Roulette Monte Carlo - Tonnar Castellano

In the following blog post I am going to detail a high level overview of a Monte Carlo simulation for a roulette game specifically focusing on the Martingale strategy. I will also include various information about what happens as you change certain parameters in the simulation.

How does a computer simulation manage to simulate roulette? First we should establish quickly how roulette is played for those that might not know. There are 38 "pockets" with 18 red, 18 black, and 2 green where the ball has the potential to stop. Upon stopping there if you choose that pockets color or number you win an amount of money relative to the risk you took. For our purposes what we need to know is that if you choose black or red you simply get your bet plus its amount no bonus. To demonstrate a simple example if you bet 2 dollars on red you get your 2 dollars back plus winnings which would be 2 dollars thus making the amount back.

Now that we have established some basic ideas about roulette lets talk about the martingale strategy. The martingale strategy says that upon losing, you double your previous bet every time until you win. For example if you start with betting a 1 dollar then lose, you bet 2 dollars then 4 dollars then 8 dollars, etc. Notice eventually if you win you recoup and actually make a bit on top. $8 - 4 - 2 - 1 = 1$, So in this example you would've made a dollar. You can walk through this on your own if you don't believe this works

Ok so obviously the martingale strategy works… just keep betting until eventually you win. Lets pack up and go to Vegas!

Not so fast - there are a few things you aren't considering which are crucial to how the simulation plays out. These are what are going to be known as the parameters. 1) Betting limit: Most casinos only allow you to bet so much at certain tables. 2) Budget: How much you got. 3) Winnings: If you do manage to fleece Vegas when do you walk? 4) Number of plays: How long are you going to sit at the table?

Now that we established the parameters lets get into the guts of the simulation! As established above we need to be able to simulate whether or not R will choose a red, black, or green pocket. The good news is R has the ability to do this using whats called a function which simply means it does some magic under the hood. In our case that function will be rbinom(1,1,18/38). Rbinom() means simulate one time the odds of getting a 1 (a success) with a 18/38 chance! So now all we need to do is check if we win or lose!

If we win we keep our betting at the starting amount and increase our budget by the winnings! If we lose we need to subtract the lose from our budget then double our bet! To do this we have an if else function and a list these work exactly like you would image. A list is just that a list! Image a grocery list you change it as you walk through the store sometimes adding sometimes removing. The if else function simply says if you win, hey R make a note on the list and update what is needed for a win (remember budget and plays)! Else update what is needed for a loss. We then have a function that checks if you are at the win limit, over budget, or the max number of plays. Below for the super nerds is the simulation code!

```
#' A single play of the Martingale strategy
#'
#' Takes a state list, spins the roulette wheel, returns the state list with updated values (for exampl
#' @param state A list with the following entries:
#'   B              number, the budget
#'   W              number, the budget threshold for successfully stopping
#'   L              number, the maximum number of plays
#'   M              number, the casino wager limit
#'   plays          integer, the number of plays executed
#'   previous_wager number, the wager in the previous play (0 at first play)
```

```r
#'   previous_win   TRUE/FALSE, indicator if the previous play was a win (TRUE at first play)
#' @return The updated state list
#'
#'
library(dplyr)
```

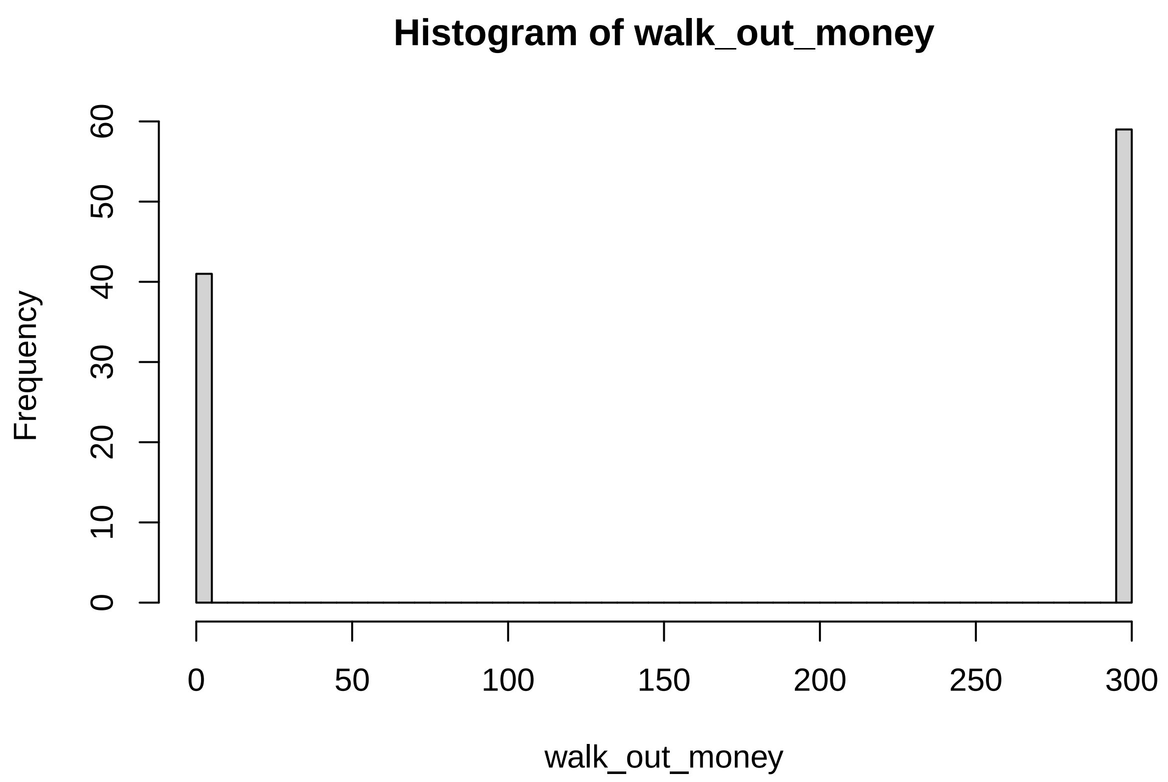
```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
one_play <- function(state){

  # Wager
  proposed_wager <- ifelse(state$previous_win, 1, 2*state$previous_wager)
  wager <- min(proposed_wager, state$M, state$B)

  # Spin of the wheel
  red <- rbinom(1,1,18/38)

  # Update state
  state$plays <- state$plays + 1
  state$previous_wager <- wager
  if(red){
    # WIN
    state$B <- state$B + wager
    state$previous_win <- TRUE
  }else{
    # LOSE
    state$B <- state$B - wager
    state$previous_win <- FALSE
  }
  state
}


#' Stopping rule
#'
#' Takes the state list and determines if the gambler has to stop
#' @param state A list.  See one_play
#' @return TRUE/FALSE
stop_play <- function(state){
  if(state$B <= 0) return(TRUE)
  if(state$plays >= state$L) return(TRUE)
  if(state$B >= state$W) return(TRUE)
  FALSE
}
```

```r
#' Play roulette to either bankruptcy, success, or play limits
#'
#' @param B number, the starting budget
#' @param W number, the budget threshold for successfully stoping
#' @param L number, the maximum number of plays
#' @param M number, the casino wager limit
#' @return A vector of budget values calculated after each play.
one_series <- function(
  B = 200
  , W = 300
  , L = 1000
  , M = 100
){

  # initial state
  state <- list(
    B = B
    , W = W
    , L = L
    , M = M
    , plays = 0
    , previous_wager = 0
    , previous_win = TRUE
  )

  # vector to store budget over series of plays
  budget <- rep(NA, L)
  counter = 0
  # For loop of plays
  for(i in 1:L){
    new_state <- state %>% one_play
    budget[i] <- new_state$B
    if(new_state %>% stop_play){
      return(budget[1:i])
      #If you want yo get the walkout money
    }
    state <- new_state
  }

}
# helper function
get_last <- function(x) x[length(x)]
```

So lets see some results! The first code chuck below is going to run the simulation and give us how much money we are walking out with at the end of each day at the table!

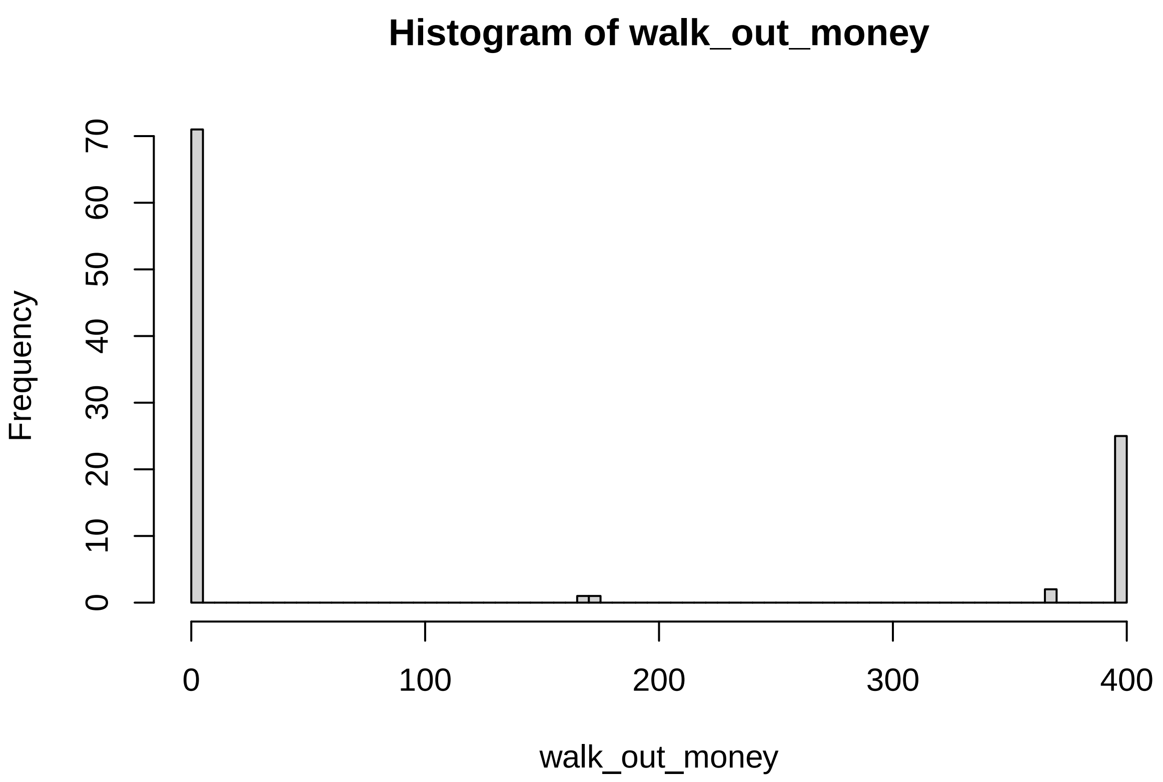
```r
walk_out_money <- rep(NA,100)
for(j in seq_along(walk_out_money)){
  walk_out_money[j] <- one_series(B = 200, W = 300, L = 1000, M = 100) %>% get_last}
hist(walk_out_money, breaks = 100)
```



Something that might strike you as interesting is that it seems like we walk out with 300 dollars a bit more than when we walk out with 0. The strategy works, lets go get rich! However, remember this is total earnings not profit. So we should probably go ahead and determine our expected value! This will be the amount we win * probability of winning + amount we lose * probability of losing, or with numbers $.55 * 100 + .45 * -200 = 55 + -90 = -40$ In the immortal words of Scooby Doo "ruh roo".

Maybe all is not lost! Being the clever person that you are, you say well that only because we cant win as much money as we can lose so Ill just change how much I can bet. Well, the good news is that I can simulate that for you as well! We will do 400 as to make it even 200 winnings and losing

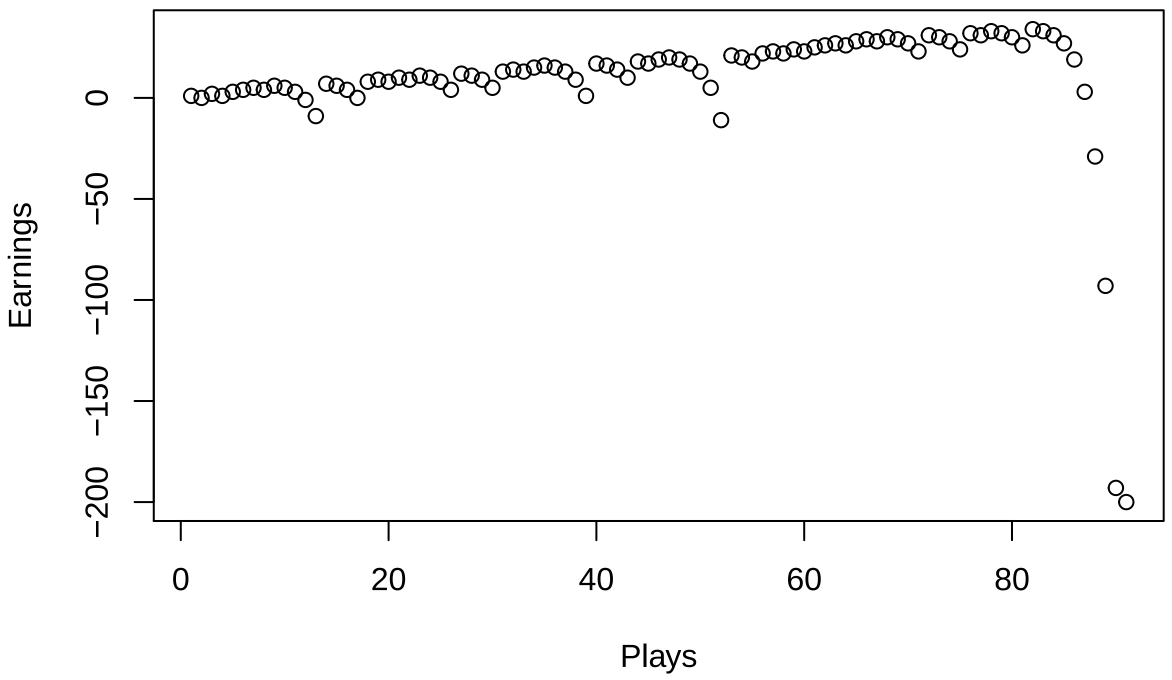
```r
walk_out_money <- rep(NA,100)
for(j in seq_along(walk_out_money)){
  walk_out_money[j] <- one_series(B = 200, W = 400, L = 1000, M = 100) %>% get_last}
hist(walk_out_money, breaks = 100)
```



Notice, suddenly you seem to lose a lot more money. From an intuitive perspective this should make sense! At the table you have a better chance to lose than you do to win ($18/38 or ~ 47%). However, you have to win "the same number of times" total as losing to make money but you have a better chance to lose, so you should lose more often!

If look below I have also included an average number of earnings over the course of a series of plays.

```
Earnings <- one_series(B = 200, W = 300, L = 1000, M = 100) - 200

plot(Earnings,xlab = 'Plays')
```



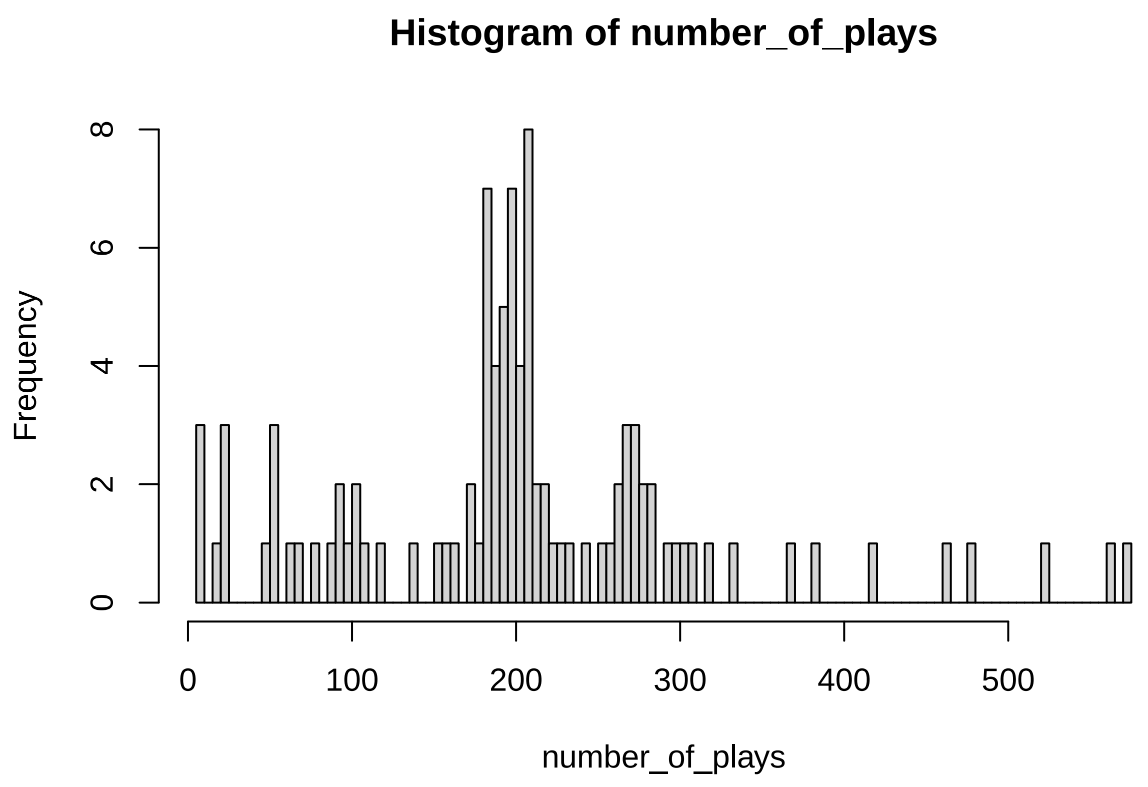
Also something worth noting would the average amount of money a gambler would earn sitting at the table. This will tell us how well this strategy actually works! *Hint: it doesnt…

```
number_of_plays <- rep(NA,100)
for(j in seq_along(number_of_plays)){
  number_of_plays[j] <- one_series(B = 200, W = 300, L = 1000, M = 100) %>% get_last}
print(mean(number_of_plays) - 200)
```

```
## [1] -59
```

Probably one of the most interesting things to notice is that the number of plays really doesn't tend to matter as long as its bigger than roughly 500 for a winnings limit of 300. The reason the average number of plays lower is that we tend to converge towards a win or lose much faster than the current limit!

```
number_of_plays <- rep(NA,100)
for(j in seq_along(number_of_plays)){
  number_of_plays[j] <- one_series(B = 200, W = 300, L = 1000, M = 100) %>% length()}
hist(number_of_plays, breaks = 100)
```



```
print(c('The average number of plays here is:',mean(number_of_plays)))
```

```
## [1] "The average number of plays here is:"
## [2] "205.21"
```

So, to wrap up this post the first take away should be you cant beat the house. Don't try. The second major takeaway is that most simulations tend to use a variety of functions to simply ideas in order to make abstractions on how a real life phenomenon works. Lastly, in this case some of the important parameters tend to be the budget and the win limit! The last couple of things to mention are some of the limitations of the Monte Carlo simulation and the martingale strategy in particular. The first is that it is only as accurate as the parameters. If you miss estimate the true probability your simulation will be totally screwed up. The next major limitation of the simulation is that in this case we are a bit more simplistic as normally you can actually bet on other colors/numbers which you could use to potentially change some of the outcomes.