UNIVERSITAT POLITÈCNICA DE CATALUNYA - ETSETB

Big Data and Programming in R

Basic exercises

Author:
Víctor Moreno Borràs

Professor:
Josep Maria Aroca Farrerons

January 10, 2024

Contents

1 Second degree polynomial equation

1.1 Script

```
compute_roots <- function(a, b, c) {
  # Calculate the discriminant
  discriminant <- b^2 - 4*a*c

  # Check if the discriminant is non-negative (real roots)
  if (discriminant >= 0) {
    # Calculate the two real roots
    root1 <- (-b + sqrt(discriminant)) / (2*a)
    root2 <- (-b - sqrt(discriminant)) / (2*a)
    return(c(root1, root2))
  } else {
    # Calculate the two complex roots
    root1 <- (-b + sqrt(as.complex(discriminant))) / (2*a)
    root2 <- (-b - sqrt(as.complex(discriminant))) / (2*a)
    return(c(root1, root2))
  }
}
```

Listing 1: Compute roots function

1.2 Output

```
> # Example real roots:
> a <- 1
> b <- -3
> c <- 2
> roots <- compute_roots(a, b, c)
> print(paste("Roots:", roots))
[1] "Roots: 2" "Roots: 1"
>
> # Example complex roots:
> a <- 4
> b <- -1
> c <- 5
> roots <- compute_roots(a, b, c)
> print(paste("Roots:", roots))
[1] "Roots: 0.125+1.11102430216445i" "Roots: 0.125-1.11102430216445i"
```

Listing 2: Example output

2 Prime numbers

2.1 Script

```
# Function to check if a number is prime
isprime <- function ( x ) {
  if(x==1) return(FALSE)
  if(x==2) return(TRUE)
  M<-max(c(floor(sqrt(x)),2))
  for(k in 2:M){
    if(x%%k ==  0) return(FALSE)
  }
  return(TRUE)
}
# Function to generate all prime numbers not exceeding a given value n
allprimes <- function ( n ) {
  primes <- c ()
  for ( i in 2: n ) {
    if ( isprime ( i ) ) {
      primes <- c ( primes , i )
    }
  }
  return ( primes )
}
```

Listing 3: Prime numbers functions

2.2 Output

```
> # Examples of usage:
> isprime(907)
[1] TRUE
> isprime(908)
[1] FALSE
> allprimes(80)
 [1] 2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
```

Listing 4: Example output

3 Basic plot

3.1 Script

```
# Define the range of x values
x_values <- seq(-3, 3, length.out = 100)

# Calculate y values for cosh(x) and sinh(x)
y_cosh <- cosh(x_values)
y_sinh <- sinh(x_values)

# Calculate asymptotic lines
asymptotic_upper <- exp(x_values)/2
asymptotic_lower <- -exp(x_values)/2

# Plot the functions
plot(x_values, y_cosh, type = "l", col = "green", ylab = "y", xlab = "
    x", ylim = c(-3, 3))
lines(x_values, y_sinh, col = "red")
lines(x_values, asymptotic_upper, col = "blue", lty = 2)
lines(x_values, asymptotic_lower, col = "blue", lty = 2)
title("Hyperbolic Functions")
legend("topleft", legend = c("cosh(x)", "sinh(x)", "asymptotic Lines")
    , col = c("green", "red", "blue"), lty = c(1, 1, 2))
```

Listing 5: plot functions

3.2 Output

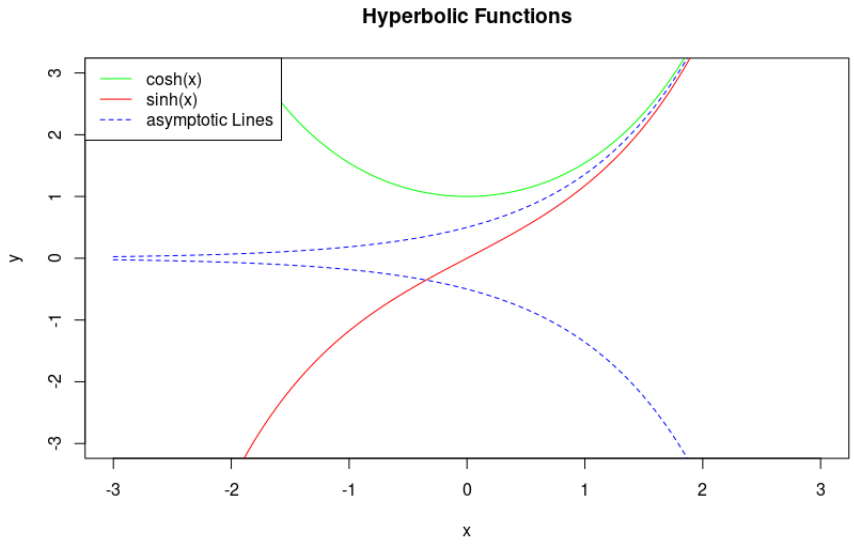

Figure 1: Hyperbolic functions plot

4 Taylor polynomials

4.1 Exponential

4.1.1 Script

```r
# Function to compute the n-order Taylor polynomial for exp(x) at x=0
texp <- function(n, x) {
  sum(x^(0:n) / factorial(0:n))
}

# Define the range of x values and calculate y values for exp(x)
x_values <- seq(-2, 2, length.out = 100)
y_exp <- exp(x_values)

# Plot exp(x) and its Taylor polynomials of different orders
plot(x_values, y_exp, type = "l", col = "blue", lty = 2, ylim = c(0,
    8), ylab = "y", xlab = "x")
for (order in 1:4) {
  y_taylor <- sapply(x_values, function(x) texp(order, x))
  lines(x_values, y_taylor, col = rainbow(4)[order], lty = 1)
}
title("Exp Taylor Functions")
legend("topright", legend = c("exp(x)", "Taylor (order 1)", "Taylor (
    order 2)", "Taylor (order 3)", "Taylor (order 4)"),col = c("blue",
    rainbow(4)), lty = c(2, 1, 1, 1, 1))
```

Listing 6: Taylor exp polynomials

4.1.2 Output

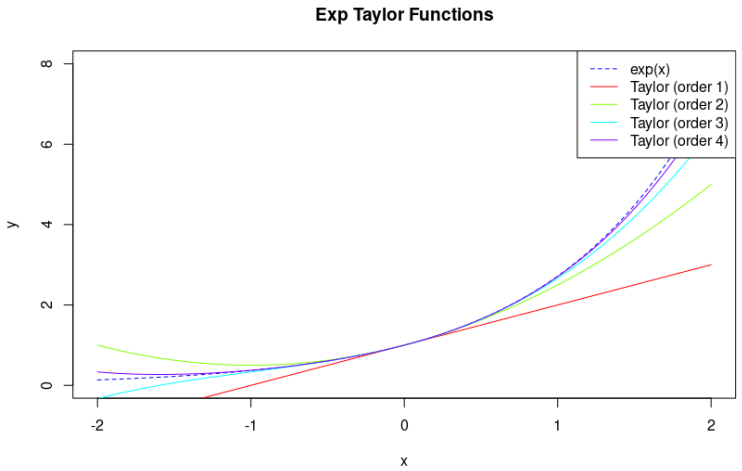

Figure 2: Exponential Taylor functions plot

4.2 Cosinus and Sinus

4.2.1 Script

```
# Function to compute the n-order Taylor polynomial for cos(x) at x=0
tcos <- function(n, x) {
  sum((-1)^(0:n) * x^(2*(0:n)) / factorial(2*(0:n)))
}

# Function to compute the n-order Taylor polynomial for sin(x) at x=0
tsin <- function(n, x) {
  sum((-1)^(0:n) * x^(1 + 2*(0:n)) / factorial(1 + 2*(0:n)))
}

# Define the range of x values and calculate y values for cos(x) and
   sin(x)
x_values <- seq(-2*pi, 2*pi, length.out = 200)
y_cos <- cos(x_values)
y_sin <- sin(x_values)

# Plot cos(x) and its Taylor polynomials of different orders
par(mfrow = c(1, 2))
plot(x_values, y_cos, type = "l", col = "blue", lty = 2, ylim = c(-2,
   2), ylab = "y", xlab = "x", main = "Cosine Function")

for (order in 1:4) {
  y_taylor_cos <- sapply(x_values, function(x) tcos(order, x))
  lines(x_values, y_taylor_cos, col = rainbow(4)[order], lty = 1)
}

legend("topright", legend = c("cos(x)", "Taylor (order 1)", "Taylor (
   order 2)", "Taylor (order 3)", "Taylor (order 4)"), col = c("blue",
    rainbow(4)), lty = c(2, 1, 1, 1, 1))

# Create a new plot for sin(x) and its Taylor polynomials
plot(x_values, y_sin, type = "l", col = "red", lty = 2, ylim = c(-2,
   2), ylab = "y", xlab = "x", main = "Sine Function")

for (order in 1:4) {
  y_taylor_sin <- sapply(x_values, function(x) tsin(order, x))
  lines(x_values, y_taylor_sin, col = rainbow(4)[order], lty = 1)
}

legend("topright", legend = c("sin(x)", "Taylor (order 1)", "Taylor (
   order 2)", "Taylor (order 3)", "Taylor (order 4)"), col = c("red",
   rainbow(4)), lty = c(2, 1, 1, 1, 1))
```

Listing 7: Taylor cos and sin polynomials

4.2.2 Output

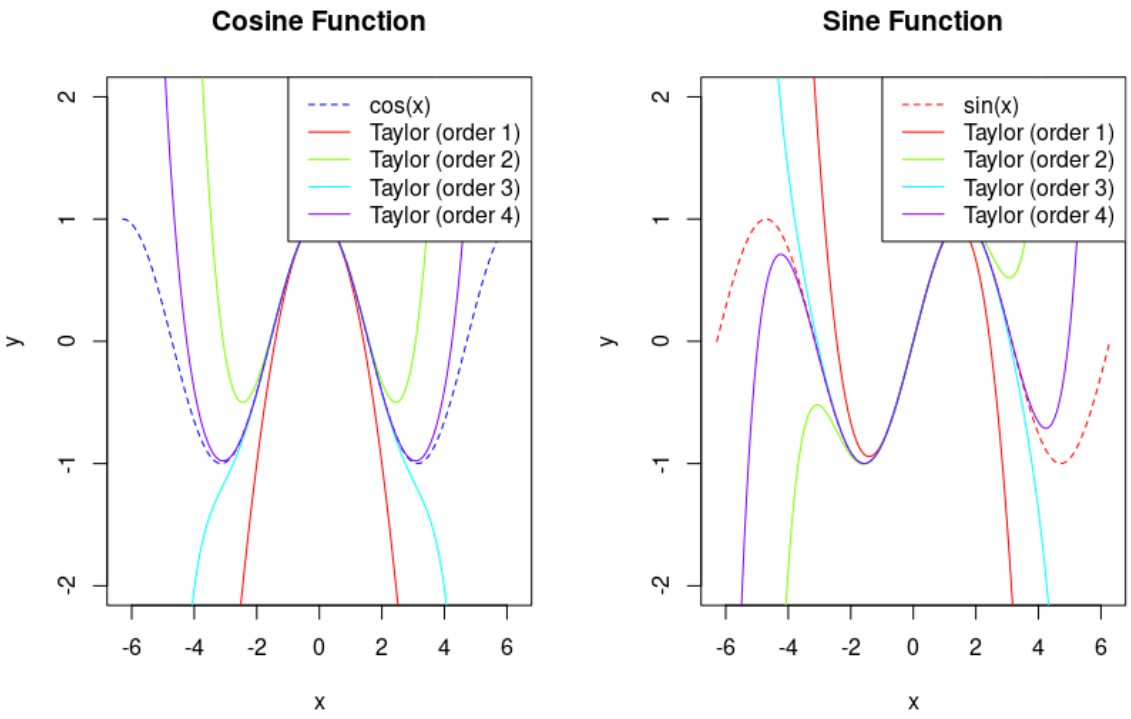

Figure 3

5 Determinant

5.1 Part 1

5.1.1 Script

```
det_rec <- function(matrix, index, is_row) {
  cat("Your matrix:\n")
  print(matrix)

  det_value <- det(matrix)
  cat("The determinant of your matrix is:", det_value, "\n")

  if (is_row) {
    cat("Developed by row", index, ": ")
    for (i in 1:ncol(matrix)) {
      if (i != index) {
        coefficient <- matrix[index, i]
        sub_matrix <- matrix[-index, -i, drop = FALSE]
        det_sub <- det(sub_matrix)
        cat(coefficient, "*", det_sub)

        if (i < ncol(matrix) - 1) {
          cat(" + ")
        }
      }
    }
  } else {
    cat("Developed by column", index, ": ")
    for (i in 1:nrow(matrix)) {
      if (i != index) {
        coefficient <- matrix[i, index]
        sub_matrix <- matrix[-i, -index, drop = FALSE]
        det_sub <- det(sub_matrix)
        cat(coefficient, "*", det_sub)

        if (i < nrow(matrix) - 1) {
          cat(" + ")
        }
      }
    }
  }

  cat("\n")
}
```

Listing 8: Calculation determinant

5.1.2 Output

```
> data <- c(4, 4, 3, 0, -3, -3, 1, 2, -1, 2, 0, -1, -1, 1, 2, 3)
> M <- matrix(data, nrow = 4, byrow = TRUE)
> det_rec(M, 3, FALSE)
Your matrix:
     [,1] [,2] [,3] [,4]
[1,]    4    4    3    0
[2,]   -3   -3    1    2
[3,]   -1    2    0   -1
[4,]   -1    1    2    3
The determinant of your matrix is: -89
Developed by column 3 : 3 * -31 + 1 * 44 + 2 * -24
```

Listing 9: Output example

5.2 Part 2

5.2.1 Script

```
mydet <- function(mat) {
  n <- nrow(mat)

  if (n == 1) {
    return(mat[1, 1])
  }else{
    det_val<-0
    for(j in 1:n){
      sign <- (-1)^(1+j)
      minor_mat <- mat[-1,-j,drop=FALSE]
      det_val <- det_val + sign * mat[1,j] * mydet(minor_mat)
    }
    return(det_val)
  }
}
```

Listing 10: mydet functions

5.2.2 Output

```
# Example usage:
M <- matrix(c(4, 4, 3, 0, -3, -3, 1, 2, -1, 2, 0, -1, -1, 1, 2, 3),
   nrow = 4, byrow = TRUE)
result <- mydet(M)
cat("The determinant of the matrix is:", result, "\n")
```

Listing 11: Output example

6 Fabius Sieve

6.1 Script

```
fabius_sieve <- function(N) {
  sequence <- 1:N
  for (i in 2:N) {
    sequence <- sequence[-(i * (1:(N %/% i)))]
  }
  return(sequence)
}
```

Listing 12: Function to generate Fabius sieve sequence

6.2 Experiments

```
N <- 1000
fabius_sequence <- fabius_sieve(N)

plot(fabius_sequence, type = "b", pch = 19, col = "blue",main = "
    Fabius Sieve Sequence", xlab = "Index", ylab = "Value")
grid()
```

Listing 13: Experiment 1:

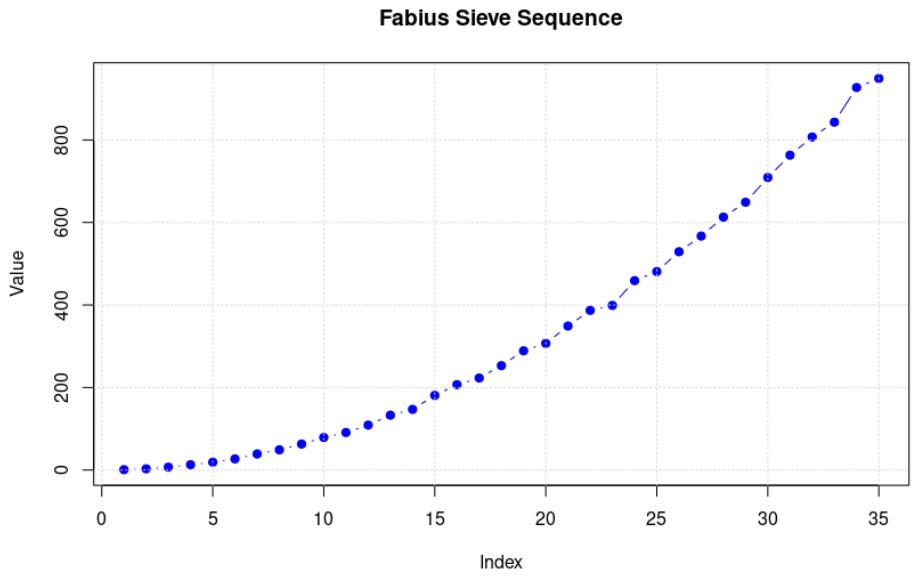

Figure 4: Fabius sieve sequence for N=1000

```r
N <- 100000
fabius_sequence <- fabius_sieve(N)

plot(fabius_sequence, type = "b", pch = 19, col = "blue",main = "
    Fabius Sieve Sequence", xlab = "Index", ylab = "Value")
grid()
```

Listing 14: Experiment 2

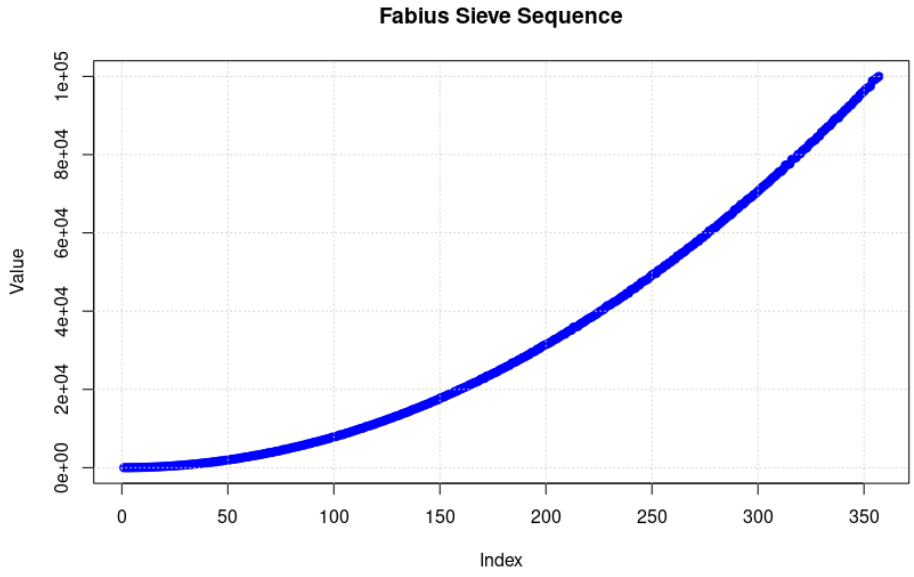

Figure 5: Fabius sieve sequence for N=100000

6.3 Estimation of coefficients an an^k

```r
data <- data.frame(index = seq_along(fabius_sequence), value = fabius_
    sequence)

# Step 1: Regression to estimate k
fit_k <- lm(log(value) ~ log(index), data = data)
k <- coef(fit_k)[2]

# Step 2: Regression to estimate a (using the guessed value for k)
data$log_index <- log(data$index)
data$log_value <- log(data$value)
data$predicted <- exp(predict(fit_k, newdata = data))

fit_a <- lm(log(value) ~ log(index) - 1, data = data)
a <- exp(coef(fit_a))
```

Listing 15: Estimation

$$\mathbf{a = 2} \qquad \mathbf{a = 7.05}$$

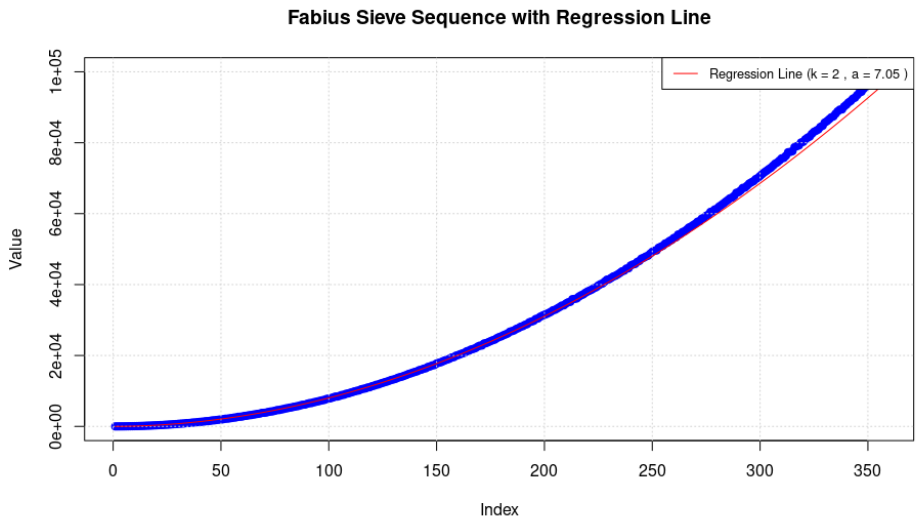

Figure 6: Plot of the regression line together with the sequence

6.4 Comments

As depicted in Figure 6, the parameters a and k identified exhibit a close approximation to the Fabius sequence. However, it is noteworthy that for exceedingly large values, there is a slight divergence observed compared to the sequence calculated from the provided data.

7 Lucky ticket

7.1 Script

```
# Function to transform a non-negative integer into a list of digits
  int_to_digits <- function(num) {
    num_str = strsplit(as.character(num), '')[[1]]
    if(length(num_str)<6){
      ceros_faltantes <- 6 - length(num_str)
      return (c(rep(0, ceros_faltantes), as.numeric(num_str)))
    } else{
      return (as.numeric(num_str[1:6]))
    }
  }

# Function to check if a number is lucky
is_lucky <- function(num) {
  digits <- int_to_digits(num)
  sum_first_half <- sum(digits[1:3])
  sum_second_half <- sum(digits[4:6])
  return(sum_first_half == sum_second_half)
}
```

Listing 16: Function to consider if a number is lucky

7.2 First Exercise

```
# Exercise 1:Simulation of the probability of getting a lucky number

# Simulation
set.seed(123)
num_simulations <- 1000
lucky_numbers_sim <- sum(replicate(num_simulations, is_lucky(sample
   (0:999999, num_simulations, replace = TRUE))))

# Estimate probability from simulation
probability_sim <- lucky_numbers_sim / num_simulations
cat("Estimated probability from simulation:", probability_sim, "\n")
```

Listing 17: Exercise one

```
1 Exact probability by counting on the whole population: 0.06
```

Listing 18: Output 1

7.3 Second Exercise

```r
# Exercise two: Exact probability by counting on the whole population
# Counting lucky numbers in the whole population
all_numbers <- 0:999999
lucky_numbers_exact <- sum(sapply(all_numbers, is_lucky))
# Exact probability
probability_exact <- lucky_numbers_exact / length(all_numbers)
cat("Exercise two: Exact probability by counting on the whole
    population:", probability_exact, "\n")
```

Listing 19: Exercise 2

```
 Exact probability by counting on the whole population: 0.055252
```

Listing 20: Output 2

7.4 Third Exercise

```r
# Exercise three: Exact probability using probability
# theory (random variables and convolution theorem)
# Define probability mass function for a single digit (0 to 9)
prob_single_digit <- rep(1/10, 10)
# Convolve the probability mass function for three digits
prob_three_digits <- convolve(prob_single_digit, prob_single_digit,
    type = "open")
prob_three_digits <- convolve(prob_three_digits, prob_single_digit,
    type = "open")

# Probability of being lucky
probability_theory <- sum(prob_three_digits^2)
cat("Exact probability using probability theory (random variables and
    convolution theorem):", probability_theory, "\n")
```

Listing 21: Exercise 3

```
Exact probability using probability theory: 0.055252
```

Listing 22: Output 3

7.5 Comments

The R script defines a function to identify lucky numbers based on digit sums. In the first exercise, simulation yields an estimated probability of approximately 0.06, while exact counting on the whole population (Exercise 2) gives a probability of 0.055252. The third exercise applies probability theory, producing an exact probability of 0.055252. Notably, the simulation aligns closely with both exact counting and theoretical results.

8 Random Walks

8.1 1 Dimension

8.1.1 Script

```
# Generate a 1D random walk with regular left-right displacements
generate_1d_walk_regular <- function(N) {
  steps <- sample(c(-1, 1), N, replace = TRUE)
  positions <- cumsum(steps)
  return(positions)
}

# Generate a 1D random walk with normal (gaussian) displacements
generate_1d_walk_normal <- function(N) {
  steps <- rnorm(N)
  positions <- cumsum(steps)
  return(positions)
}
```

Listing 23: Functions to generate random 1D walks

8.1.2 Output

```
experiments = 4
steps = 10000

par(mfrow = c(3, experiments))
for (i in 1:experiments){
    plot(generate_1d_walk_regular(steps), type = 'l', col = i)
}
for (i in 1:experiments){
    plot(generate_1d_walk_normal(steps), type = 'l', col = i)
}
```

Listing 24: Ploting random walks

The following figures depict three rows of five different experiments each for both random 1D regular walks and random normal walks. Each row corresponds to a distinct value of $N(number of steps)$; specifically $N = 100$, $N = 1000$ and $N = 100000$. Within each row, five separate instances of the respective random walk types are showcased, offering a visual representation of how the trajectories evolve with increasing numbers of steps.

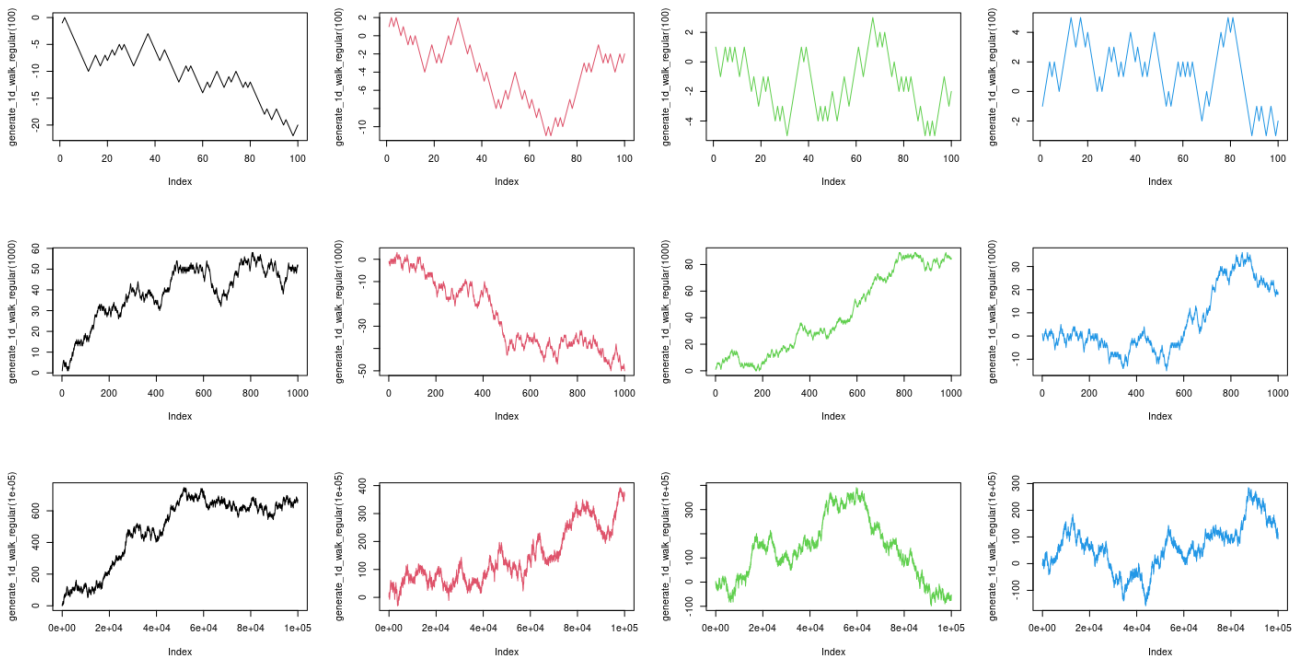

Figure 7: Five 1D random regular walks for 3 different N

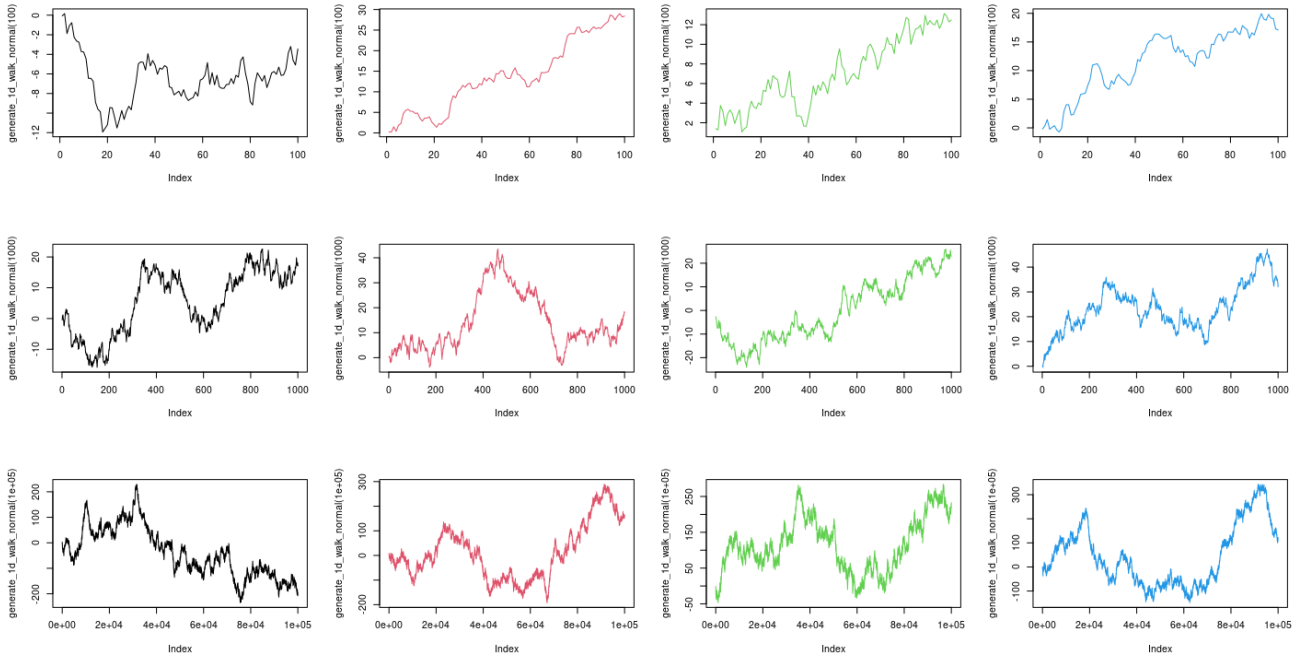

Figure 8: Five 1D random normal walks for 3 different N

8.2 2 Dimensions

8.2.1 Script

```
# Generate a 2D random walk with regular left-right displacements
generate_2d_walk_regular <- function(N) {
  x_steps <- sample(c(-1, 1), N, replace = TRUE)
  y_steps <- sample(c(-1, 1), N, replace = TRUE)
  x_positions <- cumsum(x_steps)
  y_positions <- cumsum(y_steps)
  return(data.frame(x = x_positions, y = y_positions))
}

# Generate a 2D random walk with normal (gaussian) displacements
generate_2d_walk_normal <- function(N) {
  x_steps <- rnorm(N)
  y_steps <- rnorm(N)
  x_positions <- cumsum(x_steps)
  y_positions <- cumsum(y_steps)
  return(data.frame(x = x_positions, y = y_positions))
}
```

Listing 25: Functions to generate random 2d walks

8.2.2 Output

```
1  experiments = 4
2  steps = 10000
3
4  par(mfrow = c(1, experiments))
5  for (i in 1:experiments) {
6    walk_data <- generate_2d_walk_regular(steps)
7    plot(walk_data$x, walk_data$y, type = 'l', col = i)
8  }
9  for (i in 1:experiments) {
10   walk_data <- generate_2d_walk_normal(100)
11   plot(walk_data$x, walk_data$y, type = 'l', col = i)
12 }
```

Listing 26: Ploting random walks

The following figures depict three rows of five different experiments each for both random 2D regular walks and random normal walks. Each row corresponds to a distinct value of $N(numberofsteps)$, as well as the last figures.

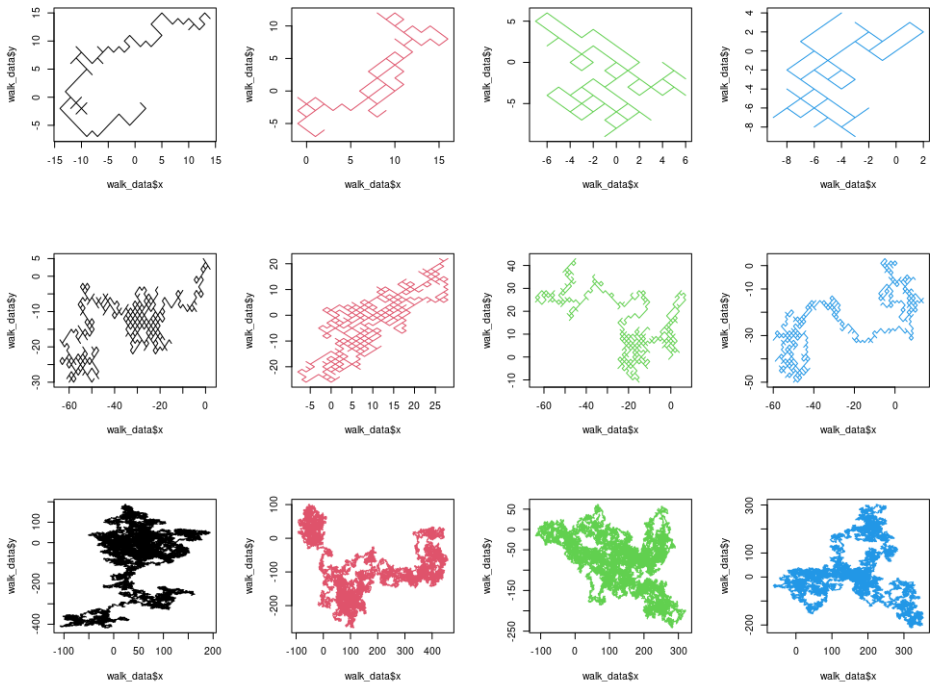

Figure 9: Five 2D random regular walks for 3 different N

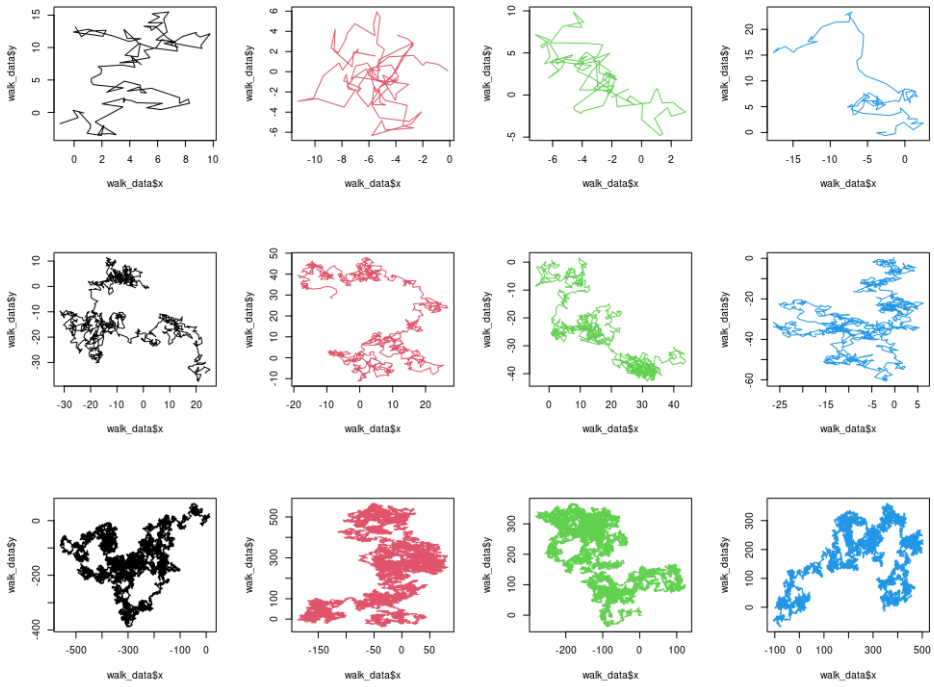

Figure 10: Five 2D random regular walks for 3 different N

8.3 Comments

After a large number (N) of steps, the positions in the regular random walk (generate 2d walk regular) become discrete and constrained to a lattice structure. The cumulative effect of taking steps of either -1 or 1 results in positions that are multiples of these discrete step sizes. The overall movement forms a grid pattern in the 2D space, and the range of possible positions is limited, reflecting the discrete nature of the step values. Additionally, the regular random walk displays a maximum possible distance from the starting point of N units, achieved when all steps align in the same direction.

On the other hand, in the normal random walk (generate 2d walk normal), the positions after a large number of steps exhibit continuous characteristics. The cumulative sum of normally distributed random variables leads to a smooth trajectory, with positions dispersed over a continuous range. Unlike the regular random walk, the normal random walk is not constrained to discrete step sizes or a limited range. As N increases, the positions can extend indefinitely, reflecting the nature of the normal distribution and the central limit theorem. The resulting trajectory appears more continuous and less constrained compared to the regular random walk.

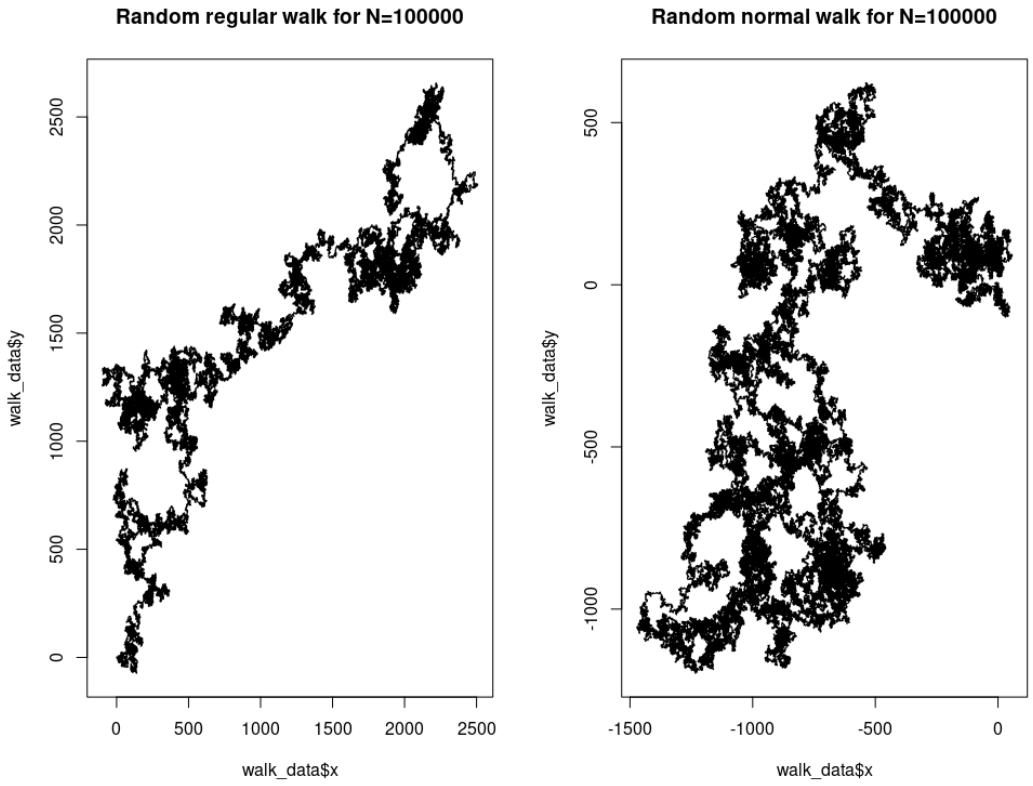

Figure 11: Random walks for big N

9 Obama vs McCain

9.1 Introduction

```
library(learningr)
library(ggplot2)
library(lattice)
library(corrplot)

election_data <- obama_vs_mccain
summary(election_data)
```

Listing 27: Data importation

As can be seen at description the data set is formed by state-by-state voting information in the 2008 US presidential election, along with contextual information on income, unemployment, ethnicity and religion. A data frame with 52 observations (one for each US state) and the following columns:

- **State** The name of the US state.
- **Region** The US Federal region.
- **Obama** Percentage of voters who voted for Barack Obama in the 2008 election.
- **McCain** Percentage of voters who voted for John McCain in the 2008 election.
- **Turnout** Percentage of people who voted in the 2008 presidential election.
- **Unemployment** Percentage of people who are unemployed.
- **Income** Mean annual income in US dollars.
- **Population** Number of people living in the state.
- **Catholic** Percentage of people identifying as Catholic.
- **Protestant** Percentage of people identifying as Protestant.
- **Other** Percentage of people identifying as religious (not Catholic nor Protestant).
- **Non.religious** Percentage of people identifying as non-religious.
- **Black** Percentage of people identifying as black.
- **Latino** Percentage of people identifying as Latino.
- **Urbanization** Percentage of people living in an urban area.

9.2 Study of the correlation

To study the correlation between the variables, I will use the Pearson correlation coefficient. The Pearson correlation coefficient, often denoted as r, is a statistical measure that quantifies the linear relationship between two continuous variables. It assesses the degree to which a change in one variable corresponds to a change in another variable. The coefficient ranges from -1 to 1:

- $r = 1$ Perfect positive correlation
- $r = -1$ Perfect negative correlation
- $r = 0$ No correlation

The formula for calculating the Pearson correlation coefficient between a pair of random variables, X and Y is:

$$\rho_{X,Y} = \frac{\mathbf{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\mathbb{E}[(X - \mu_x)(Y - \mu_Y)]}{\sigma_X \sigma_Y} \tag{1}$$

When we are workings with a sample, the Pearson correlation coefficient between two variables, X and Y, with n data points, is the following. Where X_i and Y_i are individual data points and $\bar{X}$ and $\bar{Y}$ are the means of variables X and Y.

$$r = \frac{\sum (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum (X_i - \bar{X})^2 \sum (Y_i - \bar{Y})^2}} \tag{2}$$

However, it assumes a linear relationship and is sensitive to outliers. When interpreting r, it is important to keep in mind that correlation does not imply causation, and other factors may influence the observed relationships.

We are going to create a correlation heatmap to visualize the Perason Correlations between all the different variables of the dataframe of Obama vs McCain. Nevertheless, we are going to discard these state and variables that have NA (Not Available) values in certain. In this case, Religious identification data are not available for **Alaska** and **Hawaii**. Also, variable **Turnout** has some NA, so we are going to drop it as well.

```
#Droping Alaska and Hawaii data
election_data <- election_data[!(election_data$State %in% c("Alaska",
    Hawaii")), ]
# Calculation of the Pearson Correlation Cofficients
corr_matrix <- cor(election_data[, c("Obama", "McCain", "Unemployment"
    , "Income", "Population", "Catholic", "Protestant", "Other", "Non.
    religious", "Black", "Latino", "Urbanization")])
  corrplot(corr_matrix, method = "color")
#Plot a heatmap with the cofficients
corrplot(corr_matrix, method = "color")
```

Listing 28: Calculation Pearson Cofficients and plotting they in a heat map

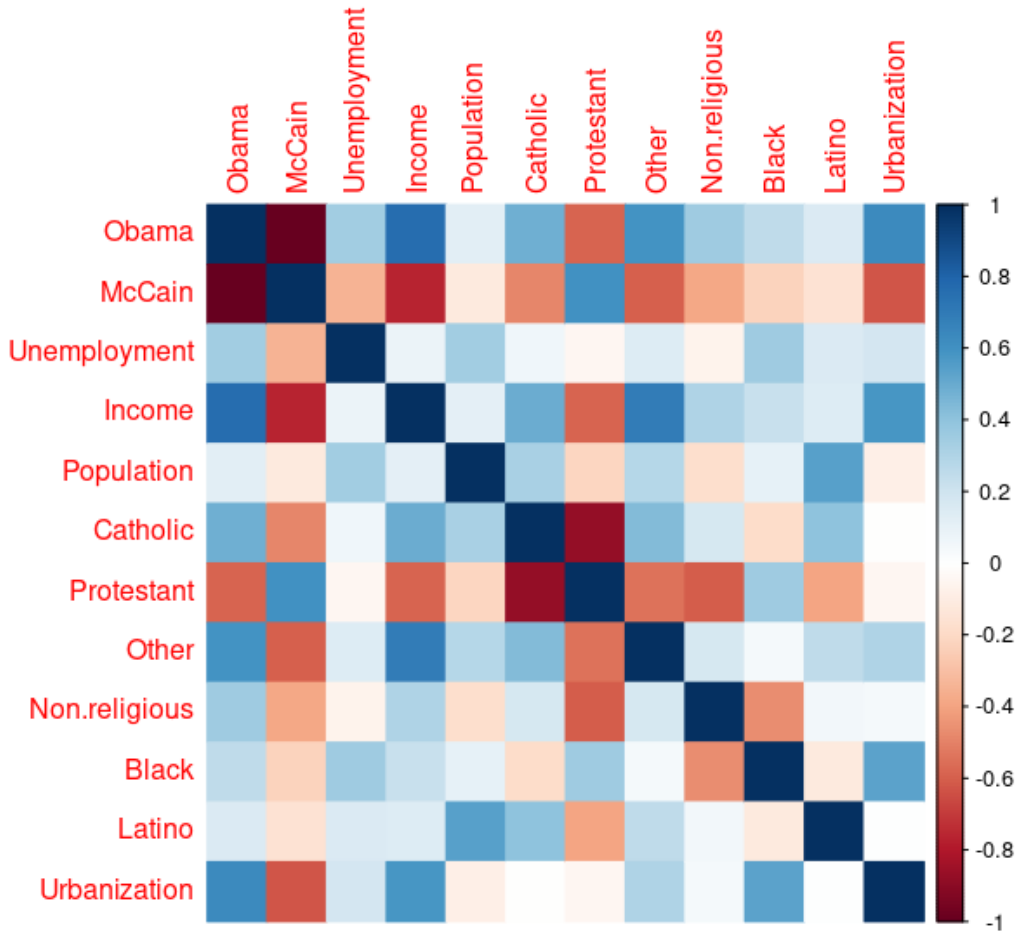

Figure 12: Heat map with Pearson Correlation Coefficients

The correlation matrix reveals several noteworthy relationships in the dataset. Firstly, the percentage of votes for Barack Obama and John McCain exhibits a highly negative correlation (-0.998), which is expected in a presidential election with two major candidates. Economically, Obama's votes correlate positively with the mean annual income (0.769) and somewhat with unemployment (0.349), suggesting that states where Obama received more votes tend to have higher income and lower unemployment rates. Regarding religious variables, correlations between Obama's votes and religious affiliations show interesting patterns, such as a positive correlation with the "Non.religious" category (0.353) and a negative correlation with "Protestant" (-0.589). Lastly, urbanization exhibits a moderate positive correlation with Obama's votes (0.631), suggesting that states with higher urban populations were more likely to favor Obama.

9.3 Religion distribution in the different states

9.3.1 Script

```
# Select relevant columns for the analysis
religion_data <- election_data[, c('State', 'Catholic', 'Protestant',
    'Other', 'Non.religious')]

religion_data_long <- tidyr::gather(religion_data, key = 'Religion',
    value = 'Percentage', -State)

ggplot(religion_data_long, aes(x = State, y = Percentage, fill =
    Religion)) +  geom_bar(stat = 'identity') +  labs(title = 'Religion
     Distribution in Different States (Obama vs. McCain)',  x = 'State'
    , y = 'Percentage') +  theme_minimal() +  theme(axis.text.x =
    element_text(angle = 45, hjust = 1))
```

Listing 29: Calculation Pearson Cofficients and plotting they in a heat map

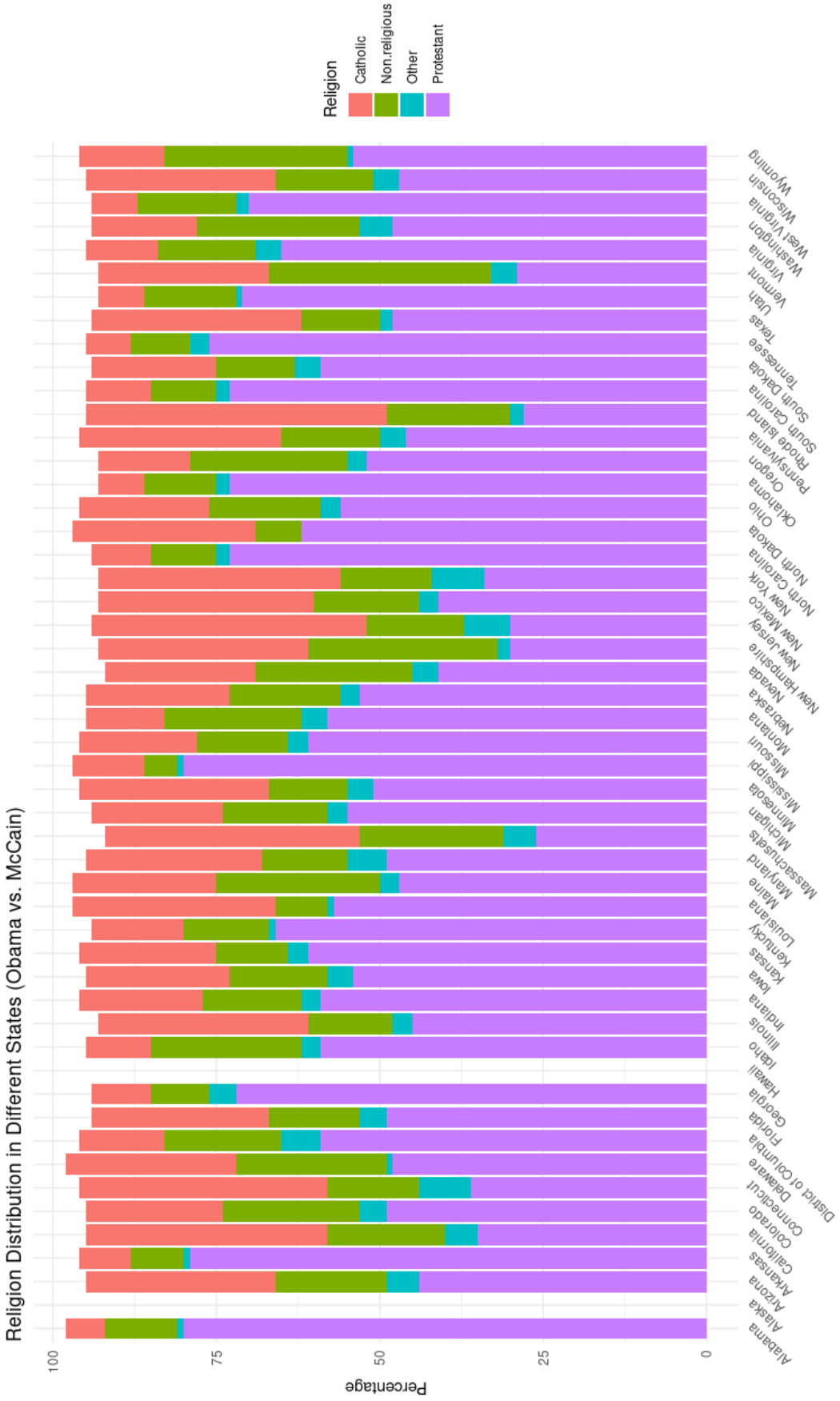

Figure 13: Bar chart of the religion distribution in the different states

10 Entropy

10.1 Script

This function aims to determine an optimal threshold 't' for partitioning a dataset with variables $X1$ and $X2$ to predict the target class Y. The target class depends deterministically on $X1$ and $X2$, with $Y = A$ if $X2 < X1^g$ and $Y = B$ otherwise, where 'g' is a positive constant. So, this script demonstrates how to use Information Gain and Gini Index criteria to find the optimal threshold 't' for partitioning a dataset based on variable X1 to minimize uncertainty or impurity in predicting the target class Y.

```
# Function to compute information gain
information_gain <- function(p1, p2) {
  - (p1 * log2(p1) + p2 * log2(p2))
}

# Function to compute Gini index
gini_index <- function(p1, p2) {
  1 - (p1^2 + p2^2)
}

# Generate sample data
set.seed(123)
n <- 1000
X1 <- runif(n)
X2 <- runif(n)
g <- 2
Y <- ifelse(X2 < X1^g, "A", "B")

# Initialize vectors to store results
t_values <- seq(0.01, 0.99, by = 0.01)
info_gain_values <- numeric(length(t_values))
gini_index_values <- numeric(length(t_values))

# Compute information gain and Gini index for different t values
for (i in seq_along(t_values)) {
  t <- t_values[i]
  # Split data based on t
  part1 <- Y[X1 <= t]
  part2 <- Y[X1 > t]

  # Compute class probabilities for each part
  p1_A <- sum(part1 == "A") / length(part1)
  p1_B <- sum(part1 == "B") / length(part1)
  p2_A <- sum(part2 == "A") / length(part2)
  p2_B <- sum(part2 == "B") / length(part2)

  # Compute information gain and Gini index
```

```
  info_gain_values[i] <- information_gain(p1_A, p1_B) * length(part1)
   / n +
    information_gain(p2_A, p2_B) * length(part2) / n

  gini_index_values[i] <- gini_index(p1_A, p1_B) * length(part1) / n +
    gini_index(p2_A, p2_B) * length(part2) / n
}

# Plot the information gain and Gini index functions
par(mfrow = c(2, 1))
plot(t_values, info_gain_values, type = "l", col = "blue", xlab = "t",
     ylab = "Information Gain",
      main = "Information Gain vs t")
abline(v = t_values[which.max(info_gain_values)], col = "red", lty =
   2)

plot(t_values, gini_index_values, type = "l", col = "green", xlab = "t
   ", ylab = "Gini Index",
      main = "Gini Index vs t")
abline(v = t_values[which.max(gini_index_values)], col = "red", lty =
   2)
```

Listing 30: Functions

10.2 Output

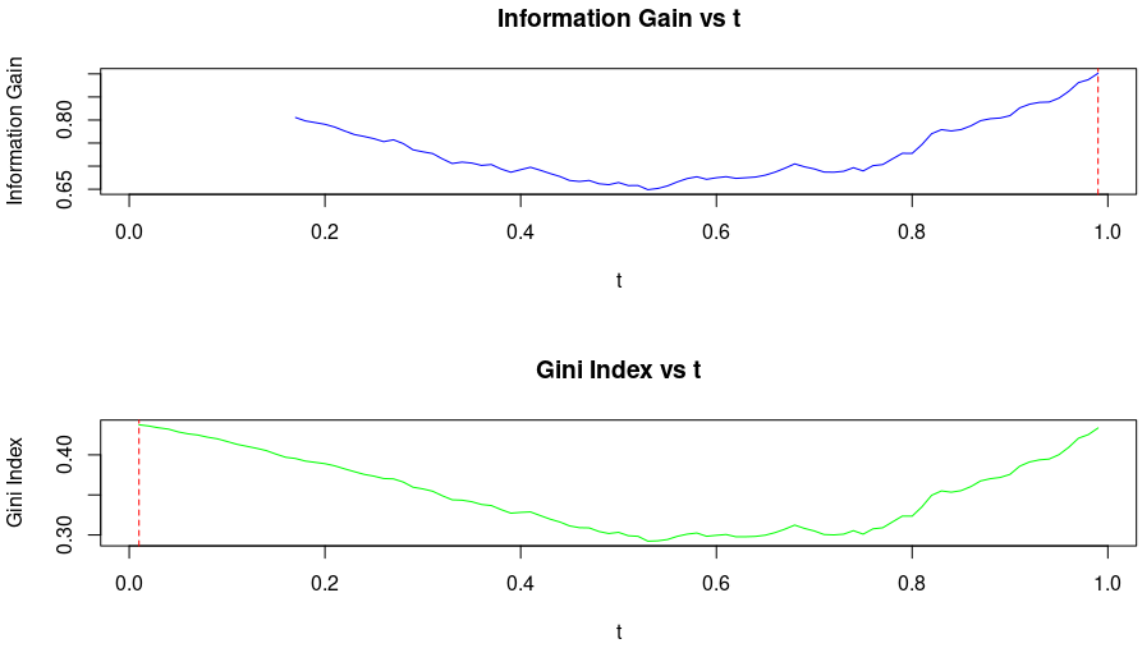

Figure 14: Information gain for diferent t values

11 Wine reloaded

11.1 Script

What I've done is training a Neural Network Model to predict wine pH based on various wine characteristics (such as fixed acidity, volatile acidity, citric acid, etc.).

```
library(neuralnet)

# Predicting whine pH

whine<-read.csv("/home/victhor/repositories/bigdata/exercices/data/
    whitewines.csv")

ls()
str(whine)
dim(whine)
n<-nrow(whine)
ncol(whine)
names<-names(whine)

par(mfrow=c(3,3))
for (k in 1:9) hist(whine[,k],col="yellow",xlab="",main=names[k])

# For normalizing numerical data
normalize<-function(x){
  return((x-min(x))/(max(x)-min(x)))
}

# Apply normalization to all numerical columns
whine_n<-as.data.frame(lapply(whine,normalize))

par(mfrow=c(3,3))
for (k in 1:9) hist(whine_n[,k],col="pink",xlab="",main=names[k])

n*0.75

nt<-773

whine_train<-whine_n[1:nt,]
whine_test<-whine_n[(nt+1):n,]

mean(whine_train$pH)
mean(whine_test$pH)

# Size 1 hidden layer
set.seed(34)
```

```
whine_model<-neuralnet(pH~fixed.acidity+volatile.acidity+citric.acid+
   residual.sugar+
                          chlorides+free.sulfur.dioxide+total.sulfur.
   dioxide+density,data=whine_train)

plot(whine_model)

whine_test_results<-compute(whine_model,whine_test[1:8])

strenght_pred<-as.vector(whine_test_results$net.result)

cor(whine_test$pH,strenght_pred)

RMSE<-sqrt(mean((whine_test$pH-strenght_pred)^2))
RMSE

# Size 5 hidden layer
whine_model2<-neuralnet(pH~fixed.acidity+volatile.acidity+citric.acid+
   residual.sugar+
                           chlorides+free.sulfur.dioxide+total.sulfur.
   dioxide+density,data=whine_train,hidden=5)

plot(whine_model2)

whine_test_results2<-compute(whine_model2,whine_test[1:8])

strenght_pred2<-as.vector(whine_test_results2$net.result)

cor(whine_test$pH,strenght_pred2)

RMSE<-sqrt(mean((whine_test$pH-strenght_pred2)^2))
RMSE
```

Listing 31: Script

11.2 Output

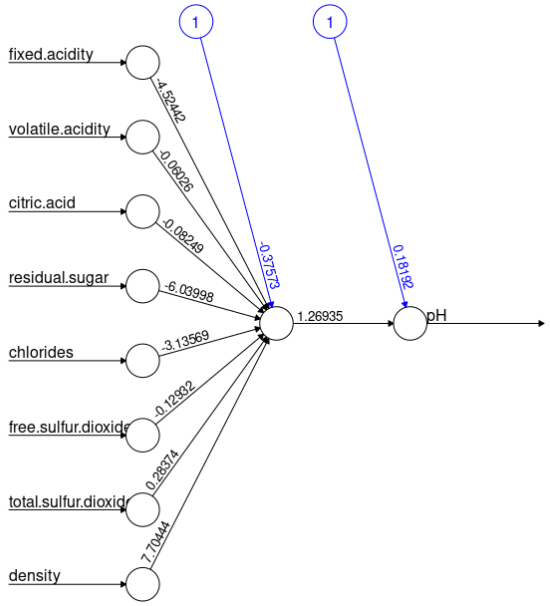

Figure 15: Neuronal network with 1 hidden neuron

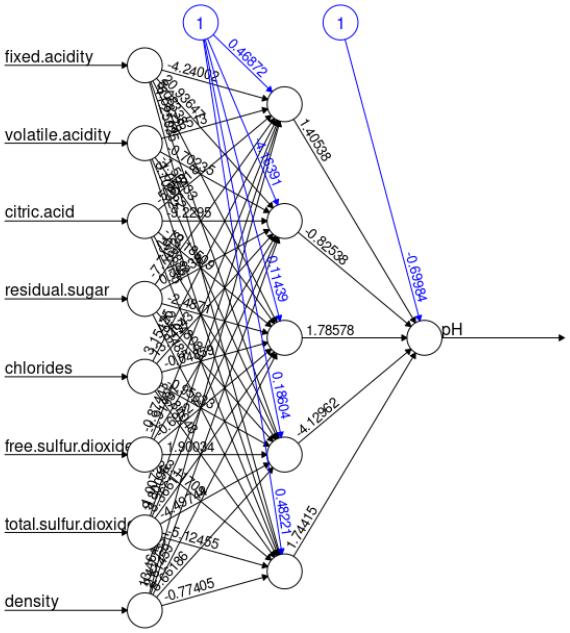

Figure 16: Neuronal network with 5 hidden neurons

11.3 Behavior comparison

Model	Mean Squared Error (MSE)
Regression Tree	0.5547409
Neural Network (1 hidden layer)	0.1128184
Neural Network (5 hidden layers)	0.1214287

Table 1: Mean Squared Error (MSE) for Different Models

In the comparison of predictive models, the mean squared error (MSE) provides a quantitative measure of their performance. Tree Regression yielded an MSE of 0.5547409, indicating a certain level of predictive error. On the other hand, Neural Networks with 1 hidden layer achieved a lower MSE of 0.1128184, suggesting improved predictive accuracy. Interestingly, increasing the complexity with 5 hidden layers in the Neural Network led to a slightly higher MSE of 0.1214287.