Tarea 5. Simulación Estocástica, introducción

Ricardo Chávez Cáliz

29 de noviembre de 2017

1. Inversa generalizada

Definir la cdf inversa generalizada F_X^- para una variable aleatoria X y demostrar que en el caso de variables aleatorias continuas esta coincide con la inversa usual. Demostrar además que en general para simular de X podemos simular $u \sim U(0,1)$ y $F_X^-(u)$ se distribuye como X.

Definición 1.1. *Sea F una función no decreciente en $\mathbb{R}$, la **inversa generalizada** de F, F^- se define como la función dada por*

$$F^-(u) = inf\{x : F(x) \geq u\}$$

En el caso en que F es una función de distribución para una variable aleatoria continua, entonces se tiene que F_X^- continua y no decreciente. Por el teorema del calor medio $F_X^-(u)$ es inyectiva y por lo tanto el ínfimo es único.

Teorema 1. *Si $U \sim \mathcal{U}_{[0,1]}$, entonces la variable aleatoria $F^-(U)$ tiene distribución F. Ver 1*

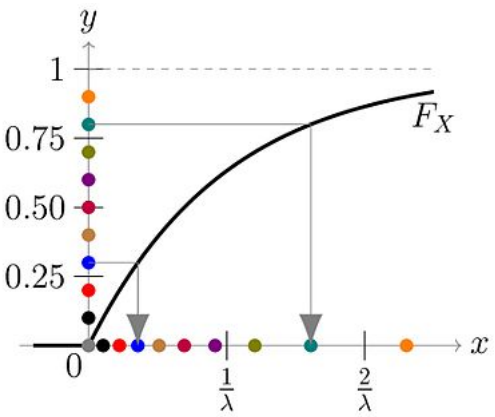

Figura 1: Se generan números aleatorios y_i generados con distribución uniforme entre 0 y 1. Los cuales se muestran como puntos de colores en el eje y. Cada punto es mapeado de acuerdo con $x = F^-(y)$, lo cual es mostrado con las dos flechas puestas en la ilustración

Demostración. Para toda $u \in [0,1]$ y para toda $x \in F^{-}([0,1])$, la inversa generalizada satisface

$$F(F^{-}(u)) \geq u \text{ y } F^{-}(F(x)) \leq x$$

.

Por lo tanto

$$\{(u,x) : F^{-}(u) \leq x\} = \{(u,x) : F(x) \geq u\} \text{ y } \mathbb{P}(F^{-}(U) \leq x) = \mathbb{P}(U \leq F(x)) = F(X)$$

□

2. Variables aleatorias uniformes

2. Implementar el siguiente algoritmo para simular variables aleatorias uniformes:

$$x_i = 107374182x_{i-1} + 104420x_{i-5} \pmod{2^{31} - 1}$$

regresa x_i y recorrer el estado, esto es $x_{j-1} = x_j; j = 1,2,3,4,5$; ¿parecen U(0,1)?

Se implementó el algoritmo recién mencionado el cual puede ser encontrado en el código adjunto de `Python` en la función `uniformeMCL`, MCL hace referencia a "Método de congruencias lineales". Se tomó como semilla los digitos de en medio del valor del tiempo al cuadrado en el que se ejecuta el algoritmo.

En 2 se muestran los histogramas normalizados de muestras de tamaño 10,000 de números pseudo-aleatorios generados con el algoritmo propuesto y se hizo uso de la librería de `numpy` para comparar. En cada uno se graficó la función de densidad de la distribución uniforme.

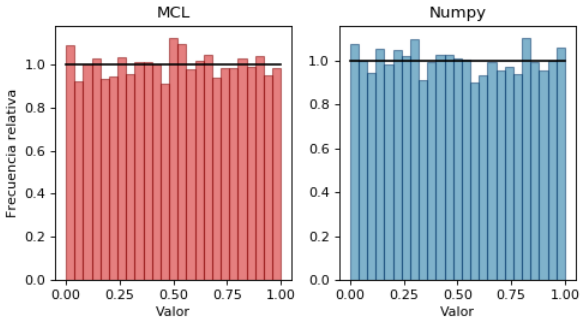

Figura 2: Histogramas normalizados y funciones de densidad

En la figura 2 se puede notar que en ambos casos el histograma normalizado se asemeja a la función de densidad apropiada.

Se aplicó es test de Kolmogrov-Smirnov de la librería `scipy.stats` el cual es llamado en el código con `kstest`. Los resultados se encuentran en 1, junto los tiempos de ejecución de cada algoritmo.

Podemos ver que algoritmo implementado no ofrece tan buenos resultados como los obtenidos por Numpy pero son *suficientemente buenos*, en base a la regla"de un p-valor *suficientemente bueno*, se puede decir que no existe evidencia muestral suficiente para rechazar la hipótesis de que los datos generados provienen de una distribución uniforme.

Sampler	estadístico (TKS)	p-valor (TKS)	tiempo de ejecución
Conguencias lineales	0.00653	0.48634	0.04548
Numpy	0.00968	0.30472	0.00065

Tabla 1: Tabla comparativa de resultados para muestras uniformes de tamaño 10,000 con distintos generadores

Para verificar la no dependencia temporal se graficó las parejas de punto $(X_t, Xt+1)$, para muestras de tamaño 1000, los resultados fueron favorables y pueden encontrarse en 3

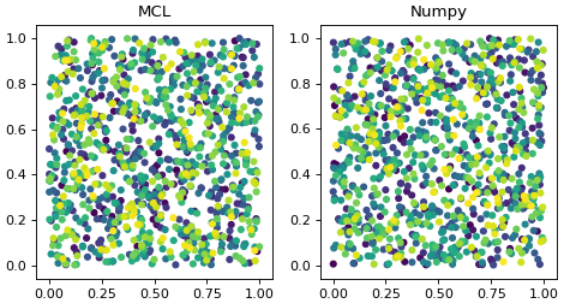

Figura 3: Gráficas de parejs (X_t, X_{t+1}) para muestras de tamaño 1000 generadas con MCL y con Numpy

3. Números aleatorios en scipy.stats

3. ¿Cuál es el algoritmo que usa scipy.stats.uniform para generar números aleatorios? ¿Cómo se pone la semilla? ¿y en R?

La librería de `scipy.stats` no tiene un generador de pseudonúmeros aleatorios, estos son obtenidos en una de las siguientes 3 maneras:

1. Directamente de numpy.random. El módulo *uniform* de la libreria `scipy.stats` utiliza `Numpy`, mtrand es la función correspondiente en stats (mtrand es un alias para numpy.random), la diferencia reside en que scipy.stats es un poco más costoso al hacer verificaciones de errores y al hacer la interfaz más flexible. La diferencia en velocidad debería de ser mínima mientras uniform.rvs no se ejecute en un ciclo para cada muestra. Es mejor obtener todas las muestras de una sola vez.

   En `scipy/numpy` se usa el algoritmo **Mersenne-Twister** PRNG (pseudorandom number generator) para generar los números aleatorios básicos. Los números aleatorios para distribuciones en numpy.random están en `cython/pyrex` y son bastante rápidos.

2. Trasformación de otro números aleatorios en numpý.random, también bastante rápido debido a que trabaja con arreglos de números enteros.

3. Genéricos: el único genrador de números aleatorios es usando la inversa de cdf para transformar variables de números uniformes. Esto es relativamente rápido, si es que hay una una expresión para la ppf, pero puede ser muy lento si la ppf se debe calcular indirectamente. por ejemplo si la única pdf está definida, entonces la cdf es obtenida a través de integración numérica y la ppf es obtenida a través de un solucionador de ecuaciones. Por lo que algunas distribuciones son muy lentas.

La **semilla** para inicializar el PRNG puede ser un entero entre 0 y $2^{32} - 1$ ó incluso un arreglo de dichos enteros o ninguno(default). En el caso por default `RandomState` tratará de leer datos de /dev/urandom (o el análogo en Windows) si está disponible, de no ser posible proveerá una semilla proveniente del reloj.

4. V.a. discretas en scipy

4. ¿En scipy que funciones hay para simular una variable aleatoria genérica discreta? ¿tienen preproceso?

Sí, esto se hace por medio de la función `"scipy.stats.rv_discrete"`. La cual es una clase base para construir clases específicas de distribuciones e instancias para variables aleatorias. También puede ser usada para construir distribuciones arbitrarias definidas por una lista de los puntos en el soporte y sus correspondientes probabilidades. Cabe mencionar que también se tiene la clase `rv_continuous`, cuyas principales diferencias son:

1. El soporte de la distribución en `rv_discrete` son los enteros
2. En lugar de una función de densidad pdf (y la correspondiente privada pdf), esta clase define la función de masa de probabilidad pmf (y la correspondiente privada pmf)
3. No está definido el parámetro de escala

Para ver un ejemplo de cómo crear una distribución discreta ver 4

```
>>> from scipy.stats import rv_discrete
>>> class poisson_gen(rv_discrete):
...     "Poisson distribution"
...     def _pmf(self, k, mu):
...         return exp(-mu) * mu**k / factorial(k)
```

para luego crear la instancia:

```
>>> poisson = poisson_gen(name="poisson")
```

Figura 4: Ejemplo de generación de v.a. poisson con `rv_discrete`

5. ARS

Implementar el algoritmo **Adaptive Rejection Sampling** y simular de una Gama(2, 1) 10,000 muestras. ¿cuándo es conveniente dejar de adaptar la envolvente?

Se implementó el siguiente algoritmo de ARS ver [1]

Algoritmo ARS

1. Inicializar n y S_n
2. Generar $X \sim g_n(X)$, $U \sim \mathcal{U}_{[0,1]}$
3. Si $U \leq \underline{f}_n(X)/\bar{\omega}_n \cdot g_n(X)$, aceptamos X;de lo contrario,
   si $U \leq f(X)/\bar{\omega}_n \cdot g_n(X)$, aceptamos X y actualizamos S_n a $S_{n+1} = S_n \cup \{X\}$

Algoritmo suplementario para ARS

1. Seleccionar intervalo $[x_i, x_{i+1}]$ con probabilidad

$$e^{\beta_i} \cdot \frac{e^{\alpha_i x_{i+1}} - e^{\alpha_i x_i}}{\bar{\omega}_n \alpha_i}$$

2. Generar $U \sim \mathcal{U}_{[0,1]}$ y tomar

$$X = \alpha_i^{-1} \cdot log\left[e^{\alpha_i x_i} + U\left(e^{\alpha_i \cdot x_{i+1}} - e^{\alpha_i x_i}\right)\right]$$

La implementación de la primer parte del algoritmo puede ser encontrada en el código de Python adjunto en la función `ARS`, la segunda parte se puede encontrar en la función `simulaG`.

Para comparar se usó que en este caso podemos obtener la v.a. simulando 2 variables aleatorias exponenciales, las cuales son muy fáciles de simular haciendo uso del Teorema 1 aquí enunciado y ya que en este caso la función inversa de $exp(\lambda)$ es $(-1/\lambda) \cdot log(1-y)$.

En 5 se muestran los histogramas normalizados de muestras de tamaño 10,000 de números pseudo-aleatorios generados con la suma de la simulación de dos v.a. exponenciales, de la librería de `numpy` y del con el algoritmo ARS, para comparar los resultados. En cada uno se graficó la función de densidad de la distribución gama.

En la figura 2 se puede notar que todos los casos el histograma normalizado se asemeja a la función de densidad apropiada.

Se aplicó es test de Kolmogrov-Smirnov de la librería `scipy.stats`. Los resultados se encuentran en 2, junto los tiempos de ejecución de cada algoritmo.

[1] Robert, Casella - Monte Carlo Statistical Methods Pag(56 y 71)

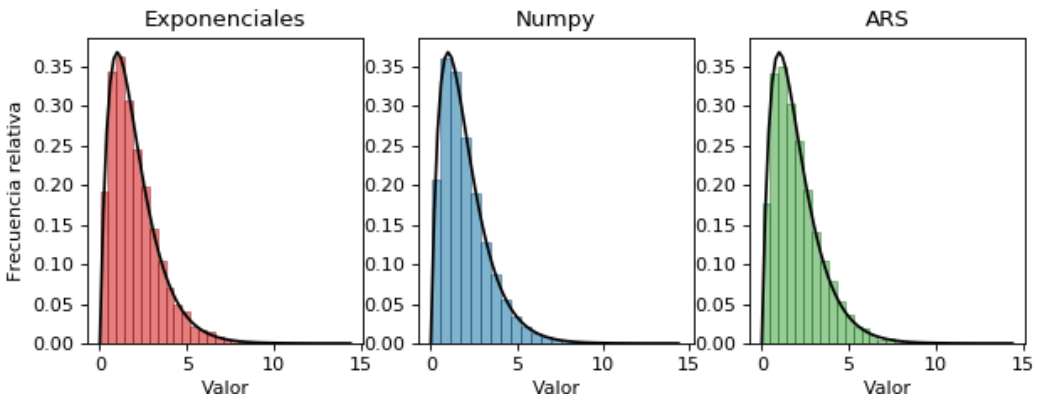

Figura 5: Histogramas normalizados y funciones de densidad

Sampler	estadístico (TKS)	p-valor (TKS)	tiempo de ejecución
Exponenciales	0.00636	0.81287	0.00529
Numpy	0.00793	0.55465	0.00634
ARS	0.007832	0.57155	3.73731

Tabla 2: Tabla comparativa de resultados para muestras $Gama(2, 1)$ de tamaño 10,000 con distintos generadores

Podemos ver que algoritmo implementado ofrece muy buenos resultados como los obtenidos por Numpy o los de la suma de exponenciales ya que el p-valor es *suficientemente pequeño*; en cada caso se puede decir que no existe evidencia muestral suficiente para rechazar la hipótesis de que los datos generados provienen de una distribución Gama. Como es de suponerse el tiempo de ejecución del algoritmo ARS es mucho mayor pero considerando que puede ser aplicado cuando los parámetros no son tan fáciles como en este caso podemos decir que es un algoritmo de gran utilidad. Para comparar los tiempos de ejecución en un caso no entero se midió el tiempo que tardaba ARS y Numpy para generar muestras de tamaños entre 0 y 10,000 (sólo múltiplos de 500) y se graficó su resultado en escala logartimica ver 6. La diferencia parece indicar una diferencia por una constante entre los tiempos de ejecución, lo cual es bueno.

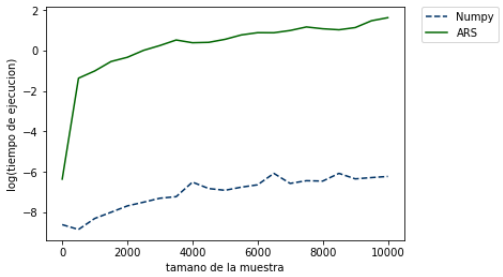

Figura 6: Tiempos de ejecución de Numpy y ARS en escala logarítmica