- S0 * exp((r - 0.5 * sigma^2) * T + sigma * sqrt(T) * Z)  # Asset price at maturity
  payoffs[i] <- max(ST - K, 0)  # Call option payoff
}

# Calculate the option price (discounted expected payoff)
discount_factor <- exp(-r * T)
option_price <- discount_factor * mean(payoffs)

# Compute 95% confidence interval
std_error <- sd(payoffs) / sqrt(num_simulations)
conf_interval <- 1.96 * std_error

# Output results
cat(sprintf("Option Price: %.4f\n", option_price))
cat(sprintf("95%% Confidence Interval: (%.4f, %.4f)\n", option_price - conf_interval, option_price + conf_interval))
```

```
Option Price: 11.2294
95% Confidence Interval: (10.8759, 11.5829)
```

Python

CODE 8. Pricing European options in Python

```python

import numpy as np

# Parameters
S0 = 100  # Initial asset price
K = 100  # Strike price
r = 0.02  # Risk-free rate
sigma = 0.25  # Volatility
T = 1  # Time to maturity (1 year)
num_simulations = 10000  # Number of Monte Carlo simulations

# Simulate asset prices at maturity
payoffs = []
for _ in range(num_simulations):
    Z = np.random.normal(0, 1)  # Standard normal random variable
    ST = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * Z)  # Asset price at maturity
    payoff = max(ST - K, 0)  # Call option payoff
    payoffs.append(payoff)

# Calculate the option price (discounted expected payoff)
discount_factor = np.exp(-r * T)
option_price = discount_factor * np.mean(payoffs)
```

```python

# Compute 95% confidence interval
std_error = np.std(payoffs) / np.sqrt(num_simulations)
conf_interval = 1.96 * std_error

# Output results
print(f"Option Price: {option_price:.4f}")
print(f"95% Confidence Interval: ({option_price - conf_interval:.4f
    }, {option_price + conf_interval:.4f})")
```

```
Option Price: 10.8355
95% Confidence Interval: (10.4847, 11.1863)
```

The option price is approximately 10.84 euros (Python result) or 11.23 euros (R result), representing the estimated price of the European call option today based on 10,000 simulations. The 95% confidence interval for the Python result suggests that the true option price lies between 10.48 euros and 11.19 euros, while the 95% confidence interval for the R result suggests it lies between 10.88 euros and 11.58 euros. Both intervals indicate a small range of uncertainty around the option price estimate, with the two results being quite close to each other.

FINAL CONCLUSION

This task involved solving three distinct problems that required different approaches and techniques. The first problem focused on random permutations for assigning bathroom turns among Alice, Bob, Charly, and Dave, with the restriction that Alice must take one of the first turns. We analyzed whether the proposed method was fair and concluded that it wasn't, as the randomness of the selection process was influenced by the specific restriction on Alice. We also provided a fair algorithm that respected this constraint while ensuring all flatmates had an equal chance of getting their turn.

The second problem was a simulation of a compound Poisson process for modeling the number of purchases on a webpage, where we simulated 10,000 processes to calculate the average time required to complete 100 purchases and the expected time to generate 25,000 euros in sales. This demonstrated how to handle time-dependent intensity functions and perform stochastic simulations in such scenarios.

The final problem was about pricing a European call option using Monte Carlo simulations. We estimated the option price and its 95% confidence interval by simulating the asset price evolution based on a risk-neutral model. The Monte Carlo method proved effective in calculating the option price and assessing the uncertainty around the estimate.

Overall, this task highlighted the use of random processes, stochastic modeling, and simulations to solve real-world problems, from ensuring fairness in everyday situations to modeling financial instruments in the context of options pricing.