Capítulo 11

Herramientas de programación adicionales

Como han mostrado los últimos capítulos, R ofrece a los usuarios flexibilidad y oportunidades para análisis de datos avanzados que no se encuentran en muchos programas. En este capítulo final, exploraremosRherramientas de programación, que permiten al usuario crear código que aborde cualquier problema único que enfrente.

Tenga en cuenta que muchas de las herramientas relevantes para la programación ya se han presentado anteriormente en este libro. En el Cap.10, las herramientas para el álgebra matricial en R se introdujeron, y muchos programas requieren procesamiento matricial. Además, las declaraciones lógicas (o booleanas) son esenciales para la programación.RLos operadores lógicos de se introdujeron en el Cap. 2, así que vea la tabla 2.1 para recordar cuál es la función de cada operador.

Las secciones siguientes presentarán varias otras herramientas que son importantes para la programación: distribuciones de probabilidad, definición de nuevas funciones, bucles, ramificaciones y optimización (que es particularmente útil para la estimación de máxima verosimilitud). El capítulo terminará con dos grandes ejemplos aplicados. El primero, basado en Monogan (2013b), presenta *programación orientada a objetos* en R y aplica varias herramientas de programación de este capítulo para encontrar soluciones a un problema de teoría de juegos insoluble. El segundo, dibujo de Signorino (1999), ofrece un ejemplo de *Análisis de Monte Carlo* y una estimación de máxima verosimilitud más avanzada en R.[1]

Juntas, las dos aplicaciones deben mostrar cómo todas las herramientas de programación pueden unirse para resolver un problema complejo.

Electrónico suplementario material: La en línea versión de esto capítulo (doi: 10.1007 / 978-3-319-23446-5_11) contiene material complementario destinado a disponible usuarios autorizados.

[1] Véase también: Signorino (2002) y Signorino y Yilmaz (2003).

JE Monogan III, *Análisis político con R,* Utilice R !,
DOI 10.1007 / 978-3-319-23446-5_11

Cuadro 11.1 Usando distribuciones de probabilidad en R

Prefijo	Sufijo de uso		Distribución
pag	Norma de función de distribución acumulada		Normal
D	Función de densidad de probabilidad	logis	Logístico
q	Función cuantil	t	*t*
r	Sorteo aleatorio de la distribución	f	*F*
		unif	Uniforme
		pois	Poisson
		Exp	Exponencial
		chisq	Chi-cuadrado
		binom	Binomio

11.1 Distribuciones de probabilidad

R le permite utilizar una amplia variedad de distribuciones para cuatro propósitos. Para cada distribución,R le permite llamar a la función de distribución acumulativa (CDF), la función de densidad de probabilidad (PDF), la función de cuantiles y las extracciones aleatorias de la distribución. Todos los comandos de distribución de probabilidad constan de un prefijo y un sufijo. Mesa11,1 presenta los cuatro prefijos y su uso, así como los sufijos de algunas distribuciones de probabilidad de uso común. Las funciones de cada distribución toman argumentos únicos para los parámetros de esa distribución de probabilidad. Para ver cómo se especifican, utilice archivos de ayuda (por ejemplo,?punif,? pexp, o ?pnorm).[2]

Si desea saber la probabilidad de que una observación normal estándar sea inferior a 1,645, utilice la *función de distribución acumulativa (*Comando CDF) pnorm:

```
normal (1.645)
```

Suponga que desea dibujar un escalar de la distribución normal estándar: para dibujar *a* *N*.0; 1 /,utilizar el *dibujo aleatorio* mando rnorm:

```
a <- normal (1)
```

Para dibujar un vector con diez valores de un $_2$ distribución con cuatro grados de libertad, use el comando de dibujo aleatorio:

```
c <- rchisq (10, gl = 4)
```

Recuerde del Cap. 10 que el muestra El comando también nos permite simular valores, siempre que proporcionemos un vector de valores posibles. Por eso,R ofrece una amplia gama de comandos de simulación de datos.

[2]Se pueden cargar otras distribuciones a través de varios paquetes. Por ejemplo, otra distribución útil es la normal multivariante. Al cargar elMASA biblioteca, un usuario puede tomar muestras de la distribución normal multivariante con la mvrnorm mando. Incluso de manera más general, lamvtnorm
El paquete permite al usuario calcular multivariante normal y multivariante *t* probabilidades, cuantiles, desviaciones aleatorias y densidades.

Suponga que tenemos una probabilidad dada, 0.9, y queremos saber el valor de un ^{2} distribución con cuatro grados de libertad en la que la probabilidad de ser menor o igual a ese valor es 0,9. Esto requiere el*función cuantil:*

```
qchisq (.9, gl = 4)
```

Podemos calcular la probabilidad de un cierto valor a partir del *función de probabilidad (* PMF) para una distribución discreta. De una distribución de Poisson con parámetro de intensidad 9, ¿cuál es la probabilidad de un conteo de 5?

```
dpois (5, lambda = 9)
```

Aunque suele ser de menor interés, para una distribución continua, podemos calcular el valor de la *función de densidad de probabilidad (*PDF) de un valor particular. Esto no tiene un significado inherente, pero ocasionalmente es necesario. Para una distribución normal con media 4 y desviación estándar 2, la densidad en el valor 1 viene dada por:

```
dnorm (1, media = 4, sd = 2)
```

11.2 Funciones

R le permite crear sus propias funciones con el función mando. La función El comando utiliza la siguiente sintaxis básica:

function.name <- function (ENTRADAS) {BODY}

Note que el función El comando primero espera que las entradas se enumeren entre paréntesis, mientras que el cuerpo de la función se enumera entre llaves. Como se puede ver, hay pocas restricciones en lo que el usuario elige poner en la función, por lo que una función se puede diseñar para lograr lo que el usuario desee.

Por ejemplo, supongamos que estamos interesados en la ecuación y D 2 C ^{1} $\overline{x_2}$. Pudimos definir esta función fácilmente en R. Todo lo que tenemos que hacer es especificar que nuestra variable de entrada es X y nuestro cuerpo es el lado derecho de la ecuación. Esto creará una función que devuelve valores para$y$. Definimos nuestra función, llamada first.fun, como sigue:

```
first.fun <-función (x) {
        y <-2 + x ^ {- 2}
        retorno (y)
        }
```

Aunque está dividido en unas pocas líneas, todo esto es un gran comando, ya que las llaves ({}) abarcan varias líneas. Con elfunción comando, comenzamos declarando que X es el nombre de nuestra única entrada, basado en el hecho de que nuestra ecuación de interés tiene X como el nombre de la variable de entrada. A continuación, dentro de las llaves, asignamos la saliday como siendo la función exacta de X que nos interesan. Como último paso antes de cerrar las llaves, utilizamos la regreso comando, que le dice a la función cuál es el resultado de salida después de que se llama. Laregreso El comando es útil para determinar la salida de la función por un par de razones: Primero,

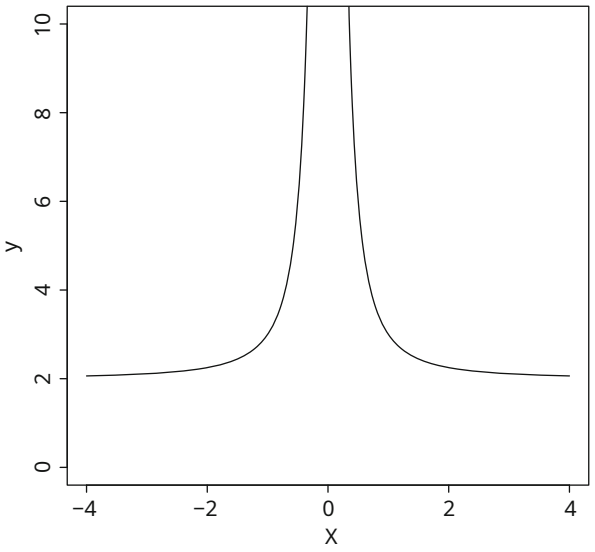

Figura 11.1 Gráfico de la función y D 2 C $\frac{1}{x^2}$

si definimos varios objetos dentro de un comando, esto nos permite especificar lo que debería imprimirse en la salida versus lo que era estrictamente interno a la función. Segundo, regreso nos permite informar una lista de elementos como salida, si lo deseamos. Alternativamente, podríamos haber sustituido el comandoinvisible, que funciona de la misma manera que regreso, excepto que no imprime la salida en la pantalla, sino que simplemente almacena la salida.

Con esta función, si queremos saber qué valor y toma cuando X D 1, solo necesitamos escribir: first.fun (1). Desde que usamos regreso en vez de invisible, la impresión simplemente dice:

[1] 3

Por supuesto, el resultado de que y D 3 aquí se puede verificar fácilmente a mano. Del mismo modo, sabemos que si$X$ D 3, luego y D 2⅓. Para verificar este hecho, podemos escribir: first.fun (3).
R nos dará la impresión correspondiente:

[1] 2.111111

Como tarea final con esta función, podemos trazar cómo se ve insertando un vector de X valores en él. Considere el código:

```
my.x <-seq (-4,4, por = .01)
plot (y = first.fun (my.x), x = my.x, type = "l",
      xlab = "x", ylab = "y", ylim = c (0,10))
```

Esto producirá el gráfico que se muestra en la Fig. 11,1.

Como ejemplo más complicado, considere una función que usaremos como parte de un ejercicio más amplio en la Secta. 11,6. La siguiente es una función de utilidad esperada para

un partido político en un modelo de cómo los partidos elegirán una posición cuando compitiendo en elecciones secuenciales (Monogan 2013b, Eq. (6)):

$$UE_{A.\ A;\ D/}\text{D f.}metro_1 \quad {}_{A/2}\text{C .}metro_1 \quad {}_{D/2}\text{C }V\text{gramo} \qquad (11{,}1)$$
$$\text{CI f.}metro_2 \quad {}_{A/2}\text{C .}metro_2 \quad {}_{D2}\text{gramo}$$

En esta ecuación, la utilidad esperada para un partido político (partido *A)* son la suma de las dos funciones de distribución logística acumulativa (las funciones), con la segundo ponderado por un término de descuento (0 I 1). La utilidad depende de las posiciones tomado por fiesta *A* y fiesta *D (A* y *D),* una ventaja de valencia para la fiesta *A (V),* y la posición de emisión del votante medio en la primera y segunda elección (*metro*1 y *metro*2). Esto ahora es una función de varias variables, y requiere que usemos el CDF del distribución logística. Para ingresar esta función en R, escribimos:

```
Función cuadrática.A <(m.1, m.2, p, delta, theta.A, theta.D) {
    util.a <-plogis (- (m.1-theta.A) ^ 2 +
          (m.1-theta.D) ^ 2 + p) +
        delta * plogis (- (m.2-theta.A) ^ 2 +
          (m.2-theta.D) ^ 2)
    volver (util.a)
    }
```

La plogis comando calcula cada relevante **pag**robabilidad del **logis**tic CDF, y todos los demás términos se nombran evidentemente a partir de la Ec. (11,1) (excepto eso pag se refiere al término de valencia *V).* Aunque esta función era más complicada, todo lo que teníamos que hacer era asegurarnos de nombrar cada entrada y copiar Eq. (11,1) completamente en el cuerpo de la función.

Esta función más compleja, Cuadrático.A, todavía se comporta como nuestra función simple. Por ejemplo, podríamos seguir adelante y proporcionar un valor numérico para cada argumento de la función como este:

```
Cuadrático.A (m.1 = .7, m.2 = -. 1, p = .1, delta = 0, theta.A = .7, theta.D = .7)
```

Al hacerlo, se imprime una salida de:

```
[1] 0.5249792
```

Por lo tanto, ahora sabemos que 0.52 es la utilidad esperada para la fiesta. *A* cuando los términos adquieren estos valores numéricos. De forma aislada, este resultado no significa mucho. En teoría de juegos, normalmente estamos interesados en cómo el partido*A (*o cualquier jugador) puede maximizar su utilidad. Por lo tanto, si tomamos todos los parámetros como fijos en los valores anteriores, excepto permitimos fiesta *A* elegir *A* como se ve adecuado, podemos visualizar lo que *A*La mejor opción es. El siguiente código produce el gráfico que se ve en la Fig.11,2:

```
posiciones <-seq (-1,1, .01) util.A <-Quadratic.A (m.1 = .7, m.2 = -. 1, p
= .1, delta = 0,
        theta.A = posiciones, theta.D = .7) plot (x =
posiciones, y = util.A, type = "l",
        xlab = "Posición del Partido A", ylab = "Utilidad")
```

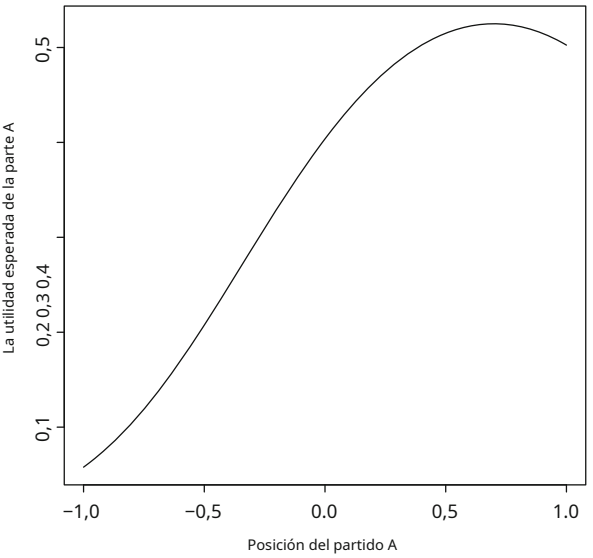

Figura 11.2 Gráfico de la utilidad esperada de un partido aventajado durante dos elecciones dependiendo de la posición del asunto

En la primera línea, generamos un rango de posiciones partido *A* puede elegir por *UNA.* En la segunda línea, calculamos un vector de utilidades esperadas para la fiesta *A* en cada uno de los problemas posiciones que consideramos y estableciendo los demás parámetros en su valor mencionado anteriormente. Por último, usamos elgráfico función para dibujar un gráfico lineal de estas utilidades. Resulta que el máximo de esta función está en *A* D 0: 7, por lo que nuestra utilidad de 0.52 mencionada anteriormente es la mejor fiesta *A* puede hacer después de todo.

11.3 Bucles

Los bucles son fáciles de escribir R y se puede utilizar para repetir cálculos que sean idénticos o que varíen solo en unos pocos parámetros. La estructura básica de un bucle con el por comando es:

para (i en 1: M) {COMANDOS}

En este caso, METRO es el número de veces que se ejecutarán los comandos. R también admite bucles con el tiempo comando que sigue una estructura de comando similar:

```
j <- 1
mientras que (j <M) {
        COMANDOS
        j <- j + 1
        }
```

Si una por bucle o un tiempo El bucle funciona mejor puede variar según la situación, por lo que el usuario debe utilizar su propio criterio al elegir una configuración. En general,tiempo Los bucles tienden a ser mejores para problemas en los que tendría que hacer algo hasta que se cumpla un criterio (como un criterio de convergencia), y por los bucles son generalmente mejores para las cosas que desea repetir un número fijo de veces. Bajo cualquier estructura,
R permitirá incluir una amplia gama de comandos en un bucle; la tarea del usuario es cómo administrar la entrada y salida del bucle de manera eficiente.

Como demostración simple de cómo funcionan los bucles, considere un ejemplo que ilustra la ley de los grandes números. Para hacer esto, podemos simular**r**observaciones de andom del estándar **norma**distribución al (fácil con la rnorm) mando. Dado que la normal estándar tiene una media poblacional de cero, esperaríamos que la media muestral de nuestros valores simulados sea cercana a cero. A medida que el tamaño de nuestra muestra aumenta, la media muestral debería estar más cerca de la media poblacional de cero. Un bucle es perfecto para este ejercicio: queremos repetir los cálculos de la simulación a partir de la distribución normal estándar y luego tomar la media de las simulaciones. Lo que difiere de una iteración a otra es que queremos que el tamaño de nuestra muestra aumente.

Podemos configurar esto fácilmente con el siguiente código:

```
set.seed (271828183)
almacenar <- matriz (NA, 1000,1)
para (i en 1: 1000) {
        a <- norma (i)
        almacenar [i] <- media (a)
        }
plot (store, type = "h", ylab = "Sample Mean",
        xlab = "Número de observaciones") abline
(h = 0, col = 'red', lwd = 2)
```

En la primera línea de este código, llamamos al comando set.seed, para que nuestro experimento de simulación sea replicable. CuándoR dibuja números aleatoriamente de cualquier manera, utiliza un generador de números pseudoaleatorios, que es una lista de 2.1 mil millones de números que se asemejan a sorteos aleatorios.[3] Al elegir cualquier número del 1 al 2,147,483,647, otros deberían poder reproducir nuestros resultados usando los mismos números en su simulación. Nuestra elección de 271,828,183 fue en gran medida arbitraria. En la segunda línea de código, creamos un vector en blanco llamadoTienda de longitud 1000. Este vector es donde almacenaremos nuestra salida del ciclo. En las siguientes cuatro líneas de código, definimos nuestro ciclo. El ciclo va de 1 a 1000, y el índice de cada iteración se denomina
I. En cada iteración, nuestro tamaño de muestra es simplemente el valor de I, por lo que la primera iteración simula una observación y la milésima iteración simula 1000 observaciones. Por tanto, el tamaño de la muestra aumenta con cada pasada del bucle. En cada pasada del programa,R muestras de un *N*.0; 1 /distribución y luego toma la media de esa muestra. Cada media se registra en elIla celda de Tienda. Después de que se cierra el ciclo, graficamos nuestras medias muestrales contra el tamaño de la muestra y usamos abline dibujar una línea roja

[3]Formalmente, la composición de esta lista se basa en una distribución uniforme estándar, que luego se convierte a la distribución que queramos utilizando una función de cuantiles.

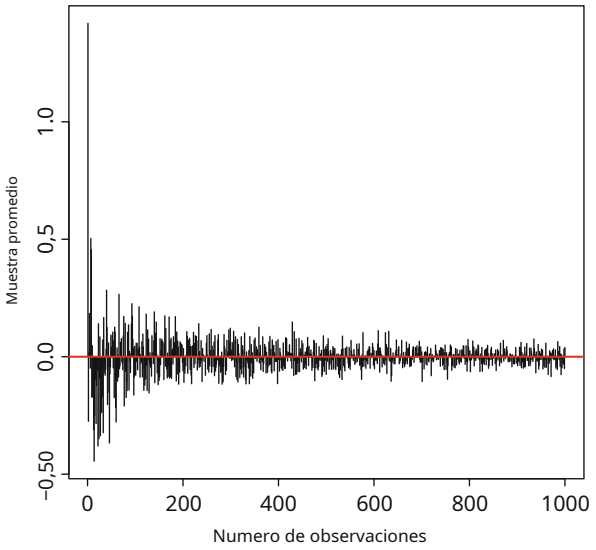

Figura 11.3 La ley de los grandes números y los bucles en acción.

en la media poblacional de cero. El resultado se muestra en la figura.11,3. De hecho, nuestro gráfico muestra la media de la muestra que converge a la media real de cero a medida que aumenta el tamaño de la muestra.

Los bucles son necesarios para muchos tipos de programas, por ejemplo, si desea hacer un análisis de Monte Carlo. En algunos casos (pero no en todos), sin embargo, los bucles pueden ser más lentos que las versiones vectorizadas de los comandos. Puede valer la pena probar elsolicitar comando, por ejemplo, si puede lograr el mismo objetivo que un bucle. Cuál es más rápido a menudo depende de la complejidad de la función y la sobrecarga de memoria para calcular y almacenar resultados. Entonces, si un enfoque es demasiado lento o demasiado complicado para que su computadora lo maneje, puede valer la pena probar el otro.

11.4 Ramificación

R los usuarios tienen la opción de ejecutar comandos condicionados a una declaración booleana usando el Si mando. Esto puede ser útil siempre que el usuario solo quiera implementar algo para ciertos casos, o si los diferentes tipos de datos deben tratarse de manera diferente. La sintaxis básica de unSi declaración es:

```
si (expresión_lógica) {
        expresión_1
        . . .
}
```

En este marco, la expresión_lógica es una declaración booleana, y expresión_1 representa los comandos que el usuario quisiera aplicar siempre que la expresión lógica sea verdadera.

Como un ejemplo de juguete de cómo funciona esto, supongamos que quisiéramos simular un proceso en el que dibujamos dos números del 1 al 10 (con números repetidos permitidos). La muestra El comando nos permite simular este proceso fácilmente, y si lo incrustamos en un por bucle podemos repetir el experimento varias veces (digamos, 100 intentos). Si quisiéramos saber en cuántos ensayos salieron ambos números, podríamos determinar esto con unSi declaración. Nuestro código se junta así:

```
even.count <-0
para (i en 1: 100) {
        a <-muestra (c (1:10), 2, reemplazar = VERDADERO)
        si (suma (a %% 2) == 0) {
                even.count <-even.count + 1}

}
even.count
```

La primera línea crea un escalar llamado even.count y lo pone a cero. La siguiente línea inicia el ciclo que nos da 100 pruebas. La tercera línea crea nuestra muestra de dos números del 1 al 10 y nombra la muestrauna. La cuarta línea define nuestro Si declaración: Usamos la función módulo para encontrar el resto cuando cada término de nuestra muestra se divide por dos.[4] Si el resto de cada término es cero, entonces todos los números son pares y la suma es cero. En ese caso, hemos extraído una muestra en la que los números son pares. Por tanto, cuandosuma (a %% 2) == 0 es cierto, entonces queremos agregar uno a un recuento continuo de cuántas muestras de dos números pares tenemos. Por lo tanto, la quinta línea de nuestro código se suma a nuestro recuento, pero solo en los casos que cumplen con nuestra condición. (Observe que una asignación recursiva paraeven.count es aceptable. R tomará el valor anterior, le agregará uno y actualizará el valor guardado). Pruebe este experimento usted mismo. Como sugerencia, la teoría de la probabilidad diría que el 25% de los ensayos producirán dos números pares, en promedio.

Los usuarios también pueden hacer Si. . . demásdeclaraciones de ramificación tales que un conjunto de operaciones se aplica siempre que una expresión es verdadera, y otro conjunto de operaciones se aplica cuando la expresión es falsa. Esta estructura básica se ve así:

```
si (expresión_lógica) {
        expresión_1
        . . .
} demás {
        expresión_2
        . . .
}
```

[4] Recuerde del Cap. 1 que la función módulo nos da el resto de la división.

En este caso, expresión_1 solo se aplicará la expresión en los casos en que la lógica se es verdadera, y expresión_2 solo será una expresión aplica en los casos en que la lógica es falsa.

Finalmente, los usuarios tienen la opción de ramificarse aún más. Con los casos en los que la primera expresión lógica es falsa, por lo que se llama a las expresiones siguientesdemás, estos casos se pueden volver a ramificar con otro Si. . . demásdeclaración. De hecho, el programador puede anidar tantosSi. . . demásdeclaraciones como le gustaría. Para ilustrar esto, considere nuevamente el caso cuando simulamos dos números aleatorios del 1 al 10. Imagine que esta vez queremos saber no solo con qué frecuencia sacamos dos números pares, sino también con qué frecuencia sacamos dos números impares y con qué frecuencia sacamos un número par y uno impar. Una forma en que podríamos abordar esto es manteniendo tres recuentos activos (a continuación,cuenta par, cuenta impar, y split.count) y agregando declaraciones de ramificación adicionales:

```
even.count <-0
cuenta impar <-0
split.count <-0
para (i en 1: 100) {
        a <-muestra (c (1:10), 2, reemplazar = VERDADERO)
        si (suma (a %% 2) == 0) {
                even.count <-even.count + 1} else if
                (sum (a %% 2) == 2) {
                        recuento.imor <-cuenta.odd + 1
                                } demás{
                                        split.count <-split.count + 1}

}
even.count
cuenta impar
split.count
```

Nuestra por El ciclo comienza igual que antes, pero después de nuestro primer Si declaración, seguimos con una demás declaración. Cualquier muestra que no consista en dos números pares ahora está sujeta a los comandos bajodemás, y el primer comando bajo demás es. . . otroSi declaración. Esta siguienteSi La declaración observa que si ambos términos en la muestra son impares, entonces la suma de los residuos después de dividir por dos será dos. Por lo tanto, todas las muestras con dos entradas impares ahora están sujetas a los comandos de este nuevoSi declaración, donde vemos que nuestro cuenta impar El índice se incrementará en uno. Por último, tenemos undemás declaración: todas las muestras que no constan de dos números pares o impares ahora estarán sujetas a este conjunto final de comandos. Dado que estas muestras constan de un número par y otro impar, lasplit.count El índice se incrementará en uno. Pruébelo usted mismo. Nuevamente, como sugerencia, la teoría de la probabilidad indica que en promedio el 50% de las muestras deben consistir en un número par y uno impar, el 25% debe consistir en dos números pares y el 25% debe consistir en dos números impares.

11.5 Optimización y estimación de máxima verosimilitud

La optim comando en R permite a los usuarios encontrar el mínimo o el máximo de una función utilizando una variedad de **optim**métodos de ización.[5] En otras palabras, optim tiene varios algoritmos que buscan de manera eficiente el valor más alto o más bajo de una función, varios de los cuales permiten al usuario restringir los valores posibles de las variables de entrada. Si bien hay muchos usos de la optimización en la investigación de ciencias políticas, el más común es*estimación de máxima verosimilitud (*MLE).[6]

La optim comando, emparejado con Rde fi nición de función simple (discutida en la Secc. 11,2), permite R para utilizar fácilmente la plena fl exibilidad de MLE. Si un modelo enlatado como los descritos en los Cap.1-7 no se adapta a su investigación y desea derivar su propio estimador utilizando MLE, entonces R puede acomodarlo.

Para ilustrar cómo funciona la programación de anMLE en R, considere un ejemplo simple: el estimador del parámetro de probabilidad () en una distribución binomial. A modo de recordatorio, la motivación detrás de una distribución binomial es que estamos interesados en alguna prueba que tenga uno de dos resultados cada vez, y repetimos esa prueba varias veces. El ejemplo más simple es que lanzamos una moneda, que saldrá cara o cruz en cada lanzamiento. Usando este ejemplo, suponga que nuestros datos consisten en 100 lanzamientos de moneda y 43 de los lanzamientos salieron cara. ¿Cuál es la estimación MLE de la probabilidad de que salga cara? Por intuición o derivación, debes saber quesobredosis $\frac{43}{100}$ D 0:43. Seguiremos estimando esta probabilidad en R para practicar para casos más complicados.

Para definir más formalmente nuestro problema, una distribución binomial está motivada por completar *norte* Ensayos de Bernoulli, o ensayos que pueden terminar en un éxito o en un fracaso. Registramos el número de veces que logramos nuestro*norte* ensayos como *y.* Además, por definición, nuestro parámetro de probabilidad () debe ser mayor o igual a cero y menor o igual a uno. Nuestra función de verosimilitud es:

$$L.\,\text{j}norte;\,y\,/\text{D}\,{}_{y.}1\qquad {}_{/\,Nueva\,York} \tag{11,2}$$

Para facilitar el cálculo y el cálculo, podemos obtener un resultado equivalente maximizando nuestra función logarítmica de verosimilitud:

$$\text{". j}norte;\,y\,/\text{D Iniciar sesión }L.\,\text{j}norte;\,y\,/\text{D }y\,\text{Iniciar sesión. /C .}Nueva\,York/\text{Iniciar sesión.1}\qquad /\tag{11,3}$$

Podemos definir fácilmente nuestra función logarítmica de verosimilitud en R con el función mando:

```
binomial.loglikelihood <- función (prob, y, n) {
    loglikelihood <- y * log (prob) + (ny) * log (1-prob) return
    (loglikelihood)
}
```

[5] Ver Nocedal y Wright (1999) para una revisión de cómo funcionan estas técnicas.

[6] Cuando se utiliza la estimación de máxima verosimilitud, como alternativa al uso optim es usar el maxLik paquete. Esto puede ser útil específicamente para la máxima verosimilitud, aunqueoptim tiene la ventaja de poder maximizar o minimizar otros tipos de funciones también, cuando sea apropiado.

Ahora para estimar Oh nosotros necesitamos R para encontrar el valor de p que maximiza el log-verosimilitud dados los datos. (A este término lo llamamosproblema en el código para evitar confusiones con Rel uso del comando Pi para almacenar la constante geométrica.) Podemos hacer esto con el optim mando. Calculamos:

```
prueba <- optim (c (.5),                    # valor inicial para el problema
    binomial.loglikelihood, method =        # la función logarítmica de verosimilitud
    "BFGS",                                 # método de optimización
    arpillera = VERDADERO,                  # return arpillera numérica
    control = lista (fnscale = -1), y = 43, n   # maximizar en lugar de minimizar
    = 100)                                  # los datos
imprimir (prueba)
```

Recuerde que todo lo que sigue a un signo de almohadilla (#) es un comentario que R ignora, por lo que estas notas sirven para describir cada línea de código. Siempre comenzamos con un vector de valores iniciales con todos los parámetros para estimar (solo en este caso), nombre el función logarítmica de verosimilitud que definimos en otro lugar, elija nuestro método de optimización (**B** Royden-**F**letcher**GRAMO**oldfarb-**S**hanno es a menudo una buena opción), e indicar que queremos R para devolver el hessiano numérico para poder calcular los errores estándar más tarde. La quinta línea de código es de vital importancia: por defectooptim es un *minimizador,* así que tenemos que especificar fnscale = -1 para que sea un *maximizador.* En cualquier momento que uses optim por *máximo* estimación de probabilidad, esta línea deberá incluirse. En la sexta línea, enumeramos nuestros datos. A menudo, aquí llamaremos a una matriz o marco de datos, pero en este caso solo necesitamos enumerar los valores de$y$ y *norte.*

Nuestra salida de imprimir (prueba) se parece a lo siguiente:

```
$ par
[1] 0,4300015

$ valor
[1] -68.33149

$ cuenta
gradiente de función
          13           4

$ convergencia
[1] 0

$ mensaje
NULO

$ arpillera
             [, 1]
[1,] -407,9996
```

Para interpretar nuestro resultado: par término enumera las estimaciones de los parámetros, por lo que nuestra estimación es sobredosis 0:43 (como anticipamos). La probabilidad logarítmica de nuestra solución final es68: 33149, y se presenta bajo valor. El termino cuenta nos dice con qué frecuencia optim Tuve que llamar a la función y al gradiente. El terminoconvergencia se codificará como 0 si la optimización se completó con éxito; cualquier otro valor es un código de error. Lamensaje El elemento puede devolver otra información necesaria del optimizador. Por último, elarpillera es nuestra matriz de segundas derivadas de la función de verosimilitud. Con solo un parámetro aquí, es un simple1 1 matriz. En general, si el usuario quiere*errores estándar* a partir de un modelo de máxima verosimilitud estimada, la siguiente línea los devolverá:

```
sqrt (diag (resolver (-prueba $ arpillera)))
```

Esta línea se basa en la fórmula para errores estándar en la estimación de máxima verosimilitud. Todo lo que el usuario necesitará cambiar es reemplazar la palabraprueba con el nombre asociado a la llamada a optim. En este caso, R informa que el estándar el error es SE. O / D 0: 0495074.

Finalmente, en este caso en el que tenemos un solo parámetro de interés, tenemos la opción de usar nuestra función de probabilidad logarítmica definida para dibujar una imagen del problema de optimización. Considere el siguiente código:

```
regla <- seq (0,1,0.01)
loglikelihood <- binomial.loglikelihood (regla, y = 43, n = 100) plot (regla,
loglikelihood, type = "l", lwd = 2, col = "blue",
    xlab = expresión (pi), ylab = "Log-Likelihood", ylim = c (-300, -70), main = "Log-
    Likelihood for Binomial Model")
abline (v = .43)
```

La primera línea define todos los valores que posiblemente pueden tomar. La segunda línea inserta en la función logarítmica de verosimilitud el vector de valores posibles de, más los valores verdaderos de *y* y *norte.* Las líneas tercera a quinta son una llamada a gráfico eso nos da un gráfico lineal de la función de probabilidad logarítmica. La última línea dibuja una línea vertical en nuestro valor estimado paraO. El resultado de esto se muestra en la Fig. 11,4. Aunque las funciones logarítmicas de verosimilitud con muchos parámetros normalmente no se pueden visualizar, esto nos recuerda que la función que hemos de fi nido todavía se puede utilizar para otros propósitos además de la optimización, si es necesario.

11.6 Programación orientada a objetos

Esta sección da un giro hacia lo avanzado, al presentar una aplicación en programación orientada a objetos. Si bien esta aplicación sintetiza muchas de las otras herramientas discutidas en este capítulo, es menos probable que los usuarios novatos utilicen la programación orientada a objetos que las características discutidas hasta ahora. Dicho esto, para usos avanzados, el trabajo orientado a objetos puede ser beneficioso.

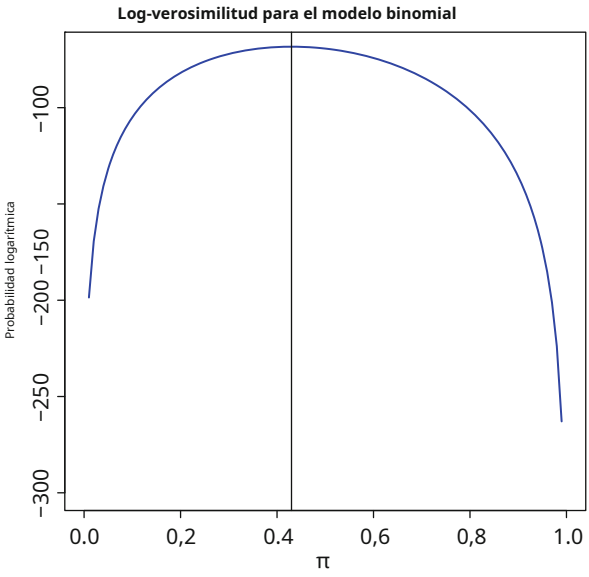

Figura 11.4 Logaritmo de verosimilitud binomial en todos los valores posibles del parámetro de probabilidad cuando los datos constan de 43 éxitos en 100 ensayos

Como se mencionó en el Cap. 1, R es un entorno orientado a objetos. Esto significa que usted, como investigador, tiene la oportunidad de utilizar sofisticadas herramientas de programación en su propia investigación. En la programación orientada a objetos, crea un *clase* de artículo que tiene una variedad de características. Luego puede crear elementos dentro de la clase (llamados *objetos* o *variables)* que tendrá características únicas. EnR, puedes crear clases a partir de los sistemas de objetos S3 o S4.

Para ilustrar, un "modelo lineal" es una clase de objetos creados por el lm mando. Es unS3 clase. De vuelta en el Cap.6, estimamos un modelo que nombramos mod. de horas, que era un objeto de la clase de modelo lineal. Este objeto tenía características que incluyen coeficientes, residuos, y cov. sin escala. En todo S3-objetos de clase, podemos llamar a una característica escribiendo el nombre del objeto, el signo de dólar y el nombre de la característica que queremos. Por ejemplo,coeficientes de $ mod. enumeraría el vector de coeficientes de ese modelo. (S4 utiliza referencias ligeramente diferentes, que se describen más adelante.) Cuando de fi ne sus propios objetos, puede utilizar S3 o S4 sistema de objetos. El siguiente ejemplo guarda nuestras salidas en elS3 sistema de objetos, aunque las notas a pie de página ilustran cómo se puede utilizar el S4 sistema de objetos si lo prefiere.

11.6.1 Simular un juego

Como ejemplo práctico de programación orientada a objetos, consideramos una versión algo simplificada del trabajo presentado en Monogan (2013b).[7] Tocamos este modelo en la Secta. 11,2, presentando la función de utilidad de un actor. En resumen, este artículo desarrolla un modelo de teoría de juegos de cómo dos partes elegirán una posición sobre un tema de acuerdo con el modelo de proximidad espacial de la política. El modelo espacial, descrito algo en el Cap.7, asume que las posiciones de los problemas se pueden representar como ubicaciones en el espacio. Los votantes eligen partidos que adoptan posiciones de emisión más cercanas a su propia posición en el espacio de emisión, por lo que los partidos tratan de maximizar los votos eligiendo estratégicamente sus posiciones de emisión.

En el juego que consideramos, la motivación sustantiva es el hecho de que la opinión pública muestra fuertes tendencias en ciertos temas de política debido a tendencias demográficas o diferencias generacionales. Temas como este incluyen la política ambiental, la política de inmigración y la protección de los derechos de los homosexuales. Suponemos que dos partidos competirán en dos elecciones. Sin embargo, para considerar el hecho de que los partidos deben tener en cuenta el futuro y pueden tener que rendir cuentas por posiciones de asuntos pasados, la posición que toman en la primera elección es la posición en la que están atrapados en la segunda elección. También asumimos que la posición del votante mediano cambia de una elección a la siguiente de una manera que los partidos anticipan, ya que en estos temas las tendencias de la opinión pública suelen ser claras.

Otra característica es que un partido tiene ventaja en la primera elección (debido a la titularidad o algún otro factor), por eso los jugadores son partidos. *A* (una fiesta sin valencia *ventaja* en las primeras elecciones) y partido *D* (una parte desfavorecida). Se supone que la segunda elección es menos valiosa que la primera, porque la recompensa está más lejos. Finalmente, los votantes también consideran factores desconocidos para los partidos, por lo que el resultado de la elección es más probabilístico que seguro. En este juego, entonces, los partidos intentan maximizar su probabilidad de ganar cada una de las dos elecciones. ¿Cómo se posicionan ante un tema que cambia la opinión pública?

La característica engañosa de este juego es que no se puede resolver algebraicamente (Monogan 2013b, pag. 288). Por lo tanto, recurrimos aR para ayudarnos a encontrar soluciones a través de la simulación. Lo primero que debemos hacer es limpiar y definir las funciones de utilidad esperadas para las partes.*A* y *D:*

```
rm (lista = ls ())
Función cuadrática.A <(m.1, m.2, p, delta, theta.A, theta.D) {
      util.a <-plogis (- (m.1-theta.A) ^ 2 +
            (m.1-theta.D) ^ 2 + p) +
        delta * plogis (- (m.2-theta.A) ^ 2 +
```

[7]Para ver la versión original completa del programa, consulte http://hdl.handle.net/1902.1/16781. Tenga en cuenta que aquí usamos elS3 familia de objetos, pero el código original usa el S4 familia. El programa original también considera funciones alternativas de utilidad para votantes además de la función de proximidad cuadrática que se enumera aquí, como una función de proximidad absoluta y el modelo direccional de utilidad para votantes (Rabinowitz y Macdonald1989).

```
            (m.2-theta.D) ^ 2)
        volver (util.a)
        }
D <-función cuadrática (m.1, m.2, p, delta,          theta.A, theta.D) {
        util.d <- (1-plogis (- (m.1-theta.A) ^ 2 +
            (m.1-theta.D) ^ 2 + p)) +
          delta * (1-plogis (- (m.2-theta.A) ^ 2 +
            (m.2-theta.D) ^ 2))
        volver (util.d)
        }
```

Ya consideramos la función Cuadrático.A en la Secta. 11,2, y está formalmente definido por la Ec. (11,1). Cuadrático.D es similar en estructura, pero produce diferentes utilidades. Ambas funciones se utilizarán en las simulaciones que hagamos.

Nuestro siguiente paso es definir una función larga que usa nuestras funciones de utilidad para ejecutar una simulación y produce una salida con el formato que queremos.[8] Todo el siguiente código es una gran definición de función. Los comentarios después de los signos de almohadilla (#) se incluyen nuevamente para ayudar a distinguir cada parte del cuerpo de la función. La función, llamada simular, simula nuestro juego de interés y guarda nuestros resultados en un objeto de clase juego.simulación:

```
simular <-función (v, delta, m.2, m.1 = 0.7, theta = seq (-1,1, .1)) {

        # definir parámetros internos, matrices y precisión de vectores <-length
        (theta)
        resultadoA <-matriz (NA, precisión, precisión) resultadoD
        <-matriz (NA, precisión, precisión) bestResponseA <-rep
        (NA, precisión)
        bestResponseD <-rep (NA, precisión)
        equilibrioA <- 'NA'
        equilibrioD <- 'NA'

        # atributos de matriz
        nombres de fila (resultA) <- colnames (resultA) <- nombres de fila (resultD) <-
            colnames (resultD) <- nombres (bestResponseA) <nombres
            (bestResponseD) <- theta

        # complete las utilidades para todas las estrategias para el partido A para (i
        en 1: precisión) {
                para (j en 1: precisión) {
                    resultadoA [i, j] <- Cuadrático.A (m.1, m.2, v, delta,
                        theta [i], theta [j])
                    }
```

[8] Si prefieres usar el S4 sistema de objetos para este ejercicio, lo siguiente que tendríamos que hacer es definir la clase de objeto antes de definir la función. Si quisiéramos llamar a nuestra clase de objetosimulación, escribiríamos: setClass ("simulación", representación (resultA = "matriz", resultD = "matriz", bestResponseA = "numérico", bestResponseD = "numérico", equilibriumA = "carácter", equilibriumD = "carácter")). La S3 sistema de objetos no requiere este paso, como veremos cuando creamos objetos de nuestra autodefinición juego.simulación clase.

```
            }

        # utilidades para la fiesta D para
        (i en 1: precisión) {
                para (j en 1: precisión) {
                    resultadoD [i, j] <- Cuadrático.D (m.1, m.2, v, delta,
                        theta [i], theta [j])
                    }
            }

        # mejores respuestas para la fiesta A
        para (i en 1: precisión) {
            bestResponseA [i] <- which.max (resultA [, i])}


        # mejores respuestas para el partido D
        para (i en 1: precisión) {
            bestResponseD [i] <- which.max (resultD [i,])}


        # encontrar los equilibrios para (i
        en 1: precisión) {
            if (bestResponseD [bestResponseA [i]] == i) {
                equilibriumA <-dimnames (resultA) [[1]] [
                    bestResponseA [i]]
                equilibriumD <-dimnames (resultD) [[2]] [
                    bestResponseD [bestResponseA [i]]]
                    }
        }

        # guardar la salida result <-list (resultA = resultA, resultD = resultD,

            bestResponseA = bestResponseA, bestResponseD = bestResponseD,
          equilibriumA = equilibriumA, equilibriumD = equilibriumD) class (result) <-
          "game.simulation"
        invisible (resultado)
    }
```

Como consejo general, al escribir una función larga, es mejor probar el código fuera del función envoltura. Para obtener puntos de bonificación y tener una idea completa de este código, es posible que desee dividir esta función en sus componentes y ver cómo funciona cada pieza. Como muestran los comentarios a lo largo de todo, cada conjunto de comandos hace algo que normalmente haríamos en una función. Observe que cuando se definen los argumentos, ambosm.1 y theta se les dan valores predeterminados. Esto significa que en usos futuros, si no especificamos los valores de estas entradas, la función utilizará estos valores predeterminados, pero si los especificamos, los valores predeterminados se anularán. Pasando a los argumentos dentro de la función, comienza definiendo los parámetros internos y estableciendo sus atributos. Cada conjunto de comandos a partir de entonces es un ciclo dentro de un ciclo para repetir ciertos comandos y completar vectores y matrices de salida.

La adición clave aquí que es única a todo lo discutido antes está en la definición de la resultado término al final de la función. Tenga en cuenta que al hacer esto,

Primero definimos la salida como un lista, en el que se nombra cada componente. En este caso, ya nombramos nuestros objetos de la misma manera que queremos etiquetarlos en la lista de salida, pero puede que no siempre sea así.[9] Luego usamos el clase comando para declarar que nuestra salida es del juego.simulación clase, un concepto que estamos creando ahora. Laclase el comando formatea esto como un objeto en el S3 familia. Escribiendoinvisible (resultado) al final de la función, sabemos que nuestra función devolverá este juego.simulación-objeto de clase.[10]

Ahora que esta función está definida, podemos ponerla en uso. Refiriéndose a los términos en la ecuación. (11,1), suponer que V D 0: 1, ı D 0, $metro_1$ D 0: 7, y $metro_2$ D 0: 1. En el siguiente código, creamos un nuevo objeto llamado tratamiento.1 utilizando la simular función cuando los parámetros toman estos valores, y luego preguntamos R para imprimir la salida:

```
tratamiento.1 <-simular (v = 0.1, delta = 0.0, m.2 = -0.1)
tratamiento.1
```

Tenga en cuenta que no especificamos m.1 porque 0,7 ya es el valor predeterminado para ese parámetro. La salida de la impresión detratamiento.1 es demasiado extenso para reproducirlo aquí, pero en su propia pantalla verá que tratamiento.1 es un objeto de la juego.simulación clase, y la impresión informa los valores para cada atributo.

Si solo estamos interesados en una característica particular de este resultado, podemos preguntar R para devolver solo el valor de una ranura específica. Por unS3 object, que es como guardamos nuestro resultado, podemos llamar a un atributo nombrando el objeto, usando el símbolo $ y luego nombrando el atributo que queremos usar. (Esto contrasta conS4 objetos, que utilizan el @ símbolo para llamar ranuras.) Por ejemplo, si solo quisiéramos ver cuáles son las opciones de equilibrio para las partes A y D fueron, simplemente podríamos escribir:

```
tratamiento.1 $ equilibrioA
tratamiento.1 $ equilibrioD
```

Cada uno de estos comandos devuelve el mismo resultado:

```
[1] "0,7"
```

Entonces, sustancialmente, podemos concluir que el equilibrio para el juego bajo estos parámetros es $_A$D $_D$D 0: 7, que es el punto ideal del votante mediano en la primera

[9] Por ejemplo, en la frase resultadoA = resultadoA, el término a la izquierda del signo igual indica que este término en la lista se llamará resultA, y el término a la derecha llama al objeto de este nombre de la función para llenar este lugar.

[10] El código diferiría ligeramente para el S4 sistema de objetos. Si de fi nimos nuestrosimulación clase como describe la nota al pie anterior, aquí reemplazaríamos la definición de resultado como sigue: resultado <-nuevo ("simulación", resultadoA = resultadoA, resultD = resultD, bestResponseA = bestResponseA, bestResponseD = bestResponseD, equilibrioA = equilibrioA, equilibrioD = equilibrioD). Reemplazo de la lista comando con el nuevo comando asigna la salida a nuestro simulación class y llena las distintas ranuras en un solo paso. Esto significa que podemos omitir el paso adicional de usar elclase comando, que necesitábamos al usar el S3 sistema.

elección. Sustancialmente, esto debería tener sentido porque la configuraciónI D 0 Es un caso especial cuando los partidos no están preocupados en absoluto por ganar la segunda elección, por lo que se posicionan estrictamente para la primera elección.[11]

Para dibujar un contraste, ¿y si realizamos una segunda simulación, pero esta vez aumentamos el valor de I a 0,1? También podríamos usar valores más precisos de las posiciones que podrían tomar las partes, estableciendo su posición hasta las centésimas decimales, en lugar de las décimas. Podríamos hacer esto de la siguiente manera:

```
tratamiento.2 <-simular (v = 0.1, delta = 0.1, m.2 = -0.1,
        theta = seq (-1,1, .01))
tratamiento.2 $ equilibrioA
tratamiento.2 $ equilibrioD
```

Logramos la precisión más fina sustituyendo nuestro propio vector por theta.

Nuestros valores de salida de las dos ranuras de equilibrio son nuevamente los mismos:

[1] "0,63"

Entonces sabemos que bajo este segundo tratamiento, el equilibrio para el juego es $_A$D $_D$D 0:63. Sustancialmente, lo que sucedió aquí es que aumentamos el valor de ganar el Segunda elección a los partidos. Como resultado, los partidos movieron su posición de tema un poco más cerca de la preferencia de tema del votante medio en la segunda elección.

11.7 Análisis de Monte Carlo: un ejemplo aplicado

Como ejemplo aplicado que sintetiza varias de las herramientas desarrolladas en este capítulo, ahora realizamos un análisis de Monte Carlo. La lógica básica del análisis de Monte Carlo es que el investigador genera datos conociendo el verdadero modelo de población. Luego, el investigador pregunta si las estimaciones muestrales de un determinado método tienen buenas propiedades, dadas las cantidades de población. Los tratamientos comunes en un experimento de Monte Carlo incluyen: elección de estimador, tamaño de muestra, distribución de predictores, varianza de errores y si el modelo tiene la forma funcional correcta (por ejemplo, ignorar una relación no lineal u omitir un predictor). Al probar varios tratamientos, un usuario puede tener una idea comparativa de qué tan bien funciona un estimador.

En este caso, presentaremos Signorino's (1999) probit multinomial estratégico modelo como estimador que usaremos en nuestro experimento de Monte Carlo. La idea detrás de este enfoque es que, cuando una situación política recurrente se puede representar con un juego (como las disputas internacionales militarizadas), podemos desarrollar un modelo empírico de los resultados que pueden ocurrir (como la guerra) basado en la estrategia del juego. Cada resultado posible tiene un*función de utilidad* para cada jugador, ya sea que el jugador sea un individuo, un país u otro actor. La utilidad representa cuánto

[11] Si en cambio hubiéramos salvado tratamiento.1 como un S4 simulación objeto como se describe en las notas a pie de página anteriores, el comando para llamar a un atributo específico sería en su lugar: tratamiento.1@equilibriumA.

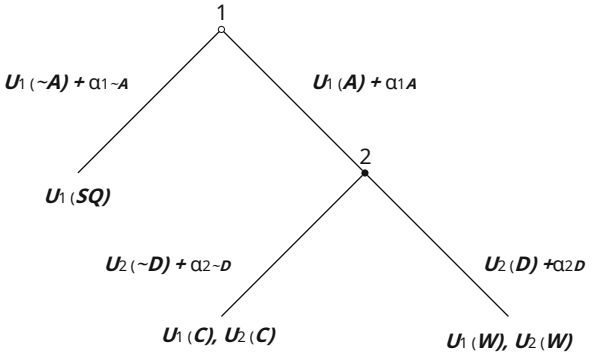

Figura 11.5 Modelo de disuasión estratégica

el beneficio que obtiene el jugador del resultado. En esta configuración, utilizamos predictores observables para modelar las utilidades para cada resultado posible. Este tipo de modelo es interesante porque el proceso de generación de datos de población generalmente tiene no linealidades que no son capturadas por enfoques estándar. Por tanto, tendremos que programar nuestra propia función de verosimilitud y utilizaroptim para estimarlo.[12]

Para motivar cómo se establece el probit estratégico multinomial, considere un modelo sustantivo de comportamiento entre dos naciones, un agresor (1) y un objetivo (2). Este es un juego de dos etapas: primero, el agresor decide si atacar (A) o no (A); luego, el objetivo decide si defender (D) o no (D). Pueden ocurrir tres consecuencias observables: si el agresor decide no atacar, el**status quo** sostiene; si el agresor ataca, el objetivo puede optar por**capitular;** finalmente, si el objetivo defiende, tenemos **guerra.** El árbol del juego se presenta en la Fig. 11,5. Al final de cada rama están las utilidades para los jugadores (1) y (2) para**status quo (**SQ), **capitulación**C) y **guerra (**W), respectivamente. Suponemos que estos jugadores son racionales y, por lo tanto, elegimos las acciones que maximizan el resultado saldar. Por ejemplo, si el objetivo es débil, la recompensa de la guerra$U_{2}.W/$ seria terriblemente negativo, y probablemente más bajo que la recompensa por la capitulación $U_{2}.C/.$ Si el agresor lo sabe, puede deducir que en caso de ataque, el objetivo capitularía. La Por lo tanto, el agresor sabría que si elige atacar, recibirá la recompensa. $U_{1}.C/$ (y no $U_{1}.W/$). Si la recompensa de la capitulación $U_{1}.C/$ es más grande que la recompensa del status quo U_{1}.SQ / para el agresor, la decisión racional es atacar (y la decisión racional del objetivo es capitular). Por supuesto, el objetivo es tener una idea de lo que sucederá a medida que cambien las circunstancias en diferentes díadas de naciones.

En general, asumimos que las funciones de utilidad responden a circunstancias específicas observables (el valor de los recursos en disputa, el precio diplomático esperado debido a las sanciones en caso de agresión, el poderío militar, etc.). Nosotros

[12]Los lectores interesados en realizar una investigación como esta en su propio trabajo deben leer sobre la juegos paquete, que fue desarrollado para estimar este tipo de modelo.

también asumirá que las utilidades para la elección de cada país son estocásticas. Esto introduce incertidumbre en nuestro modelo de pagos, lo que representa el hecho de que este modelo está incompleto y probablemente excluye algunos parámetros relevantes de la evaluación. En este ejercicio, consideraremos solo cuatro predictores *Xi* para modelar las funciones de utilidad, y suponga que los pagos son lineales en estas variables.

En particular, consideramos la siguiente forma paramétrica para los pagos:

$$\begin{aligned} U_1.SQ / &= 0 \\ U_1.C / &= X_1\check{}_1 \\ U_1.W / &= X_2\check{}_2 \\ U_2.C / &= 0 \\ U_2.W / &= X_3\check{}_3 + X_4\check{}_4 \\ \text{¸} \quad & N.0; 0: 5 / \end{aligned} \tag{11,4}$$

Cada nación hace su elección basándose en qué decisión le dará una mayor utilidad. Esto se basa en la información conocida, más una perturbación privada desconocida (¸) para cada elección. Esta perturbación privada agrega un elemento aleatorio a las decisiones de los actores. Nosotros, como investigadores, podemos mirar los conflictos pasados, medir los predictores (*X*), observar el resultado, y me gustaría inferir: ¿Qué importancia tienen X_1, X_2, X_3, y X_4 en el comportamiento de las díadas-nación? Tenemos que determinar esto basándonos en si cada dato el punto resultó en **status quo, capitulación,** o **guerra.**

11.7.1 Función log-verosimilitud de disuasión estratégica

Podemos determinar estimaciones de cada ˇÖ utilizando la estimación de máxima verosimilitud. Primero, tenemos que determinar la probabilidad (*pag*) el objetivo (2) se defenderá si es atacado:

$$\begin{aligned} pag &= PD / = PU_2.D / + {}_{¿2,D>} U_2.D / + {}_{¿2,D/} \\ &= PU_2.D / U_2.D /> {}_{¿2,D} \quad {}_{¿2,D/} \\ &= \circ.X_3\check{}_3 + X_4\check{}_4 / \end{aligned} \tag{11,5}$$

Dónde ˚ es la función de distribución acumulada para una distribución normal estándar. Si sabemos *pag,* podemos determinar la probabilidad (*q*) que el agresor (1) atacará:

$$\begin{aligned} q &= PENSILVANIA / = PU_1.A / + {}_{¿1,A>} U_{una.}A / + {}_{¿1,A/} \\ &= PU_1.A / U_1.A /> {}_{¿1,A} \quad {}_{¿1,A/} \\ &= \circ.pX_2\check{}_2 + .1 - p / X_1\check{}_1 / \end{aligned} \tag{11,6}$$

Darse cuenta de *pag* está en la ecuación para *q.* Esta característica no lineal del modelo no se adapta a los modelos enlatados estándar.

Conociendo las fórmulas para *pag* y *q,* conocemos las probabilidades del statu quo, la capitulación y la guerra. Por lo tanto, la función de verosimilitud es simplemente el producto de las probabilidades de cada evento, elevado a una variable ficticia de si el evento sucedió, multiplicado por todas las observaciones:

$$L.\check{}\text{j}\mathbf{y;\ X/}\text{D}\prod_{\text{JD1}}^{\text{norte}}.1q\,/_{D\text{SQ}}\quad .q.1\ pag//_{DC}\quad .pq\,/_{DW} \tag{11,7}$$

Dónde $D_{SQ,}$ $D_{C,}$ y $D_W$son variables ficticias iguales a 1 si el caso es statu quo, capitulación o guerra, respectivamente, y 0 en caso contrario. La función logarítmica de verosimilitud es:

$$".\check{}\text{j}\mathbf{y;\ X/}\text{D}\sum_{\text{JD1}}^{\text{norte}}{}_{D\text{SQ}}\text{Iniciar sesión.1}\ q\,/\text{C}D_C\text{Iniciar sesión}\ q.1\ pag/\text{C}D_W\text{Iniciar sesión.}pq\,/ \tag{11,8}$$

Ahora estamos listos para comenzar a programar esto en R. Comenzamos limpiando y luego definiendo nuestra función logarítmica de verosimilitud como llik, que nuevamente incluye comentarios para describir partes del cuerpo de la función:

```
rm (lista = ls ())

llik = función (B, X, Y) {
    # Separe las matrices de datos de las variables individuales: sq =
    as.matrix (Y [, 1])
    cap = as.matrix (Y [, 2])
    war = as.matrix (Y [, 3])
    X13 = as.matrix (X [, 1])
    X14 = as.matrix (X [, 2])
    X24 = as.matrix (X [, 3: 4])

    # Coeficientes separados para cada ecuación: B13 =
    as.matrix (B [1])
    B14 = como.matriz (B [2])
    B24 = como.matriz (B [3: 4])

    # Defina las utilidades como variables multiplicadas por coeficientes:
    U13 = X13% *% B13
    U14 = X14% *% B14
    U24 = X24% *% B24

    # Calcule la probabilidad de que 2 pelee (P4) o no (P3): P4 = pnorm (U24)

    P3 = 1-P4

    # Calcule la probabilidad de que 1 ataque (P2) o no (P1):
    P2 = normal ((P3 * U13 + P4 * U14))
    P1 = 1-P2

    # Definir y devolver la función logarítmica de verosimilitud:
    lnpsq = log (P1)
    lnpcap = log (P2 * P3)
```

```
    lnpwar = log (P2 * P4)
    llik = sq * lnpsq + cap * lnpcap + war * lnpwar
    return (sum (llik))
}
```

Aunque sustancialmente más largo que la función de verosimilitud que de fi nimos en la Sec. 11,5, la idea es la misma. La función aún acepta valores de parámetros, variables independientes y variables dependientes, y aún devuelve un valor de probabilidad logarítmica. Sin embargo, con un modelo más complejo, la función debe dividirse en pasos de componentes. Primero, nuestros datos ahora son matrices, por lo que el primer lote de código separa las variables por ecuación del modelo. En segundo lugar, nuestros parámetros son todos coeficientes almacenados en el argumentoB, por lo que necesitamos separar los coeficientes por ecuación del modelo. En tercer lugar, multiplicamos matricialmente las variables por los coeficientes para crear los tres términos de utilidad. Cuarto, usamos esa información para calcular las probabilidades de que el objetivo se defienda o no. En quinto lugar, utilizamos las utilidades, más la probabilidad de las acciones del objetivo, para determinar la probabilidad de que el agresor ataque o no. Por último, las probabilidades de todos los resultados se utilizan para crear la función de probabilidad logarítmica.

11.7.2 Evaluación del estimador

Ahora que hemos definido nuestra función de verosimilitud, podemos usarla para simular datos y ajustar el modelo a nuestros datos simulados. Con cualquier experimento de Monte Carlo, debemos comenzar por definir el número de experimentos que realizaremos para un tratamiento dado y el número de puntos de datos simulados en cada experimento. También necesitamos definir espacios vacíos para los resultados de cada experimento. Escribimos:

```
set.seed (3141593)
i <-100 # número de experimentos
n <-1000 # número de casos por experimento
beta.qre <-matriz (NA, i, 4) stder.qre
<-matriz (NA, i, 4)
```

Empezamos usando set.seed para hacer que nuestros resultados de Monte Carlo sean más replicables. Aquí dejamosI sea nuestro número de experimentos, que establecemos en 100 (aunque normalmente preferimos un número mayor). norte como nuestro número de casos. Esto nos permite de fi nirbeta.qre y stder.qre como matrices de salida para nuestras estimaciones de los coeficientes y errores estándar de nuestros modelos, respectivamente.

Con esto en su lugar, ahora podemos ejecutar un gran ciclo que simulará repetidamente un conjunto de datos, estimará nuestro modelo probit estratégico multinomial y luego registrará los resultados. El bucle es el siguiente:

```
para (j en 1: i) {
        # Simular variables causales x1 <-
        rnorm (n)
        x2 <-rnorm (n)
        x3 <-rnorm (n)
        x4 <-rnorm (n)
```

```
        # Crear utilidades y términos de error u11 <-
        rnorm (n, sd = sqrt (.5))
        u13 <-x1
        u23 <-rnorm (n, sd = sqrt (.5)) u14 <-
        x2
        u24 <-x3 + x4 + rnorm (n, sd = sqrt (.5)) pR <-
        pnorm (x3 + x4)
        uA <- (pR * u14) + ((1-pR) * u13) + rnorm (n, sd = sqrt (.5))

        # Crear variables dependientes sq <-
        rep (0, n)
        capit <-rep (0, n)
        guerra <-rep (0, n)
        sq [u11> = uA] <- 1
        capit [u11 <uA & u23> = u24] <- 1
        guerra [u11 <uA & u23 <u24] <- 1 Nsq
        <-abs (1-sq)

        # Matrices para Input stval
        <-rep (.1,4)
        depvar <-cbind (sq, capit, war) indvar <-
        cbind (x1, x2, x3, x4)

        # Modelo de ajuste
        strat.mle <-optim (stval, llik, hessian = TRUE, method = "BFGS",
                control = lista (maxit = 2000, fnscale = -1, trace = 1), X = indvar,
                Y = depvar)

        # Guardar resultados
        beta.qre [j,] <- estrat.mle $ par stder.qre [j,] <- sqrt (diag (solve (-strat.mle $
        hessian)))
}
```

En este modelo, establecemos ˇ D 1 para cada coeficiente en el modelo de población. De lo contrario, Eq. (11,4) define completamente nuestro modelo de población. En el ciclo, los primeros tres lotes de código generan datos de acuerdo con este modelo. Primero, generamos cuatro variables independientes, cada una con una distribución normal estándar. En segundo lugar, definimos las utilidades de acuerdo con la ecuación. (11,4), agregando los términos de perturbación aleatoria () como se indica en la Fig. 11,5. En tercer lugar, creamos los valores de las variables dependientes en función de los servicios públicos y las perturbaciones. Después de esto, el cuarto paso es limpiar nuestros datos simulados; definimos valores iniciales paraoptim y unir las variables dependientes e independientes en matrices. Quinto, usamosoptim para estimar realmente nuestro modelo, nombrando la salida dentro de la iteración estrat.mle. Por último, los resultados del modelo se escriben en las matrices. beta.qre y stder.qre.

Después de ejecutar este ciclo, podemos echar un vistazo rápido al valor promedio de nuestro coeficientes estimados (ˇŎ) escribiendo:

```
aplicar (beta.qre, 2, media)
```

En esta llamada a solicitar, estudiamos nuestra matriz de coeficientes de regresión en la que cada fila representa uno de los 100 experimentos de Monte Carlo, y cada columna representa uno de los cuatro coeficientes de regresión. Estamos interesados en los coeficientes, por lo que

tipo 2 para estudiar las columnas, y luego tomar el significar de las columnas. Si bien sus resultados pueden diferir un poco de lo que se imprime aquí, particularmente si no configuró la semilla para que sea la misma, la salida debería verse así:

[1] 1.0037491 1.0115165 1.0069188 0.9985754

En resumen, los cuatro coeficientes estimados están cerca de 1, lo cual es bueno porque 1 es el valor poblacional de cada uno. Si quisiéramos automatizar un poco más nuestros resultados para indicarnos el sesgo en los parámetros, podríamos escribir algo como esto:

```
deviate <- sweep (beta.qre, 2, c (1,1,1,1)) colMeans
(desviate)
```

En la primera línea, usamos el barrer comando, que **barrer**Saca (o resta) la estadística de resumen de nuestra elección de una matriz. En nuestro caso,beta.qre es nuestra matriz de estimaciones de coeficientes en 100 simulaciones. La2 El argumento que ingresamos por separado indica que se debe aplicar la estadística por columna (por ejemplo, por coeficiente) en lugar de por fila (lo que habría sido por experimento). Por último, en lugar de enumerar una estadística empírica que queremos restar, simplemente enumeramos los parámetros de población verdaderos para restar. En la segunda línea, calculamos el sesgo tomando el **Signi**ficar desviación por **columna**umn, o por parámetro. Nuevamente, la salida puede variar con diferentes semillas y generadores de números, pero en general debe indicar que el los valores medios no son muy a partir de los valores reales de la población. Todas las diferencias son pequeños:

[1] 0,003749060 0,011516459 0,006918824 -0,001424579

Otra cantidad que vale la pena es el error absoluto medio, que nos dice cuánto difiere una estimación del valor de la población en promedio. Esto es un poco diferente en el sentido de que las sobreestimaciones y las subestimaciones no pueden desaparecer (como podría ocurrir con el cálculo del sesgo). Un estimador insesgado aún puede tener una gran varianza de error y un gran error medio absoluto. Como ya de fi nimosdesviarse antes, ahora solo necesitamos escribir:

```
colMeans (abs (desviado))
```

Nuestra salida muestra pequeños errores absolutos promedio:

[1] 0.07875179 0.08059979 0.07169820 0.07127819

Para tener una idea de cuán bueno o malo es este rendimiento, deberíamos volver a ejecutar este experimento de Monte Carlo usando otro tratamiento para comparar. Para una comparación simple, podemos preguntar qué tan bien funciona este modelo si aumentamos el tamaño de la muestra. Presumiblemente, nuestro error absoluto medio disminuirá con una muestra más grande, por lo que en cada experimento podemos simular una muestra de 5000 en lugar de 1000. Para hacer esto, vuelva a ejecutar todo el código de esta sección del capítulo, pero reemplace una sola línea. Al definir el número de casos, la tercera línea después del inicio de la sección.11.7.2—En lugar de escribir n <-1000, escriba en su lugar: n <-5000. Al final del programa, cuando se reportan los errores absolutos medios, verá que son más pequeños. Nuevamente, variarán de una sesión a otra, pero el resultado final debería verse así:

[1] 0.03306102 0.02934981 0.03535597 0.02974488

Como puede ver, los errores son más pequeños que con 1000 observaciones. Nuestro tamaño de muestra es considerablemente mayor, por lo que esperamos que este sea el caso. Esto nos da una buena comparación.

Con la finalización de este capítulo, ahora debería tener el kit de herramientas necesario para programar en R siempre que su investigación tenga necesidades únicas que no puedan ser abordadas por comandos estándar, o incluso por paquetes aportados por el usuario. Con la finalización de este libro, debería tener una idea del amplio alcance de lo queR puede hacer por los analistas políticos, desde la gestión de datos hasta los modelos básicos y la programación avanzada. Una verdadera fuerza deR es la flexibilidad que ofrece a los usuarios para abordar las complejidades y los problemas originales que puedan enfrentar. A medida que proceda a utilizarR en su propia investigación, asegúrese de continuar consultando los recursos en línea para descubrir nuevas y prometedoras capacidades a medida que surjan.

11.8 Problemas de práctica

1. Distribuciones de probabilidad: Calcule la probabilidad de cada uno de los siguientes eventos:

   una. Una variable estándar distribuida normalmente es mayor que 3.
   B. Una variable distribuida normalmente con media 35 y desviación estándar 6 es mayor que 42.
   C. $X < 0$: 9 Cuándo X tiene la distribución uniforme estándar.

2. Bucles: Let *hx; norte*/D 1 C *X* C X_2 C C X_{norte} D PAG_{norte} $_{ID0}$ X_I. Escribe un R programa para calcular *h*.0: 9; 25 /usando un por círculo.
3. Funciones: En la pregunta anterior, escribió un programa para calcular *hx; norte*/D PAG_{norte} $_{ID0}$ $X_I$por *X* D 0: 9 y *norte* D 25. Convierta este programa en una función más general que toma dos argumentos, *X* y *norte,* y vuelve *hx; norte*/.Usando la función, determinar los valores de *h*.0: 8; 30 /,*h*.0: 7; 50 /,y *h*.0:95; 20 /.
4. Estimación de máxima verosimilitud. Considere un ejemplo aplicado de Signorino (1999) método probit estratégico multinomial. Descargue un subconjunto de disputas interestatales militarizadas del siglo XIX, el archivo en formato Statawar1800.dta, del Dataverse (consulte la página vii) o del contenido en línea de este capítulo (consulte la página 205). Estos datos provienen de fuentes como EUGene (Bueno de Mesquita y Lalman 1992) y el Proyecto Correlatos de Guerra (Jones et al. 1996). Programe una función de verosimilitud para un modelo como el que se muestra en la Fig.11,5 y estimar el modelo para estos datos reales. Los tres resultados son:**guerra (**codificado 1 si los países entraron en guerra), **sq (**codificado 1 si el status quo se mantuvo), y **capit (**codificado 1 si el objetivo país capitulado). Asumir que $U_{1.SQ}$ / es impulsado por **pacifistas**número de años desde que la díada estuvo en conflicto por última vez) y **s_wt_re1 (**Puntaje S para política similitud de estados, ponderado por la región del agresor). $U_{1.W}$ / es una función de **balanc**la capacidad militar del agresor en relación con la capacidad combinada de

la díada). $U_{2.}W/$ es una función de una constante y **balanc.** $U_{1.}C/$ es una constante, y $U_{2.}C/$ es cero.

una. Informe sus estimaciones y los errores estándar correspondientes.

B. ¿Qué coeficientes se distinguen estadísticamente de cero?

C. Bonificación: dibuje la probabilidad predicha de**guerra** Si **balanc** se manipula desde su mínimo de 0 hasta su máximo de 1, mientras que **pacifistas** y **s_wt_re1** se mantienen en sus mínimos teóricos de 0 y 1, respectivamente.

- Solo por diversión: si dibuja el mismo gráfico de probabilidades predichas, pero permite **balanc** para ir hasta 5, realmente puede ilustrar el tipo de relación no monótona que permite este modelo. Sin embargo, recuerde que no desea interpretar resultados fuera del rango de sus datos. Esta es solo una ilustración divertida para practicar más.

5. Análisis y optimización de Monte Carlo: Replica el experimento en Signorino's (1999) probit multinomial estratégico de la Sect. 11,7. ¿Hay mucha diferencia entre sus valores promedio estimados y los valores de la población? ¿Obtiene errores medios absolutos similares a los informados en esa sección? ¿Cómo se comparan sus resultados cuando prueba los siguientes tratamientos?

una. Suponga que disminuye la desviación estándar dex1, x2, x3, y x4 a 0,5 (a diferencia del tratamiento original que suponía una desviación estándar de 1). ¿Mejoran o empeoran sus estimaciones? ¿Qué sucede si reduce aún más la desviación estándar de estos cuatro predictores, a 0,25? ¿Qué patrón general ve y por qué lo ve? (*Recuerda:* Todo lo demás sobre estos tratamientos, como el tamaño de la muestra y el número de experimentos, debe ser el mismo que el del experimento de control original. De lo contrario, todo lo demás no se mantiene igual).

B. Bonificación: ¿Qué sucede si tiene una variable omitida en el modelo? Cambie su función de probabilidad de registro para excluirx4 de su procedimiento de estimación, pero continúe incluyendo x4 cuando simula los datos en el por círculo. Dado que solo estima los coeficientes parax1, x2, y x3, ¿Cómo se comparan sus estimaciones en este tratamiento con el tratamiento original en términos de sesgo y error absoluto medio?