Técnicas de Simulación

02. La Martingala

Miguel A. Castellanos

Contents

---

1 La martingala

La martingala es una estrategia muy vieja y conocida para apostar en la ruleta de los casinos. Todo el mundo sabe que no funciona, y se ha demostrado mil veces que se pierde dinero con ella (de hecho en los casinos te animan a que la utilices), pero en este caso de estudio vamos a utilizar simulaciones para estudiar por qué no es una buena táctica, y de qué dependería que se obtengan beneficios o no.

Vamos a empezar a explicar qué es una ruleta. La ruleta de los casinos es un juego creado en la edad media wikipedia, aunque en su formato actual es bastante distinto al original. Está compuesto por 36 números, la mitad de color rojo y la otra mitad de color negro, más el cero que no tiene color (o se le asigna el color verde). Hay muchas formas de apostar, tantas como se muestran en el paño de una ruleta americana (ver **Figure 1**); por ejemplo, apostando a un único número se recibe 35 veces el valor apostado, o apostando a una columna se gana 3 veces lo apostado. Aquí solo nos vamos a ocupar de la apuesta que interesa para la martingala en la que se apuesta a un color (rojo o negro) y si sale el color elegido la mesa te paga el doble de lo apostado. Es sencillo calcular que la probabilidad de ganar para este juego es de 18/37 (36 números más el 0), si apuestas 1000€ tu valor esperado es de 486.48€. Y aquí es donde entra la martingala.

Figure 1: Paño de una ruleta americana

La estrategia de la martingala es la siguiente:

1. Apuestas a Rojo o Negro x dinero (por ejemplo 1 €)
2. Si ganas, sigues apostando a Rojo o Negro x dinero (sigues apostando 1 €)
3. Si pierdes, la siguiente vez apuestas $x = 2 * x$ (apuestas 2 €)
4. Se repiten los pasos de 1 a 3 hasta ser millonario (o perder todo el dinero)

Con este procedimiento da la sensación de que a largo plazo la esperanza matemática te lleva a ganar dinero (o al menos a no perder), ya que se intercalan secuencias de pérdidas (que al final recuperas al doblar la apuesta) y otras secuencias de ganancias; sin embargo esto no es así, el valor esperado de la martingala es negativo. La cuestión clave es que en el juego siempre hay límites:

- El límite de dinero del jugador
- El límite de apuesta máxima para una mesa que establecen los casinos

Si quieres aprender más sobre casinos y la martingala hay muchísima información en internet, por ejemplo en la wikipedia, o en microsiervos, o simplemente buscando en google. Aquí con los conceptos explicados hasta ahora tenemos suficiente.

Obviamente, la martingala es un proceso estadístico conocido y definido analíticamente a través de las leyes de la probabilidad, por lo que para contestar a cualquier pregunta podríamos recurrir a la distribución binomial y dejarnos de simulaciones, pero aquí, como estamos tratando de aprender, lo vamos a tratar en forma de simulación.

1.1 Tarea 1

Modeliza el proceso de la martingala y empaquétalo en una función llamada *martingala* que reciba como argumentos de entrada:

- Cantidad de dinero que tiene el jugador (**bolsa**)
- Cantidad de dinero que se juega en cada apuesta (**apuesta**)
- Límite máximo de apuesta de la mesa (**limite**)
- Probabilidad de ganar en cada jugada (**prob**), esta probabilidad la incluimos porque esta estrategia se podría aplicar a otros juegos similares en los que la probabilidad sea distinta.

Y que la función te devuelva:

- Dinero que hay en la bolsa del jugador (**resultado**)

Estas son las condiciones por defecto de nuestro juego:

- bolsa = 100
- apuesta = 10
- limite = 500
- prob = 18/37

Por ejemplo, mi implementación de la función *martingala* con los parámetros anteriores (pon set.seed(1) para que te de lo mismo) da un valor resultante de 60. Como hemos partido de una bolsa de 100€ ese día habremos perdido en el casino 40€.

```
set.seed(1)
martingala(bolsa = 100, apuesta = 10, limite = 500, prob = 18/37)   # debe dar 60
```

```
## [1] 60
```

Y las siguientes llamadas a la función nos da los siguientes valores:

```
martingala(100, 10, 500, 18/37)
```

```
## [1] 70
```

```
martingala(100, 10, 500, 18/37)
```

```
## [1] 70
```

```
martingala(100, 10, 500, 18/37)
```

```
## [1] 280
```

```
martingala(100, 10, 500, 18/37)
```

```
## [1] 30
```

```
martingala(100, 10, 500, 18/37)
```

```
## [1] 140
```

Si tu implementación de *martingala* no te da el mismo valor revisa el código, es posible que hayas cometido algún error en las reglas del juego.

Para construir la función, en esta ocasión, vamos a hacer uso de bucles como *for*, *while* o *repeat*; más adelante en el curso intentaremos mejorar esta estrategia.

Construir esta función *martingala* es el equivalente a construir el modelo matemático representa un fenómeno real (recuerda el tema de introducción). Una vez diseñada, y si estamos seguros de que funciona correctamente, es el momento de experimentar con ella para aprender sobre el juego de la martingala y contestar a preguntas relevantes como por ejemplo: cuál es el valor esperado para este juego (con las condiciones por defecto que hemos puesto). Para ello voy a ejecutar 1000 veces (**repeticiones = 1000**, o 10000 o 100000 veces, recuerda la *ley débil de los grandes números*) y calcular la media de los resultados obtenidos.

Construir la función puede ser complejo (hay que plasmar la lógica en código) pero una vez que tenemos los resultados nos empezamos a sentir cómodos porque no hay más que usar la estadística que ya conocemos.

Si hemos ejecutado 1000 veces la función martingala habremos obtenido 1000 números (que sensatamente habremos ido almacenando en algún sitio, como un vector o una data frame), y ahora con esos 1000 números podemos utilizar simplemente las funciones mean(), y sd() para estimar el valor esperado con su intervalo de confianza.

Recuerda que un Intervalo de Confianza se puede calcular como:

$$IC = \overline{X} \pm 1.95\frac{\sigma}{\sqrt{n}}$$

Como puedes imaginarte el valor de la media obtenida depende de los valores iniciales de los argumentos, si cambiamos cada uno de ellos (por ejemplo se reduce la apuesta, o se aumenta el límite de la mesa) obtendremos valores medios distintos, con intervalos distintos. Y eso es lo que vamos a hacer ahora:

1.2 Tarea 2

Si mantenemos en las condiciones por defecto el resto de los elementos y vamos sistemáticamente cambiando el valor de la probabilidad podemos crear un gráfico que represente cómo cambia el valor esperado al cambiar la probabilidad de ganar el juego. Por ejemplo, en el *eje x* representamos los distintos valores de probabilidad y en el *eje y* representamos los valores medios obtenidos tras 1000 repeticiones del juego (con sus intervalos de confianza).

En eso consiste esta tarea, crea ese gráfico, utiliza las condiciones por defecto pero analiza los valores de probabilidad entre 0.2 y 0.6 con intervalos de 0.01. Como el objetivo es sacar algún tipo de conclusión analiza el gráfico y extrae las conclusiones pertinentes.

1.3 Tarea 3

Haz el mismo gráfico para el valor esperado, pero ahora manteniendo constantes todos los valores excepto la bolsa del sujeto, que variará entre 10€ y 200€ en intervalos de 5. Interprétalo.

1.4 Tarea 4

Mismo ejercicio pero para la el limite de la mesa, que variará entre 50€ y 500€ en intervalos de 10. Interprétalo.

1.5 Tarea 5

Ahora tienes perfectamente simulado el juego y tres gráficos que te indican cómo cambia el valor esperado en función de la probabilidad, la bolsa y el límite de la mesa. ¿A qué conclusión se podría llegar? Si quieres (muy opcional) juega con tu simulador y trata de encontrar una combinación de valores con la que ganar dinero.

2 Tiempo de ejecución

Cuando las simulaciones son exhaustivas suelen requerir una gran cantidad de tiempo de proceso (hablamos de días, semanas o meses); este tiempo, en principio, es dependiente de la cantidad de recursos que se requieren para resolver un problema y de los recursos disponibles. Sobre lo primero ya hablaremos más adelante, ya que habrá problemas que no sean computacionalmente resolubles, y por tanto no se pueden abordar desde las simulaciones o el calculo masivo. Respecto a los segundos, es de lo que vamos a hablar ahora, de cómo tener soluciones computacionales viables. Existen diversos factores que afectan a la velocidad de ejecución y cálculo, vamos a ver los principales.

El primer factor sin duda es el **hardware**. Es increíble lo que ha avanzado la potencia de los ordenadores, para hacerse una idea, en los 90 los supercomputadores más potentes (los preciosos cray que estaban refrigerados por agua, ver *Figure 2*) tenían menos capacidad de cálculo que un *iphone*. Hoy en día, la informática actual ofrece soluciones increíbles para realizar una enorme cantidad de cálculos en un tiempo mínimo, por ejemplo,

con estrategias de paralelización, uso de supercomputadores, procesamiento con GPUs, clusterizacion de procesos, etc. O incluso la computación cuántica.

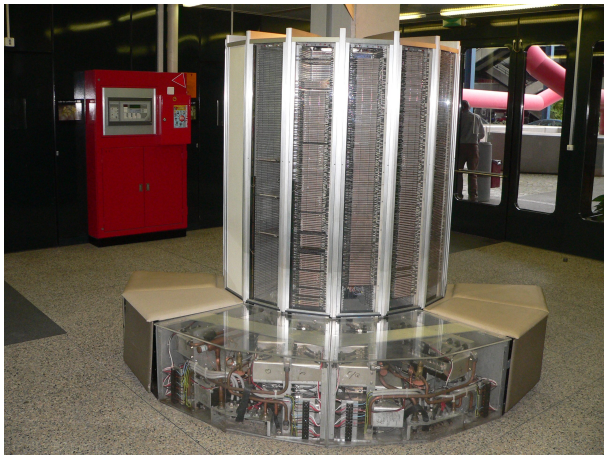

Figure 2: Cray-1

Todas estas cuestiones de hardware, aunque muy interesantes, pertenecen al mundo de la informática avanzada y exceden los contenidos del curso, entre otros motivos porque la mayoría de problemas de simulación que nos enfrentaremos aquí (y probablemente en nuestro trabajo de investigación) son resolubles con recursos limitados como los que disponemos con un pc.

Además de las cuestiones de hardware también el **software** elegido para construir el código es importante; en general se distingue entre lenguajes compilados (como C, fortran, C++, etc) y lenguajes interpretados (R, python, javascript, basic, perl, etc.) Enlace. Los primeros se dice que son de bajo nivel y los segundos de alto nivel. La diferencia principal entre ellos es que en los primeros se programan a un nivel muy bajo (casi de componentes físicos) y el código se traduce a lenguaje maquina para luego ser ejecutado (el proceso de compilación), mientras que los segundos no, en los segundos hay un intérprete que recoge el código y realiza las acciones indicadas, pero es el mismo intérprete para todos los programas. Obviamente, en el primer caso (los compilados) se consigue un código más eficiente y rápido que en los segundos (porque es específico para esa tarea), pero en los segundos es más "cómodo" y sencillo programar tareas complejas.

Dicho esto, en la actualidad, muchos procesos de cálculo masivo se llevan a cabo con código interpretado, en concreto con **python**, obteniéndose excelentes resultados. Esto es así porque se utilizan rutinas o *drivers*, escritos a muy bajo nivel y tremendamente eficientes, construidos para sacar el máximo provecho de hardware especializado. Estos lenguajes utilizan estos drivers para conseguir unos tiempos de ejecución increíbles, como es el caso de las librerías de TensorFlow para el control de las tarjetas GPU de Nvidia, con las que se lleva a cabo la mayoría de trabajos en *Deep Learning*.

El lenguaje que utilizamos en el curso, R, no es ni de lejos un lenguaje óptimo para simulaciones y computación masiva (aunque a través de librerías específicas como Keras se puede hacer uso de TensorFlow), pero a cambio tenemos toda la caja de herramientas estadísticas de R para utilizarla, y en eso, sí que R es imbatible frente a cualquier otro lenguaje. Además, como se ha dicho antes, nuestras simulaciones no son tan demandantes de recursos computacionales como para no poder resolverse con un simple PC y Rstudio.

Otro factor que afecta a la velocidad, y que es fácilmente controlable, es el **estilo del código** que escribimos. Como cada sentencia que escribamos va a ser ejecutada multitud de veces, cualquier elemento prescindible en el código debe ser eliminado para acortar tiempos. Es necesario escribir un código sencillo, limpio y legible, optimizado para ser ejecutado millones de veces.

Por ejemplo, llamar a una función siempre tiene un coste computacional, así que aunque estructuralmente sea más claro para nosotros usar funciones, hay que pensar mucho si es mejor no usarlas. O por ejemplo, algunas funciones (esto es especialmente relevante en R donde cada paquete es creado individualmente por un

autor) son más costosas que otras (requieren más recursos, tardan más), y están peor construidas por lo que es conveniente saber cuáles son preferibles frente a otras. Decimos peores funciones en el sentido de menos eficientes para la computación, pero tal vez, a cambio, pueden ser más completas o más cómodas de usar.

Y por último, la **vectorización**, esta consiste en huir, dentro de lo posible, de estrategias ineficientes como el uso de bucles, frente al álgebra lineal. En general, un código estructurado en torno a operaciones de vectores y matrices será muchísimo más rápido que uno estructurado en torno a bucles *for* o *while*. Como este, probablemente, sea el factor más determinante en las simulaciones (y el más complejo y difícil), no lo vamos a tratar aquí, sino que lo dejaremos para más adelante.

3 Medida del tiempo de proceso

Como acabamos de decir, escribir un código eficiente y rápido es imprescindible para llevar a cabo simulaciones de calidad. Esta tarea implica que es necesario ser capaces de evaluar cómo de eficiente es el código que hemos escrito y la mejor manera de hacerlo es medir cuánto tarda en realizar su función. Lo que vamos a ver a aquí son las opciones más sencillas para medir el tiempo de ejecución de nuestro código, para poder determinar si es adecuado o hay que optimizarlo. Como siempre en R hay muchas opciones y librerías especializadas en ello, aquí vemos las más básicas:

3.1 Sys.time

La más básica y sencilla, pero nos vale para la mayoría de las situaciones, cuando la llamamos simplemente nos da el tiempo en ese momento, haciendo la resta entre el tiempo antes y después del proceso sabremos cuanto ha tardado en ejecutarse. Un ejemplo de su uso:

```
inicio <- Sys.time()
Sys.sleep(2)           # duerme 2 segundos
fin <- Sys.time()

fin - inicio
```

```
## Time difference of 2.006492 secs
```

En tu ordenador saldrá un tiempo más cercano a 2.0000 de lo que aparece en este documento, es porque al compilar el texto con markdown se requiere de tiempo extra para llamar a las funciones.

3.2 system.time

La función system.time, es similar pero más precisa y da más información (distingue entre el tiempo del usuario, del sistema y el propio de la ejecución), pero a cambio el código tiene que estar encapsulado dentro de ella:

```
system.time({
  Sys.sleep(2)           # duerme 2 segundos
  })
```

```
##    user  system elapsed
##   0.000   0.000   2.003
```

3.3 proc.time

Una mezcla de las dos anteriores, no es necesario encapsular el código como en Sys.time y te da la información separada para user, system y elapsed

```
inicio <- proc.time()
Sys.sleep(2)           # duerme 2 segundos
fin <- proc.time()
```

```
fin - inicio
```

```
##    user  system elapsed
##   0.003   0.000   2.005
```

3.4 Otras opciones

Además, hay varias librerías que tienen funciones (la mayoría son wraps de las dos primeras) que pueden ayudar, una muy simple es tictoc.

```
# No ejecutar salvo que quieras instalar la libreria
# if (!require(tictoc)) {
#   devtools::install_github("collectivemedia/tictoc")
#   require(tictoc)
# }
#
# tic("Tiempo dormido:")
# Sys.sleep(2)
# toc()
```

Otros paquetes algo más complejos son: rbenchmark y microbenchmark.

4 Optimizando el código en R

Ahora que sabemos cómo medir la eficiencia de nuestro código (menor tiempo mayor eficiencia) podemos evaluar qué estrategias de programación son mejores para nuestros propósitos. Existen infinidad de páginas con recomendaciones, la mayoría de ellas con consejos de sentido común pero también con algunas afirmaciones sorprendentes, vamos a ver alguna de ellas:

4.1 Piénsate usar funciones

Las funciones son una maravilla que nos permiten simplificar y hacer más legible cualquier código, pero no son gratis, llamar a una función y pasarle sus argumentos requiere de un tiempo extra que, en caso de tener que repetirse millones de veces, puede ser contraproducente. Vamos a verlo con un ejemplo:

```
# definimos una funcion cualquiera, en este caso un suma
foo <- function(a, b, c){
  d = a + b + c
  return(d)
}

k=1:100000

# ejecutamos usando la funcion
system.time({
  for(i in k) foo(1,2,3)
})
```

```
##    user  system elapsed
##   0.046   0.000   0.045
```

```
# ejecutamos sin la función
system.time({
  for(i in k) d=1+2+3
})
```

```
##    user  system elapsed
##   0.001   0.004   0.006
```

Tus tiempos serán distintos (el importante es el elapsed) porque dependerá de la capacidad de tu ordenador, pero deberías haber obtenido un tiempo significativamente distinto para ambos casos.

En C y lenguajes compilados este problema se resuelve con el uso de MACROS pero por desgracia en R no es posible. Aunque existen diversas opciones para trabajar con macros en R su uso no es equivalente.

4.2 Tarea 6

Otra recomendación es preasignar la memoria de las estructuras de almacenamiento antes de usarlas en vez de ir asignando la memoria según se vaya necesitando. Por ejemplo, si preveo que voy a necesitar un vector de 10 elementos, es mucho más rápido crear un vector de 10 elementos desde el principio (y luego uno a uno asignarles el valor), que crear un vector vacío (sin elementos) y luego ir "añadiendo" uno a uno hasta llegar al último. En el primer caso R reserva un espacio para 10 elementos y en el segundo, cada vez que añades uno, R tiene que asignar un espacio para 1 elemento, es decir, de la segunda manera hago 10 veces el proceso en vez de una única vez. Si el vector es de un millón de elementos, imagínate el tiempo desperdiciado.

Muestra como es más rápido preasignar el tamaño de los elementos.

4.3 Tarea 7

Es mucho más rápido utilizar comparaciones vectoriales implícitas (pe: a[a<10]) que usar la sentencia if... else... Muéstralo.

4.4 Tarea 8

Analiza y decide qué bucle es más eficiente: for, while o repeat

4.5 Tarea 9

Analiza y decide qué es más eficiente: un bucle for o una función apply

4.6 Tarea 10

Analiza y decide qué es más eficiente, usar la función mean() u obtener la media con la función *describe()* del paquete *psych*.

Existirían infinidad de recomendaciones más, y poco a poco irán apareciendo. Tal vez, lo más avanzado si queremos usar R para cálculos masivos sea usar paquetes que nos permiten construir funciones de bajo nivel (como *compiler* o *Rcpp*) o paralelizar procesos (*foreach*, *parallel* o *doParallel*), pero estas cuestiones exceden el nivel del curso. El alumno, si está interesado, puede investigar por su cuenta.