Generación de números aleatorios

Introducción

Tu computadora genera números $U_1, U_2, U_3, \ldots$ que se consideran independientes y distribuidos uniformemente de manera aleatoria en el intervalo continuo $(0, 1)$.

Recuerda que la distribución de probabilidad (función de distribución acumulativa) de tal variable aleatoria uniformemente distribuida U está dada por

$$F(x) = P(U \leq x) = x, x \in (0, 1),$$

y más generalmente, para $0 \leq y < x < 1$,

$$P(y < U \leq x) = x - y$$

En otras palabras, la probabilidad de que U caiga en un intervalo $(y, x]$ es simplemente la longitud del intervalo, $x - y$.

Independencia significa que, independientemente de los valores de los primeros n números aleatorios, $U_1, \ldots, U_n$, el valor del siguiente, U_{n+1}, todavía tiene la misma distribución uniforme sobre $(0, 1)$; de ninguna manera está afectado por esos valores anteriores.

Esto es análogo a lanzar secuencialmente una moneda (justa): independientemente de los primeros n lanzamientos, el siguiente todavía aterrizará en cara (H) o cruz (T) con probabilidad $1/2$.

La secuencia de variables aleatorias $(v.a.)$ $U_1, U_2, \ldots$ es un ejemplo de una secuencia independiente e idénticamente distribuida (iid). Aquí, la distribución idéntica es la uniforme sobre $(0, 1)$, que es un análogo continuo de probabilidades 'igualmente probables'.

En Python, por ejemplo, puedes obtener tal U de la siguiente manera: import random

U = random.random()

Una vez importado random, cada vez que uses el comando U = random.random()

recibes un nuevo número uniforme dentro de $[0, 1)$.

Resulta que una vez que tenemos acceso a tales números uniformes U, podemos usarlos para construir ('simular/generar') variables aleatorias de cualquier distribución deseada, construir procesos estocásticos como paseos aleatorios, cadenas de Markov, procesos de Poisson, procesos de renovación, movimiento browniano y muchos otros procesos.

Supongamos, por ejemplo, que queremos una variable aleatoria X que tenga una distribución exponencial a una tasa λ: La función de distribución acumulativa (CDF) está dada por

$$F(x) = P(X \leq x) = 1 - e^{-\lambda x}, x \geq 0.$$

Entonces simplemente define

$$X = -\frac{1}{\lambda} \ln(U)$$

donde $\ln(y)$ denota el logaritmo natural de $y > 0$. Prueba:

$$\begin{aligned} P(X \leq x) &= P\left(-\frac{1}{\lambda} \ln(U) \leq x\right) \\ &= P(\ln(U) \geq -\lambda x) \\ &= P\left(U \geq e^{-\lambda x}\right) \\ &= 1 - e^{-\lambda x} \end{aligned}$$

(Recuerda que $P(U \geq y) = 1 - y, y \in (0, 1)$.)

Pseudorandom numbers

Resulta que los números generados por una computadora no son realmente aleatorios ni independientes como dijimos, sino lo que se llaman números Pseudoaleatorios.

Esto significa que, para todos los fines prácticos, parecen ser aleatorios (e independientes) en el sentido de que pasarían varias pruebas estadísticas para verificar la propiedad aleatoria/independiente. Por lo tanto, podemos usarlos en nuestras simulaciones como si fueran verdaderamente aleatorios, y lo hacemos.

A continuación, discutiremos cómo la computadora genera estos números. Esto está profundamente relacionado con la 'criptografía' en la que uno quiere ocultar información importante (de un oponente/enemigo) en datos que "parecen" ser aleatorios.

Como un ejemplo para hacerte pensar: Supongamos que te entrego una secuencia de ceros y unos:

$$(0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1).$$

Te digo que lancé una moneda justa 25 veces donde $1 = \text{Cara}(H)$, y $0 = \text{Cruz}\ (T)$.

¿Cómo puedes verificar con certeza que estoy diciendo la verdad?

Pero si sigo entregándote secuencias de varias longitudes, entonces puedes realizar pruebas estadísticas que te ayudarían a decidir si las secuencias son consistentes con lanzamientos de moneda.

Generadores Congruenciales Lineales

El generador de números aleatorios más común, fácil de entender e implementar se llama Generador Congruencial Lineal (GCL) y se define mediante una recursión de la siguiente manera:

$$Z_{n+1} = (aZ_n + c) \mod m, \quad n \geq 0$$
$$U_n = \frac{Z_n}{m}$$

donde $0 < a < m, 0 \leq c < m$ son enteros constantes, y $\mod m$ significa módulo m que implica dividir por m y quedarse con el residuo. Por ejemplo, $6 \mod 4 = 2, 2 \mod 4 = 2, 7 \mod 4 = 3, 12 \mod 4 = 0$. Así, todos los Z_n caen entre 0 y $m-1$; los U_n están, por lo tanto, entre 0 y 1.

Se llama semilla a $0 \leq Z_0 < m$. Se elige m para que sea muy grande, usualmente de la forma $m = 2^{32}$ o $m = 2^{64}$ porque la arquitectura de tu computadora se basa en 32 o 64 bits por palabra; el cálculo del módulo simplemente implica la truncación por la computadora, por lo tanto, es inmediato.

Por ejemplo, si $m = 2^3 = 8$, entonces en binario, hay 8 números que representan $\{0, 1, 2, 3, 4, 5, 6, 7\}$ dados por una terna de $0s$ y $1s$, denotados por

$$(i_0, i_1, i_2) = i_0 2^0 + i_1 2^1 + i_2 2^2, \quad i_j \in \{0, 1\}, \quad j = 0, 1, 2.$$

Así, $(0, 0, 0) = 0, (1, 0, 0) = 2^0 = 1, (1, 1, 0) = 2^0 + 2^1 = 3, (0, 0, 1) = 2^2 = 4$ y $(1, 1, 1) = 2^0 + 2^1 + 2^2 = 7$, y así sucesivamente.

Nota cómo $7+1 = 8 = 0 \mod 8$ se calcula mediante truncamiento: $(1, 1, 1) + (1, 0, 0) = (0, 0, 0, 1) = 2^3$. El último componente se trunca, resultando en $(0, 0, 0) = 0$. $10 = (0, 1, 0, 1)$ se trunca a $(0, 1, 0) = 2$, y así sucesivamente.

Por ejemplo, si $m = 2^3 = 8$, entonces en binario, hay 8 números que representan $\{0, 1, 2, 3, 4, 5, 6, 7\}$ dados por una 3-tupla de 0s y 1s, denotados por

$$(i_0, i_1, i_2) = i_0 2^0 + i_1 2^1 + i_2 2^2, i_j \in \{0, 1\}, j = 0, 1, 2.$$

Así, $(0, 0, 0) = 0, (1, 0, 0) = 2^0 = 1, (1, 1, 0) = 2^0 + 2^1 = 3, (0, 0, 1) = 2^2 = 4$ y $(1, 1, 1) = 2^0 + 2^1 + 2^2 = 7$, y así sucesivamente.

Nótese cómo se calcula $7+1 = 8 = 0 \mod 8$ mediante truncamiento: $(1, 1, 1) + (1, 0, 0) = (0, 0, 0, 1) = 2^3$. El último componente se trunca dando como resultado $(0, 0, 0) = 0$. $10 = (0, 1, 0, 1)$ se trunca a $(0, 1, 0) = 2$, y así sucesivamente.

Aquí hay un ejemplo más típico:

$$Z_{n+1} = (1664525 \times Z_n + 1013904223) \bmod 2^{32}$$

Así $a = 1664525$ y $c = 1013904223$ y

$$m = 2^{32} = 4,294,967,296$$

más de 4.2 billones.

Los números a, c, m deben ser cuidadosamente escogidos para obtener un generador de números 'aleatorios' bueno, en particular querríamos que todos los valores c $0, 1, \ldots, c-1$ sean generados en cuyo caso decimos que el GLC tiene un periodo completo de longitud c. Tales generadores recorrerán cíclicamente los números una y otra vez.

Para ilustrar, considera

$$Z_{n+1} = (5Z_n + 1) \bmod 8, n \geq 0$$

con $Z_0 = 0$. Entonces

$$(Z_0, Z_1, \ldots, Z_7) = (0, 1, 6, 7, 4, 5, 2, 3)$$

y $Z_8 = 16 \bmod 8 = 0$, causando así que la secuencia se repita.

Si aumentamos c a $c = 16$,

$$Z_{n+1} = (5Z_n + 1) \bmod 16, n \geq 0,$$

con $Z_0 = 0$, entonces

$$(Z_0, Z_1, \ldots, Z_{15}) = (0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3)$$

y $Z_{16} = 16 \bmod 16 = 0$, causando así que la secuencia se repita.

Escoger buenos números a, c, m involucra el uso sofisticado de la teoría de números; números primos y tal, y ha sido extensamente investigado/estudiado por científicos de la computación y matemáticos por muchos años.

Nótese que los números generados son completamente deterministas: Si conoces los valores a, c, m, entonces una vez que conoces un valor (Z_0, digamos) los conoces todos. Pero si te entregan una larga secuencia de los U_n, ciertamente parecen aleatorios, y ese es el punto.

Resumen

Un **Generador Congruencial Lineal (GCL)** es un método para generar secuencias de números que aparentan ser aleatorios, basado en una relación de recurrencia lineal. Las principales características que definen a un GCL son:

Relación de Recurrencia Lineal

El núcleo de un GCL es la fórmula:

$$Z_{n+1} = (aZ_n + c) \mod m$$

donde:

- Z_{n+1} es el siguiente número pseudoaleatorio,
- Z_n es el número actual en la secuencia,
- a es el multiplicador, un entero positivo,
- c es el incremento, un entero no negativo,
- m es el módulo, un entero positivo mayor que a y c,
- mod representa la operación de módulo.

Elección de Parámetros

La elección de los parámetros a, c y m es crítica para las propiedades estadísticas y el período de la secuencia generada. El Teorema de Hull-Dobell proporciona condiciones para que un GCL tenga un período máximo.

Semilla

El valor inicial de la secuencia, Z_0, se conoce como la semilla y puede influir en la secuencia generada, aunque la elección de parámetros adecuados es más crucial.

Periodicidad

Los GCL tienen un período finito y, bajo ciertas condiciones, este período puede ser máximo y equivalente a m. Sin embargo, es posible que aparezcan ciclos más cortos con una mala elección de parámetros.

Tipos de GCL

Hay dos tipos principales de GCL:

- **GCL Multiplicativo:** Cuando $c = 0$, se simplifica a $Z_{n+1} = aZ_n \mod m$.
- **GCL Mixto:** Cuando $c \neq 0$, puede ofrecer un período más largo y mejores propiedades estadísticas, dependiendo de los parámetros.

Eficiencia y Portabilidad

Los GCL son algoritmos simples y eficientes, fáciles de implementar en diversos entornos computacionales.

Limitaciones

A pesar de sus ventajas, los GCL pueden no ser adecuados para aplicaciones que requieren una alta calidad de aleatoriedad, como en criptografía o simulaciones de alta precisión, debido a su linealidad inherente y limitaciones en las pruebas estadísticas.

Las ventajas de usar un GLC (Generador Lineal Congruencial):

1. Muy rápido de implementar.
2. No requiere almacenamiento de los números, solo del valor más reciente.
3. Replicación: Utilizando la misma semilla, puedes generar exactamente la misma secuencia una y otra vez, lo cual es extremadamente útil al comparar sistemas/modelos alternativos: Al usar los mismos números, estás reduciendo la variabilidad de las diferencias que serían causadas por el uso de números diferentes; cualquier diferencia en las comparaciones se debe así a la diferencia inherente en los modelos mismos.

Validación de Generadores Congruenciales Lineales

La validación de generadores congruenciales lineales (GCL) es esencial para asegurar que produzcan secuencias de números pseudoaleatorios con propiedades adecuadas de aleatoriedad. Los aspectos clave en la validación incluyen:

Período Completo

Un GCL ideal debería tener un período completo, generando todas las secuencias posibles dentro de su rango antes de repetir cualquier número. Según el Teorema de Hull-Dobell, un GCL tiene un período completo si:

1. c y m son coprimos.
2. $a - 1$ es divisible por todos los factores primos de m.
3. Si m es múltiplo de 4, $a - 1$ debe ser múltiplo de 4.

Distribución Uniforme

Se espera que los números generados por un GCL sigan una distribución uniforme en el intervalo $[0, 1)$ para los U_n o $[0, m-1]$ para los Z_n. La uniformidad se puede evaluar mediante pruebas estadísticas, como la prueba de chi-cuadrado.

Independencia Estadística

Los números generados deben ser estadísticamente independientes. Las pruebas de independencia, como la prueba de corridas arriba y abajo, pueden ayudar a evaluar esta propiedad.

Pruebas Empíricas

Las pruebas empíricas como el test de Kolmogorov-Smirnov, la prueba de poker y la prueba de huecos se utilizan para comparar la distribución de los números generados con la distribución esperada y detectar patrones no aleatorios.

Consideraciones Prácticas

Además de las pruebas teóricas, es crucial considerar aspectos prácticos como el rendimiento computacional y la facilidad de implementación del GCL. La elección de los parámetros a, c y m debe equilibrar la calidad de la aleatoriedad con las necesidades prácticas de la aplicación.

Al validar un GCL, se debe prestar atención tanto a los aspectos teóricos como prácticos para asegurar que el generador sea adecuado para su propósito previsto.

Introducción al generador de registros desfasados

El Generador de Registros Desfasados es un método para generar secuencias de números pseudoaleatorios, especialmente conocido por su aplicación en criptografía y simulaciones. Este método utiliza registros de desplazamiento y operaciones binarias para producir la secuencia deseada.

Conceptos Clave

- **Registros de Desplazamiento:** Son estructuras de datos que 'desplazan' sus contenidos, ya sean bits o números, en cada paso del tiempo.
- **Operación XOR:** Una operación lógica fundamental en muchos generadores de registros desfasados, aplicada a pares de bits para generar el siguiente bit en la secuencia.

Algoritmo

El algoritmo para el generador de registros desfasados se puede describir de la siguiente manera:

1. **Inicialización:** Comenzar con una semilla, que es una secuencia inicial de bits o números que llenan el registro.
2. **Desplazamiento y Generación:** En cada paso, se desplazan los contenidos del registro y se genera un nuevo bit o número. Este nuevo valor suele resultar de aplicar una operación, como XOR, a ciertos bits o números del registro.
3. **Repetición:** Este proceso se repite para generar la secuencia de números o bits pseudoaleatorios.

Implementación en R

A continuación, se proporciona una implementación básica del generador de registros desfasados en el lenguaje de programación R.

```
generadorRegistrosDesfasados <- function(semilla, longitud) {
  n <- length(semilla)
  secuencia <- numeric(longitud)
  registro <- semilla

  for (i in 1:longitud) {
    nuevoBit <- xor(registro[n], registro[n-1]) # Ejemplo simple con XOR
    secuencia[i] <- nuevoBit
    registro <- c(nuevoBit, registro[1:(n-1)])
  }

  return(secuencia)
}

# Ejemplo de uso
semilla <- c(1, 0, 1, 1) # Semilla inicial de 4 bits
longitudSecuencia <- 10  # Longitud de la secuencia deseada
generadorRegistrosDesfasados(semilla, longitudSecuencia)
```

Nota: Este es un ejemplo simplificado del generador de registros desfasados. En la práctica, la elección de los bits para las operaciones y la función utilizada pueden variar según el requerimiento específico.

Método de los Cuadrados Medios (John von Neumann, 1940s)

```
  cuadradosMedios <- function(semilla, n, digitos) {
    numeros <- numeric(n)  # Vector para almacenar los numeros generados
    actual <- semilla
    for (i in 1:n) {
      # Cuadrado de la semilla
      cuadrado <- as.numeric(substr(format(actual^2, scientific = FALSE, trim
          = TRUE),
      start = 1, stop = 2 * digitos))
      # Extraer digitos del medio
      numero <- as.numeric(substr(format(cuadrado, scientific = FALSE, trim =
          TRUE),
      start = (2 * digitos)/2 - digitos/2 + 1, stop = (2 * digitos)/2 +
         digitos/2))
      # Normalizar y almacenar
      numeros[i] <- numero / 10^digitos
      # Actualizar semilla
      actual <- numero
    }
    return(numeros)
  }
```

Método Congruencial Lineal (Derrick Lehmer, 1951)

```
MCL <- function(a, c, m, semilla, n) {
 numeros <- numeric(n)
 actual <- semilla
 for (i in 1:n) {
  actual <- (a * actual + c) %% m
  numeros[i] <- actual / m
 }
 return(numeros)
}
```

Generador de Medios Cuadrados (Middle-square method)

Este método es una variante directa del Método de los Cuadrados Medios, pero se destaca por ser una de las primeras propuestas y por los problemas que presenta en términos de ciclos y longitud de secuencia.

```
  generadorCuadradoMedio <- function(semilla, n, digitos) {
    numeros <- numeric(n)  # Vector para almacenar los nUmeros generados
```

```
  actual <- semilla
  for (i in 1:n) {
    # Cuadrado de la semilla
    cuadrado <- actual^2
    # Convertir a cadena y asegurarse de que tenga 2*digitos, rellenando
       con ceros si es necesario
    cuadradoStr <- sprintf("%0*s", 2*digitos, cuadrado)
    # Longitud deseada del nUmero (debe ser par)
    longitudDeseada <- 2*digitos
    # Asegurarse de que el cuadrado tenga la longitud deseada, rellenando
       con ceros
    if (nchar(cuadradoStr) < longitudDeseada) {
      cuadradoStr <- paste0(rep("0", longitudDeseada - nchar(cuadradoStr)),
          cuadradoStr)
    }
    # Extraer dIgitos del medio
    inicio <- (nchar(cuadradoStr) / 2) - (digitos / 2) + 1
    final <- (nchar(cuadradoStr) / 2) + (digitos / 2)
    medioStr <- substr(cuadradoStr, inicio, final)
    medio <- as.numeric(medioStr)
    # Verificar si se ha llegado a un ciclo o cero
    if (medio == 0 || medio == actual) {
      warning(paste("Se detecto un ciclo o cero en la iteracion", i))
      break
    }
    # Almacenar el nUmero y actualizar la semilla
    numeros[i] <- medio
    actual <- medio
  }
  return(numeros)
}

# Ejemplo de uso

semilla <- 5738  # Semilla inicial arbitraria
n <- 10  # Cantidad de nUmeros a generar
digitos <- 4  # Cantidad de digitos a conservar

# Generar numeros
numerosGenerados <- generadorCuadradoMedio(semilla, n, digitos)
print(numerosGenerados)
```

Método Congruencial Multiplicativo

Este es un caso especial del Método Congruencial Lineal donde el incremento c es cero.

```
MCM <- function(a, m, semilla, n) {
 numeros <- numeric(n)
 actual <- semilla
 for (i in 1:n) {
  actual <- (a * actual) %% m
  numeros[i] <- actual / m
 }
 return(numeros)
}
```

Introducción al Generador Congruencial Cuadrático

El generador congruencial cuadrático es un tipo de generador de números pseudoaleatorios no lineal que introduce un término cuadrático en la relación de recurrencia. La forma general de este generador es:

$$X_{n+1} = (aX_n^2 + bX_n + c) \mod m$$

donde a, b y c son coeficientes constantes, y m es el módulo.

0.1 Ventajas

- Mayor complejidad en la secuencia generada en comparación con los generadores lineales.
- Posibilidad de ciclos más largos si se eligen adecuadamente los parámetros.

0.2 Desafíos

- La elección de los parámetros a, b, c y m es crítica para asegurar la calidad de la secuencia pseudoaleatorias.
- Puede ser más computacionalmente intensivo que los métodos lineales debido a la operación cuadrática.

Test Serial en Generadores de Números Pseudoaleatorios

Un test serial evalúa la hipótesis de que cada subsecuencia de k elementos dentro de una secuencia generada por un GNPA es igualmente probable, como se esperaría en una secuencia verdaderamente aleatoria. Para una secuencia de números generados y un valor fijo k, el test considera todas las posibles subsecuencias de longitud k y calcula la frecuencia de cada una de ellas.

Dada una secuencia de números $X_1, X_2, ..., X_n$, generada por un GNPA, y un valor fijo k, el objetivo es examinar las subsecuencias de longitud k: $(X_1, X_2, ..., X_k), (X_2, X_3, ..., X_{k+1}), ..., (X_{n-k+1}, X_{n-k+2}, ..., X_n)$.

Para un espacio de muestreo de tamaño m, hay m^k subsecuencias posibles de longitud k. El test serial evalúa la distribución de estas m^k subsecuencias en la secuencia generada.

Hipótesis

La hipótesis nula H_0 del test serial es que la secuencia de números es aleatoria, lo que implica que cada una de las m^k subsecuencias posibles de longitud k es igualmente probable. La hipótesis alternativa H_1 es que la secuencia no es aleatoria.

Estadístico de prueba

El estadístico de prueba se basa en la diferencia entre las frecuencias observadas y esperadas de cada subsecuencia de longitud k. Si denotamos por O_i la frecuencia observada de la i-ésima subsecuencia y por $E = \frac{n-k+1}{m^k}$ la frecuencia esperada (bajo la hipótesis de aleatoriedad), el estadístico de prueba χ^2 se calcula como:

$$\chi^2 = \sum_{i=1}^{m^k} \frac{(O_i - E)^2}{E}$$

Bajo la hipótesis nula, este estadístico sigue una distribución χ^2 con $m^k - 1$ grados de libertad, ajustando por las restricciones en las frecuencias totales.

Decisión

Se rechaza la hipótesis nula de aleatoriedad si el valor calculado de χ^2 es mayor que el valor crítico de la distribución χ^2 con $m^k - 1$ grados de libertad, para un nivel de significancia α dado. Esto indicaría que la secuencia generada no pasa el test serial y, por lo tanto, podría no ser adecuadamente aleatoria para ciertas aplicaciones.

```r
# Test Serial en R para evaluar la uniformidad de pares de numeros
    consecutivos

testSerial <- function(numeros) {
  n <- length(numeros)

  # Crear pares consecutivos
  pares <- embed(numeros, 2)[, 2:1]

  # Contar la frecuencia de cada par
  frecuencias <- table(paste(pares[,1], pares[,2]))

  # Frecuencia esperada para cada par si son uniformemente distribuidos
  frecuenciaEsperada <- (n - 1) / length(frecuencias)
```

```
  # Calcular el estadistico chi-cuadrado
  chiCuadrado <- sum((frecuencias - frecuenciaEsperada)^2 /
      frecuenciaEsperada)

  # Calcular el p-valor
  pValor <- pchisq(chiCuadrado, df = length(frecuencias) - 1, lower.tail =
      FALSE)

  return(list(chiCuadrado = chiCuadrado, pValor = pValor))
}

# Ejemplo de uso
numeros <- runif(100) # Generar 100 numeros pseudoaleatorios uniformemente
    distribuidos
resultado <- testSerial(numeros)
print(resultado)
```

Test de Rachas en Generadores de Números Pseudoaleatorios

El test de rachas es una herramienta estadística utilizada para analizar patrones dentro de una secuencia de números generados por un Generador de Números Pseudoaleatorios (GNPA) para determinar si la secuencia muestra un nivel de aleatoriedad aceptable. Una 'racha' se define como una secuencia ininterrumpida de elementos similares (por ejemplo, secuencias crecientes o decrecientes).

Considérese una secuencia de números $X_1, X_2, ..., X_n$ generada por un GNPA. El test de rachas observa las transiciones entre números consecutivos para identificar cambios de crecimiento a decrecimiento o viceversa, incluyendo secuencias de números iguales si son pertinentes para el análisis.

Hipótesis

La hipótesis nula H_0 del test de rachas afirma que la secuencia es aleatoria, lo que significa que la probabilidad de cambiar de una racha creciente a una decreciente (o viceversa) es constante y no depende de las rachas anteriores. La hipótesis alternativa H_1 sugiere que la secuencia no es aleatoria, indicando dependencia entre los elementos de la secuencia.

Estadístico de prueba

Para aplicar el test, primero se cuenta el número total de rachas en la secuencia, denotado como R. Bajo la hipótesis de aleatoriedad, el número esperado de rachas $E(R)$ y su varianza $Var(R)$ se pueden aproximar para una secuencia larga mediante:

$$E(R) = \frac{2n - 1}{3}$$

$$Var(R) = \frac{16n - 29}{90}$$

El estadístico de prueba Z se calcula entonces como:

$$Z = \frac{R - E(R)}{\sqrt{Var(R)}}$$

Bajo la hipótesis nula, Z sigue aproximadamente una distribución normal estándar $N(0, 1)$ para secuencias suficientemente largas.

Decisión

Se rechaza la hipótesis nula H_0 si el valor absoluto de Z es mayor que el valor crítico $z_\alpha/2$ de la distribución normal estándar para un nivel de significancia α dado. Un valor de Z fuera de este rango sugiere que la secuencia no es aleatoria con respecto a la propiedad de independencia entre elementos consecutivos.

La implementación del Test de Rachas en R para una secuencia de números podría verse así:

```r
# Implementacion del Test de Rachas en R
testDeRachas <- function(numeros) {
  n <- length(numeros)
  rachas <- 1
  for (i in 2:n) {
    if (sign(numeros[i] - numeros[i-1]) != sign(numeros[i-1] - numeros[i-2]))
        {
      rachas <- rachas + 1
    }
  }
  E <- (2*n - 1) / 3
  Var <- (16*n - 29) / 90
  Z <- (rachas - E) / sqrt(Var)
  p_valor <- 2 * pnorm(-abs(Z))
  return(list(rachas = rachas, Z = Z, p_valor = p_valor))
}
```

Test de Kolmogorov-Smirnov en Generadores de Números Pseudoaleatorios

El test de Kolmogorov-Smirnov es una herramienta estadística que compara la distribución empírica de una secuencia de números generados por un GNPA con la distribución teórica esperada, que, en el caso de los GNPA, es típicamente la distribución uniforme.

Sea $X_1, X_2, ..., X_n$ una secuencia de números generados por un GNPA. Se define $F_n(x)$ como la función de distribución empírica de la muestra, que para cualquier número x se calcula como la proporción de elementos en la muestra que son menores o iguales a x. La función de distribución teórica, que se espera que siga la muestra bajo la hipótesis de perfecta aleatoriedad, se denota como $F(x)$.

Hipótesis

La hipótesis nula H_0 del test de Kolmogorov-Smirnov establece que la muestra sigue la distribución teórica esperada, es decir, $F_n(x) = F(x)$ para todo x. La hipótesis alternativa H_1 indica que la muestra no sigue la distribución teórica esperada.

Estadístico de Prueba

El estadístico de prueba D_n se define como la máxima diferencia absoluta entre la función de distribución empírica y la teórica:

$$D_n = \sup_x |F_n(x) - F(x)|$$

donde sup denota el supremo, es decir, la máxima diferencia observada.

Decisión

El test de Kolmogorov-Smirnov rechaza la hipótesis nula H_0 si el estadístico de prueba D_n es mayor que el valor crítico $D_{n,\alpha}$, que depende del tamaño de la muestra n y del nivel de significancia α escogido. Los valores críticos se pueden obtener de tablas precalculadas para el test de Kolmogorov-Smirnov.

Si se rechaza la hipótesis nula, se concluye que hay evidencia estadística suficiente para afirmar que la secuencia de números generados no sigue la distribución uniforme esperada y, por lo tanto, puede no ser adecuadamente aleatoria para algunas aplicaciones.