

PRÁCTICA 2: Computación Numérica. Curso 2021-2022

Inferencia Bayesiana

Guillermo Hoyo Bravo, Adrián Rubio Pintado

In [1]:

```
import scipy.stats
import matplotlib.pyplot as plt
import numpy as np
# for latex equations
from IPython.display import Math, Latex
# for inline plots in jupyter
if True:
    %matplotlib inline
else:
    %matplotlib notebook
from scipy import integrate
```

In [2]:

```
np.random.seed(124) # semilla para reproducibilidad
```

In [18]:

```
#Versión modificada de el profesor
def resultados (x, y, n, alpha) :
    z_alpha = scipy.stats.norm.ppf(1 - (alpha / 2) )
    size = np.arange (start = 1, stop = n+1, step=1)

    # media muestral
    media = np.cumsum (y) / size

    # Varianza muestral, s2
    media2 = media * media
    y2 = y * y
    s2 = np.cumsum (y2) / size - media2

    # varianza estimador
    sigma2 = s2 / size

    #intervalo de confianza
    sigma = np.sqrt(sigma2)
    ci_inf = media - z_alpha * sigma
    ci_sup = media + z_alpha * sigma

    resul = {'size':size, 'media' : media, 's2': s2, 'sigma2': sigma2,
             'ci_inf': ci_inf, 'ci_sup':ci_sup, 'inter_conf':z_alpha * sigma}

    return resul
```

EJERCICIO 1

Estimar la probabilidad de que un paciente ingresado en planta acabe en la UCI. Para realizar este problema mediante inferencia Bayesiana debemos tener en cuenta: *La probabilidad de que un paciente ingresado en planta acabe en la UCI será nuestro parámetro θ . Como hemos visto, en el enfoque bayesiano debemos considerar que θ es una variable aleatoria.* Debemos conocer la distribución de la v.a θ **antes** de realizar el experimento, es decir su *densidad de probabilidad a*

priori $\pi(\theta)$. Supongamos, por ejemplo, que $\pi(\theta) = B(\theta|\alpha = 5, \beta = 10)$. Debemos incorpora el resultado del experimento. Considerad el siguiente experimento: se contabiliza cuantos de los pacientes que hay en planta (n) deben ingresar en la UCI (k). Se observa que de los $n = 20$ pacientes en planta $k = 1$ ingresan en la UCI. Debemos calcular la función de verosimilitud $\pi(x|\theta)$ del resultado del experimento. La función de verosimilitud será proporcional a la probabilidad de observar el resultado del experimento en función del valor de θ):

$$\pi(x|\theta) \propto \theta^k (1-\theta)^{n-k}$$

donde, en este caso, habría que substituir $n = 20$ y $k = 1$. Notad como en la ecuación anterior $\pi(x|\theta)$ **no** está normalizada. Debemos elegir un predictor adecuado a nuestro problema. Como predictor de θ la probabilidad de que un paciente de la planta acabe en la UCI* utilizaremos su valor esperado $E[\theta]$. Nos preguntarnos entonces: 1. ¿Cuál es el valor de $E[\theta]$ **antes** de realizar el experimento?

1 ¿Cuál es el valor de $E[\theta]$ antes de realizar el experimento?

Al ser la distribución a priori $\pi(\theta)$ la distribución Beta, su valor esperado será

$$\mu_{\text{prior}} = E[\theta]_{\text{prior}} = \int_{-\infty}^{\infty} \theta B(\theta|\alpha, \beta) d\theta = \frac{\alpha}{\alpha + \beta}$$

Dado que tenemos $\alpha = 5$ y $\beta = 10$, tenemos que:

$$E[\theta]_{\text{prior}} = 5/(5 + 10) = 0.3333333333\text{periodico}$$

Graficamos la función de distribución a priori beta con los parámetros dados

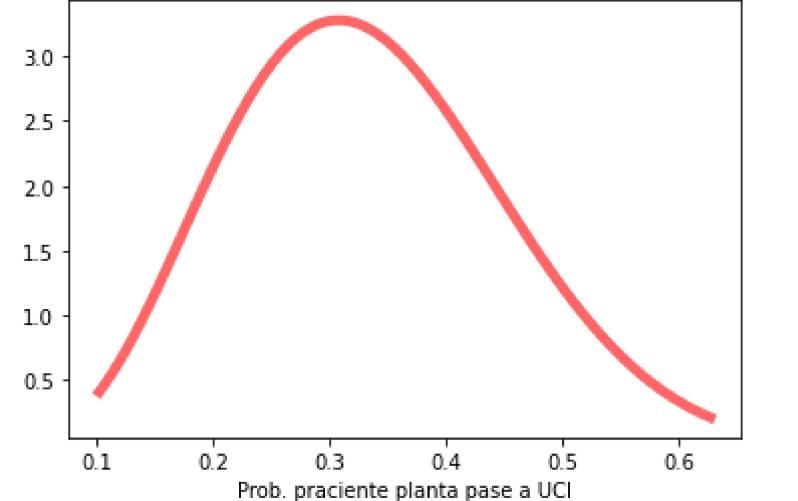
In [90]:

```
a = 5
b = 10

beta = scipy.stats.beta(a,b)
fig, ax = plt.subplots(1, 1)
x = np.linspace(beta.ppf(0.01), beta.ppf(0.99), 100)
ax.plot(x, scipy.stats.beta.pdf(x, a, b), 'r-', lw=5, alpha=0.6, label='prior \pi(\theta)')
ax.set_xlabel('Prob. paciente planta pase a UCI')
plt.title('Distribución a priorio de theta: Prob. paciente planta pase a UCI')
```

Out[90]:

Text(0.5, 1.0, 'Distribución a priorio de theta: Prob. paciente planta pase a UCI')



2. ¿Cuál es el valor de $E[\theta]$ después de observar el resultado del experimento?

Utilizando el teorema de Bayes, tras re-ordenar términos se obtiene:

$$\begin{aligned}
 E[\theta]_{\text{posterior}} &= \int_{-\infty}^{\infty} \theta \pi(\theta|x) d\theta \\
 &= \int_{-\infty}^{\infty} \theta \frac{\pi(x|\theta) \pi(\theta)}{\int_{-\infty}^{\infty} \pi(x|\theta) \pi(\theta) d\theta} d\theta = \frac{\int_{-\infty}^{\infty} \theta \pi(x|\theta) \pi(\theta) d\theta}{\int_{-\infty}^{\infty} \pi(x|\theta) \pi(\theta) d\theta} \\
 &= \frac{\int_{-\infty}^{\infty} \theta \theta^k (1-\theta)^{n-k} \pi(\theta) d\theta}{\int_{-\infty}^{\infty} \pi(x|\theta) \pi(\theta) d\theta} = \frac{\int_0^1 \theta^{k+1} (1-\theta)^{n-k} B(\theta|\alpha=5, \beta=10) d\theta}{\int_0^1 \theta^k (1-\theta)^{n-k} B(\theta|\alpha=5, \beta=10) d\theta}
 \end{aligned}$$

donde se han sustituído los valores de $\pi(\theta)$ y de la verosimilitud $\pi(x|\theta)$.

Para calcular dicho cociente, vamos a estimar el numerador y el denominador mediante integración de Monte Carlo. Para ello vamos a emplear el método de la función de importancia.

In [35]:

```

a = 5
b = 10

n_simul = 100000 #number of simulations

#Utilizamos muestreo por importancia

g Numerador = lambda x : (x**(k+1)) * ((1 - x)**(n-k))
g Denominador = lambda x : (x**k) * ((1 - x)**(n-k))

g Numerador_Final = lambda x : g Numerador(x) * scipy.stats.beta.pdf(x,a,b)
g Denominador_Final = lambda x : g Denominador(x) * scipy.stats.beta.pdf(x,a,b)

# Cociente g(x) / f(x)
def h(x, g, f, a, b):
    return g(x) / f(x, a, b)

def verosimilitud(t,k,n):
    #print("t",t , "k",k , "n",n )
    #print("verosimilitud", (t**k)*((1-t)**(n-k)))
    return (t**k)*((1-t)**(n-k))

```

Definimos una función para estimar por MonteCarlo la función de probabilidad a posteriori:

In [136...]

```

def get_posteriori_simulacion_montecarlo(a,b, n,k,nsimul,alpha_ic):

    #Utilizamos simulación de Monte Carlo mediante función de importancia
    x =scipy.stats.beta.rvs(a, b, size=n_simul)
    y Numerador = h(x = x, g = g Numerador_Final, f = scipy.stats.beta.pdf, a =
    y Denominador = h(x = x, g = g Denominador_Final, f = scipy.stats.beta.pdf,

    e_posteriori_20_1 = np.mean(y Numerador)/np.mean(y Denominador)
    #print('Expected probability a posteriori', e_posteriori_20_1 )

    # Evolución de E[theta] en función del número de simulaciones
    y Numerador_Estimador= np.cumsum (y Numerador) / n_simul
    y Denominador_Estimado= np.cumsum (y Denominador) / n_simul
    y_Final = y Numerador_Estimador/y Denominador_Estimado
    #r_norm = resultados(x=x, y=y_Final, n=n_simul, alpha=0.05)
    r_norm = resultados(x=x, y=y_Final, n=n_simul, alpha=alpha_ic)

```

```

print('Expected probability a posteriori', y_final[-1] , ' +- ', r_norm['int
print('Confidence interval α =', alpha_ic)
print('Maximum likelihood:', (k/n))
print('Experimento:')
print('\t Total pacientes:', n, 'UCI: ',k)

#fig, ax = plt.subplots(1,1, figsize=(10, 3))
fig, ax = plt.subplots()

desde = 1000

#ax.plot (r_norm['size'][desde:], r_norm['media'][desde:], lw=1.0, label='Exp
ax.plot (r_norm['size'][desde:], r_norm['media'][desde:], c='red', linestyle=
ax.plot (r_norm['size'][desde:], r_norm['ci_inf'][desde:], c='blue', alpha=0
ax.plot (r_norm['size'][desde:], r_norm['ci_sup'][desde:], c='blue', alpha=0
ax.fill_between(r_norm['size'][desde:],r_norm['ci_inf'][desde:],r_norm['ci_s
#ax.hlines(1, 0, n, linestyles='dashed', lw=0.8)
ax.set_xlabel('Montecarlo sample size')
ax.set_ylabel('Expected probability a posteriori')
ax.legend(loc='best', frameon=False)
ax.grid(lw=0.2)
ax.set_title('Estimación de probabilidad de theta a posteriori por Monte Car
#ax.set_ylim(0, 0.4)
plt.show()

fig, ax = plt.subplots()
desde = int(nsimal/2)
ax.plot (r_norm['size'][desde:], r_norm['media'][desde:], c='red', linestyle=
ax.plot (r_norm['size'][desde:], r_norm['ci_inf'][desde:], c='blue', alpha=0
ax.plot (r_norm['size'][desde:], r_norm['ci_sup'][desde:], c='blue', alpha=0
ax.fill_between(r_norm['size'][desde:],r_norm['ci_inf'][desde:],r_norm['ci_s
#ax.hlines(1, 0, n, linestyles='dashed', lw=0.8)

ax.set_xlabel('Montecarlo sample size')
ax.set_ylabel('Expected probability a posteriori')
ax.legend(loc='best', frameon=False)
ax.grid(lw=0.2)
ax.set_title('Estimación de probabilidad de theta a posteriori por Monte Car
#ax.set_ylim(0, 0.4)
plt.show()

```

In [137...]

```

n = 20
k = 1
nsimul = 100000
alpha_ic = 0.22
get_posteriori_simulacion_montecarlo(a,b,n,k,nsimul,alpha_ic)

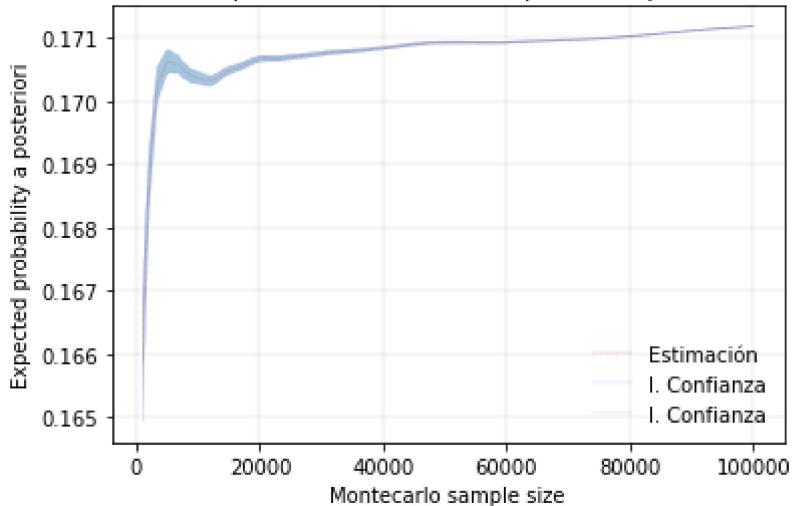
```

```

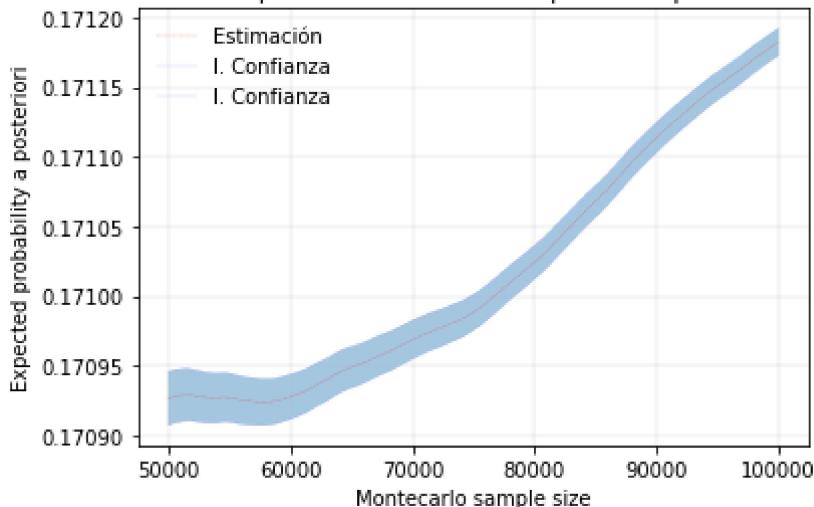
Expected probability a posteriori 0.17174669634818532 +- 9.756707095564671e-06
Confidence interval α = 0.22
Maximum likelihood: 0.05
Experimento:
    Total pacientes: 20 UCI:  1

```

Estimación de probabilidad de theta a posteriori por Monte Carlo



Estimación de probabilidad de theta a posteriori por Monte Carlo



In [138...]

```
n = 20*5
k = 1*5
n_simul = 100000
alpha_ic = 0.22
get_posteriori_simulacion_montecarlo(a,b,n,k,n_simul,alpha_ic)
```

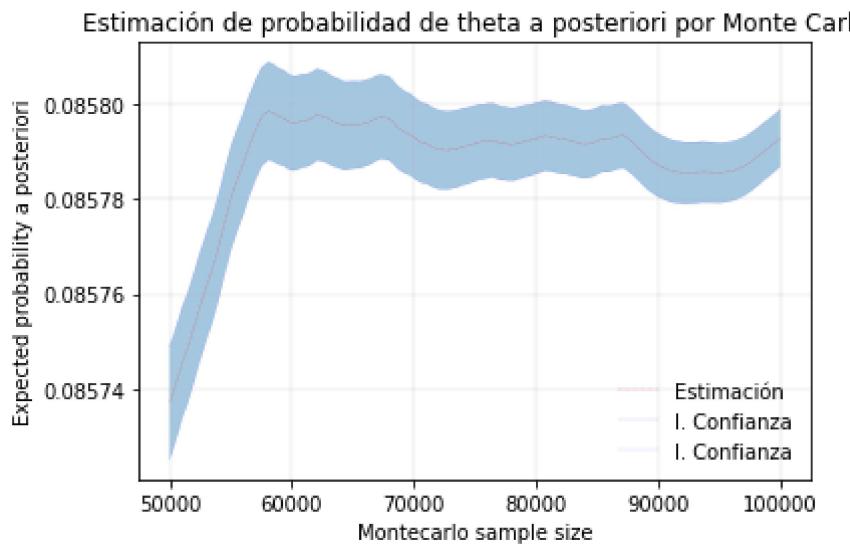
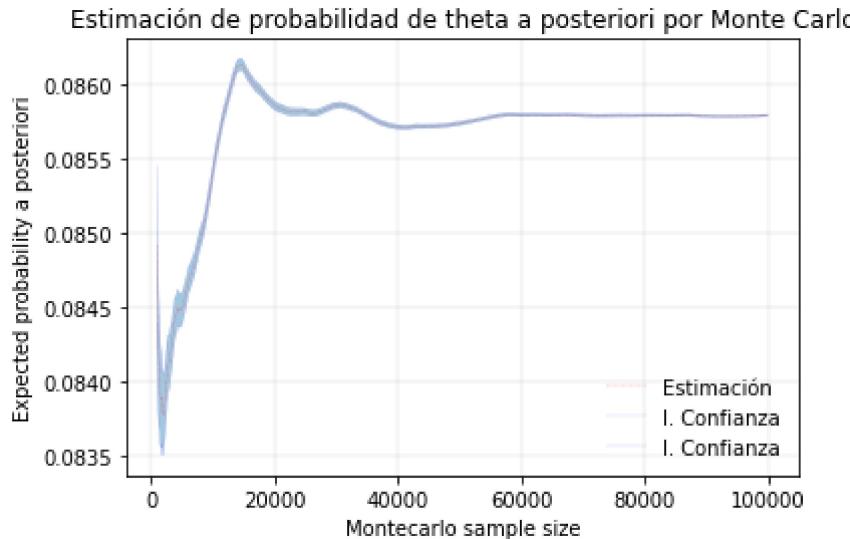
Expected probability a posteriori 0.08598439361655257 +- 5.964026374634053e-06

Confidence interval $\alpha = 0.22$

Maximum likelihood: 0.05

Experimento:

Total pacientes: 100 UCI: 5



Vemos como tras conocer los resultados del experimento, actualizamos la información de la que disponemos, y el valor esperado de nuestra variable aleatoria theta cambia. Además, gracias al método de Monte Carlo, hemos podido calcular el valor esperado de θ a posteriori sin necesidad de disponer de una muestra de la distribución a posteriori $\pi(\theta|x)$

Cuestiones:

¿Por que se debe utilizar la misma muestra de θ en el numerador y denominador para estimar μ posterior ?

Porque si nos fijamos hemos utilizado Bayes(ecuación 1) para evitar utilizar directamente la función de densidad a posteriori $\pi(\theta|x)$. Ahí utilizamos la función de máxima verosimilitud. De ahí nos sale acabar teniendo 2 integrales en la ecuación 10, pero ellas utilizan la misma función de verosimilitud, ya que vienen de la misma función de densidad a posteriori. Por tanto si tomamos muestras, numerador y denominador deben de utilizar la misma.

¿Por que a la hora de hacer la simulación de M.C hemos podido utilizar una función sin normalizar proporcional a $\pi(x|\theta)$ eq. (5) y no $\pi(x|\theta)$?

Porque la distribución a priori $\pi(\theta)$ es una distribución Beta($\theta|\alpha,\beta$), y la función de verosimilitud vemos que es proporcional a la probabilidad de observar el resultado del experimento en función del valor de θ) Dado que $\pi(x|\theta)$ está dada por (5), la distribución a posteriori $\pi(\theta|x)$

tiene una expresión cerrada. En este caso $\pi(\theta|x)$ es también una distribución Beta (se dice que ambas son conjugadas a priori).

Siempre debes validar tu simulación:

En este caso puedes validar tu simulación frente al resultado teórico conocido ¿Se comporta correctamente tu estimación cuando aumenta el tamaño del experimento?

Definimos las funciones que nos dan la probabilidad a priori/posteriori de forma teórica.

In [9]:

```
def prob(a,b):
    return a/(a+b)
```

Dado que la probabilidad a priori solo depende de alpha y de beta, no de n ni de k, validamos el resultado para los 2 casos de estudio:

In [10]:

```
print('Probabilidad a priori(Teórica): ',prob(a,b))
```

Probabilidad a priori(Teórica): 0.3333333333333333

Validamos ahora la probabilidad a posteriori, la cuál sí depende de n y de k:

In [11]:

```
def posteriori_parameters(a, b, n, k):
    alphap = a + k
    betap = b + n - k

    return alphap, betap
```

In [12]:

```
#Caso n=20, k= 1
n=20
k= 1

aa, bb = posteriori_parameters(a, b, n, k)
post = prob(aa, bb)
print('Experimento:')
print('\t Total pacientes:', n, 'UCI: ',k)
print('Probabilidad a posteriori(Estimada)', e_posteriori_20_1)

print('\nProbabilidad a posteriori(Teórica): ',post)
print('Error cometido(Teorico vs estimacion): ', abs(post - e_posteriori_20_1))

i_numeric = integrate.quad(g Numerador_final,0,1)[0]
i_numeric2 = integrate.quad(g Denominador_final,0,1)[0]

print('\nProbabilidad a posteriori(Teórica por integración numérica)', i_numeric/i_n
print('Error cometido(Integración vs estimacion): ', abs(post - i_numeric/i_numeric2)
```

Experimento:

Total pacientes: 20 UCI: 1
Probabilidad a posteriori(Estimada) 0.17165447893335062

Probabilidad a posteriori(Teórica): 0.17142857142857143
Error cometido(Teorico vs estimacion): 0.00022590750477918808

Probabilidad a posteriori(Teórica por integración numérica) 0.17142857142857143
Error cometido(Integración vs estimacion): 0.0

In [13]:

```
#Caso n=100, k = 5
```

```

n=100
k = 5
aa, bb = posteriori_parameters(a, b, n, k)
post = prob(aa, bb)
print('Experimento:')
print('\t Total pacientes:', n, 'UCI: ',k)
print('Probabilidad a posteriori(Estimada)', e_posteriori_100_5)

print('\nProbabilidad a posteriori(Teórica): ',post)
print('Error cometido(Teorico vs estimacion): ', abs(post - e_posteriori_20_1))

i_numeric = integrate.quad(g Numerador_final,0,1)[0]
i_numeric2 = integrate.quad(g Denominador_final,0,1)[0]

print('\nProbabilidad a posteriori(Teórica por integración numérica)', i_numeric/i_n
print('Error cometido(Integración vs estimacion): ', abs(post - i_numeric/i_numeric2

```

Experimento:

Total pacientes: 100 UCI: 5
 Probabilidad a posteriori(Estimada) 0.08681814779469985

Probabilidad a posteriori(Teórica): 0.08695652173913043
 Error cometido(Teorico vs estimacion): 0.08469795719422019

Probabilidad a posteriori(Teórica por integración numérica) 0.08695645045706057
 Error cometido(Integración vs estimacion): 7.128206985784757e-08

Hemos validado nuestras simulaciones para los dos casos de estudio, observando como el valor esperado es el mismo que el teórico, con un error muy pequeño.

Hemos empleado tanto el resultado teórico que teníamos como integración numérica de scipy para validar nuestro resultado. Por lo que sí, podemos afirmar que se comporta correctamente la estimación cuando aumenta el tamaño del experimento, es decir, hemos validado el correcto comportamiento de nuestro modelo.

¿Cómo de fiable es la estimación del error que hemos hecho? Discute que valor de $z\alpha/2$ es apropiado.y algo más

El error de nuestro estimador S_n dependerá del tamaño de la muestra n y de la varianza de la v.a $h(x) = g(x)/f(x)$ para cada integral. Suponemos que Z es el cociente de dos v.a $Z=X/Y$ (integral estimada en el numerador y la estimada también en el denominador) e interpretamos X e Y como una medida con un error asociado ($x \pm \delta_x$, $y \pm \delta_y$).

Es decir, dichas medidas las tomaremos como las medias muestrales (salida de nuestras estimaciones Monte Carlo) junto con su intervalo de confianza para un α dado.

Entonces el error máximo sería en el peor de los casos cuando ambos errores se suman al ser un cociente. Es decir, tendríamos:

$$\delta z/z = \delta x/x + \delta y/y$$

Definimos una función para calcularlo, por simplificación solo haremos el caso de estudio con $n=20, k=1$:

In [66]:

```
def get_posteriori_propagacion_error_maxima(a,b, n,k,nsimul,alpha_ic):
```

```
#Utilizamos simulación de Monte Carlo mediante función de importancia
x =scipy.stats.beta.rvs(a, b, size=n_simul)
```

```

y Numerador = h(x = x, g = g Numerador_Final, f = scipy.stats.bta.pdf, a =
y Denominador = h(x = x, g = g Denominador_Final, f = scipy.stats.bta.pdf, b = b)

# Evolución de E[theta] en función del número de simulaciones
y Numerador_Estimador = np.cumsum(y Numerador) / n_Simul
y Denominador_Estimado = np.cumsum(y Denominador) / n_Simul
y_Final = y Numerador_Estimador / y Denominador_Estimado
#r_norm = resultados(x=x, y=y_Final, n=n_Simul, alpha=0.05)
r_norm_num = resultados(x=x, y=y Numerador_Estimador, n=n_Simul, alpha=alpha)
r_norm_denom = resultados(x=x, y=y Denominador_Estimado, n=n_Simul, alpha=alpha)
r_norm_total = resultados(x=x, y=y_Final, n=n_Simul, alpha=alpha_ic)

#Sacamos Los términos del error para el tamaño mas grande de la simulación
x = y Numerador_Estimador[-1]
d_x=r_norm_num['inter_conf'][-1]
y = y Denominador_Estimado[-1]
d_y = r_norm_denom['inter_conf'][-1]

suma propagacion errores = (d_x/x) + (d_y/y)
print('Peor caso propagación de errores(suma numerador y denominador):', suma)

#suma_z propagacion errores = (d_z/z)
#z=y_Final[-1]
#d_z=r_norm_total['inter_conf'][-1]
#print('Peor caso propagación de errores(Cociente):', suma_z propagacion errores)

print('\n\nExpected probability a posteriori', y_Final[-1], ' +- ', r_norm[-1])
print('Confidence interval α =', alpha_ic)
print('Experimento:')
print('\t Total pacientes:', n, 'UCI: ', k)

```

In [67]:

```

n = 20*5
k = 1*5
n_Simul = 100000
alpha_ic = 0.05
get_posteriori_propagacion_error_maxima(a,b,n,k,n_Simul,alpha_ic)

```

Peor caso propagación de errores(suma numerador y denominador): 0.003547076728133547

Expected probability a posteriori 0.08635648830119663 +- 4.961932584470011e-06
 Confidence interval α = 0.05

Experimento:

Total pacientes: 100 UCI: 5

Observamos como en el peor de los casos, la propagación máxima de ambas variables aleatorias es de 0.003547076728133547. Por ello, deberíamos escoger un intervalo de confianza más ajustado que dicho error, para poder minimizarlo, y dar un buen estimador.

Vemos que cogiendo un α = 0.05, obtenemos un intervalo de confianza con exponente e-06, es decir, más pequeño que el error máximo con exponente e-03, por lo que podemos decir que α = 0.05 es un valor apropiado para el intervalo de confianza.

Vemos como a nivel práctico el error cometido es muy pequeño y el intervalo de confianza está bastante ajustado (se ajusta a cifras decimales a partir de orden 3). Por ello podemos concluir que el estimador que hemos proporcionado es bastante fiable.

Supón que no se dispone de un generador de números aleatorios para la distribución a priori (porqué o bien no dispones de un algoritmo o este es muy ineficaz) ¿Podrías estimar

μposterior ? Estima el valor de μposterior sin utilizar el generador de una muestra de Beta (por ejemplo utiliza una distribución U(0,1)). Discute el intervalo de confianza de la estimación.

Utilizamos el metodo de la inversa, para generar una muestra de la distribución beta a través de su función de distribución inversa. Definimos para ello la función apropiada. Si nos fijamos, el cambio con el caso anterior es que ahora generamos una muestra de la distribución uniforme, y le pasamos dicha muestra a la función de distribución inversa de la distribución Beta.

El resto del proceso es igual al caso anterior

In [145...]

```
def get_posteriori_muestreo_uniforme(a,b, n,k,n_simul,alpha_ic):

    #Utilizamos simulación de Monte Carlo mediante función de importancia
    x_uniform = scipy.stats.uniform.rvs(loc=0, scale=1, size=n_simul)
    x = scipy.stats.beta.ppf(x_uniform,a,b)

    y Numerador = h(x = x, g = g Numerador_final, f = scipy.stats.beta.pdf, a =
    y Denominador = h(x = x, g = g Denominador_final, f = scipy.stats.beta.pdf,

    # Evolución de E[theta] en función del número de simulaciones
    y Numerador_estimador= np.cumsum (y Numerador) / n_simul
    y Denominador_estimado= np.cumsum (y Denominador) / n_simul
    y Final = y Numerador_estimador/y Denominador_estimado
    r_norm = resultados(x=x, y=y Final, n=n_simul, alpha=0.05)

    print('\n\nExpected probability a posteriori', y Final[-1], ' +- ', r_norm[
    print('Confidence interval α =', alpha_ic)
    print('Maximum likelihood:', (k/n))
    print('Experimento:')
    print('\t Total pacientes:', n, 'UCI: ',k)

    fig, ax = plt.subplots()

    desde = 1000

    ax.plot (r_norm['size'][desde:], r_norm['media'][desde:], c='red' ,linestyle
    ax.plot (r_norm['size'][desde:], r_norm['ci_inf'][desde:], c='blue', alpha=0
    ax.plot (r_norm['size'][desde:], r_norm['ci_sup'][desde:], c='blue', alpha=0
    ax.fill_between(r_norm['size'][desde:],r_norm['ci_inf'][desde:],r_norm['ci_s
    #ax.hlines(1, 0, n, linestyles='dashed', lw=0.8)
    ax.set_xlabel('Montecarlo sample size')
    ax.set_ylabel('Expected probability a posteriori')
    ax.legend(loc='best', frameon=False)
    ax.grid(lw=0.2)
    ax.set_title('Estimación de probabilidad de theta a posteriori por Monte Car
    #ax.set_ylim(0, 0.4)
    plt.show()

    fig, ax = plt.subplots()
    desde = int(n_simul/2)
    ax.plot (r_norm['size'][desde:], r_norm['media'][desde:], c='red' ,linestyle
    ax.plot (r_norm['size'][desde:], r_norm['ci_inf'][desde:], c='blue', alpha=0
    ax.plot (r_norm['size'][desde:], r_norm['ci_sup'][desde:], c='blue', alpha=0
    ax.fill_between(r_norm['size'][desde:],r_norm['ci_inf'][desde:],r_norm['ci_s
    #ax.hlines(1, 0, n, linestyles='dashed', lw=0.8)

    ax.set_xlabel('Montecarlo sample size')
    ax.set_ylabel('Expected probability a posteriori')
```

```
ax.legend(loc='best', frameon=False)
ax.grid(lw=0.2)
ax.set_title('Estimación de probabilidad de theta a posteriori por Monte Carlo')
#ax.set_ylim(0, 0.4)
plt.show()
```

In [147...]

```
n = 20
k = 1
n_simul = 10000
alpha_ic = 0.05
get_posteriori_muestreo_uniforme(a,b, n,k,n_simul,alpha_ic)
```

Expected probability a posteriori 0.1730841585585165 +- 9.495093233396467e-05

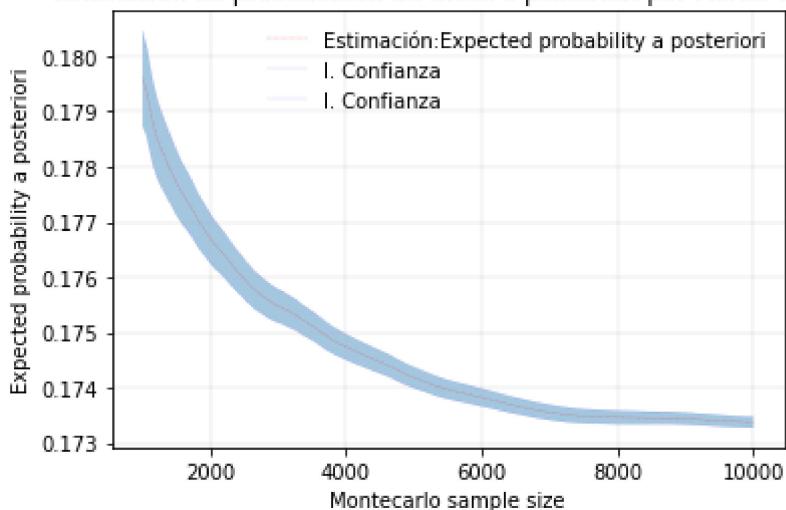
Confidence interval α = 0.05

Maximum likelihood: 0.05

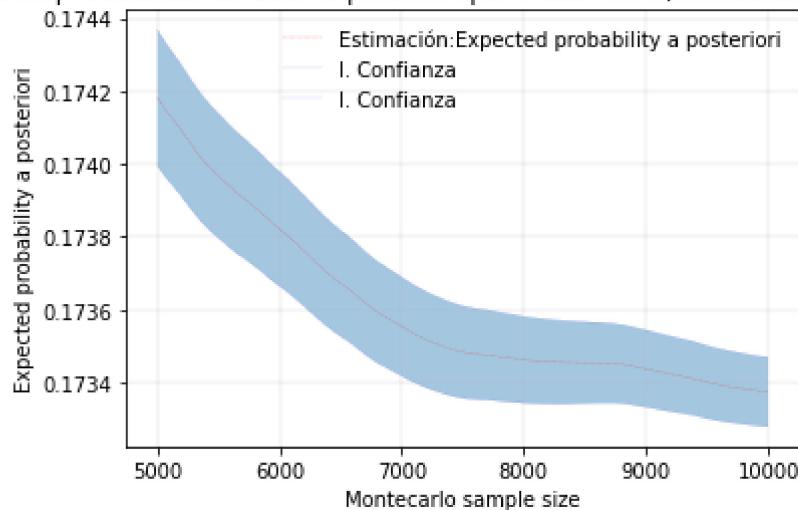
Experimento:

Total pacientes: 20 UCI: 1

Estimación de probabilidad de theta a posteriori por Monte Carlo



Estimación de probabilidad de theta a posteriori por Monte Carlo(Generador muestras uniforme)



Repetimos aumentando la proporción n/k como en los anteriores casos de estudio.

In [150...]

```
n = 20*5
k = 1*5
n_simul = 10000
get_posteriori_muestreo_uniforme(a,b, n,k,n_simul,alpha_ic)
```

Expected probability a posteriori 0.08472453723048073 +- 0.00015046172479600108

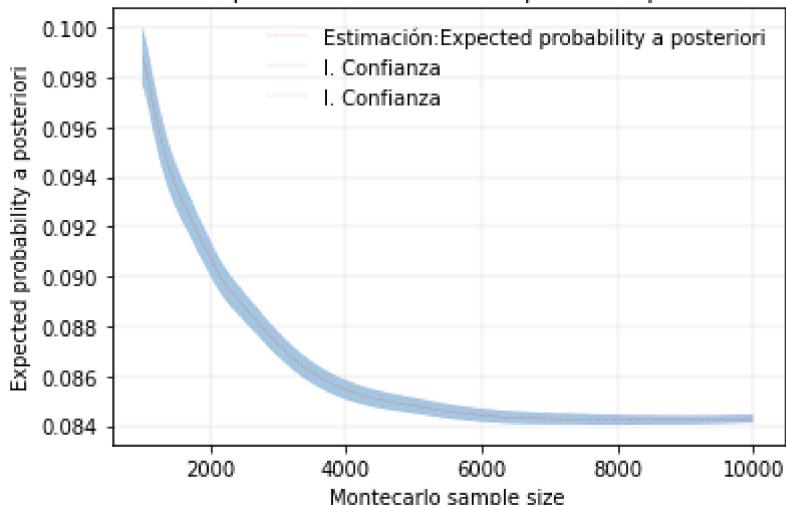
Confidence interval $\alpha = 0.05$

Maximum likelihood: 0.05

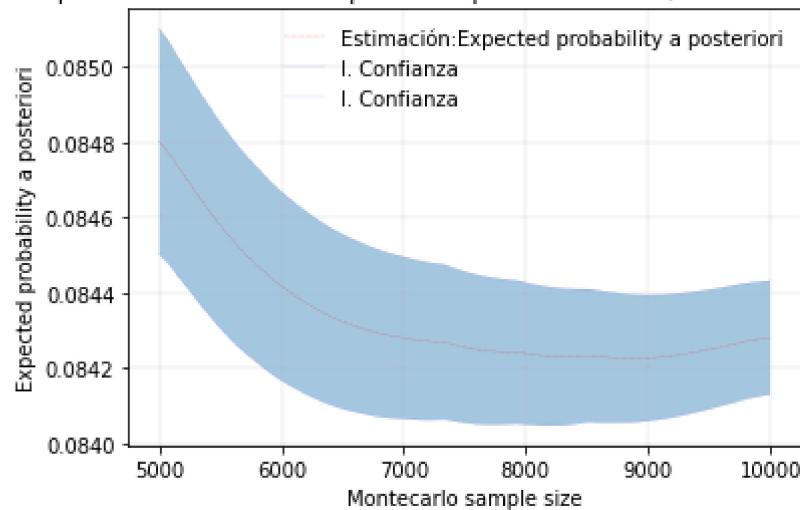
Experimento:

Total pacientes: 100 UCI: 5

Estimación de probabilidad de theta a posteriori por Monte Carlo



Estimación de probabilidad de theta a posteriori por Monte Carlo(Generador muestras uniforme)



A efectos prácticos vemos como ambos métodos son equivalentes (usar generador de la distribución vs. usar muestra uniforme + función de distrib. inversa).

Tal vez la ligera diferencia que observamos en los resultados es que con el mismo $\alpha = 0.05$, obtenemos intervalos de confianza de un orden de magnitud mayor, es decir con exponente e-06 en el caso de utilizar el generador de muestras propio de la distribución y con exponente e-05 en el caso de usar muestra uniforme + función de distrib. inversa para los dos casos de estudio.

Es decir, que aunque a efectos prácticos son equivalentes, lo cierto es que obtenemos un error ligeramente menor utilizando el generador de muestras propio de la distribución

EJERCICIO 2

Estimar empíricamente la función de densidad de probabilidad a posteriori $\pi(\theta|x)$ del ejercicio anterior y validarla con el resultado teórico. Discutir los resultados

Para resolver este ejercicio hemos utilizado el algoritmo de aceptación y rechazo. Este algoritmo se basa en generar muestras de una uniforme, de manera que si una muestra de nuestra distribución, dividida por una constante C , es mayor que una muestra de la distribución

uniforme, esta muestra se acepta. Si por el contrario nuestra muestra es menor, está será rechazada. Es decir:

$$u \in \mathcal{U}(0, 1)$$

$$b \in \mathcal{B}(\alpha, \beta)$$

$$C = Constante \Rightarrow C = Maxima\ Verosimilitud = k/n$$

El algoritmo resulta del siguiente modo: 1: Calcular $C = k/n = N^o Casos/N^o Muestras$ 2: Calcular muestra de distribución $\text{Beta}(\alpha = 5, \beta = 10) = b$ 3: Calcular muestra de distribución Uniforme(0,1) = u 4: Si $u < b/C \Rightarrow b$ se acepta.

Una vez terminado este algoritmo con la ayuda de scipy, pasamos a calcular la eficiencia y realizamos el test de kolmogorov.

La eficiencia es la tasa de muestras aceptadas. Por ejemplo, si se aceptan 20 de un total de 100, la tasa es $20/100 = 0,5$.

Este test nos ayuda a contrastar si nuestra muestra es ideal o compatible a la hora de compararla con la el valor de la distribución teórica. Este test se basa en estimar la distancia máxima entre la muestra obtenida y una $\mathcal{U}(0, 1)$, la cual es una variable aleatoria.

El test de kolmogorov nos da 2 valores: el pvalor y las estadísticas. Un pvalor bajo significa que la probabilidad se aleja mucho de la distribución, es decir es una probabilidad muy rara para la distribución. Por tanto significa ambos grupos o distribuciones fueron sampleados de diferentes poblaciones, aunque las poblaciones pueden diferir en cuanto a medias, variabilidad o forma de la distribución. Por todo esto queremos conseguir un pvalor lo mas alto posible.

Cuanto mayor sea el pvalor, mayor serán las estadísticas recibidas. Unas estadísticas altas significa que ambas muestras han sido sampleadas de las mismas poblaciones.

In [151...]

```
def prob_aceptar_rechazar_muestra(t, cte_a, k, n):
    return verosimilitud(t, k, n)/cte_a
```

In [154...]

```
n_simul = 100000
n = 20
k = 1

#cte_a = verosimilitud(k/n, k, n)
cte_a = k/n

# Generamos una muestra de la distrib. a priori
x = scipy.stats.beta.rvs(a, b, size=n_simul)
# print(len(x), x)

probs_x = prob_aceptar_rechazar_muestra(x, cte_a, k, n)
probs_u = scipy.stats.uniform.rvs(size=n_simul)
muestras_distrib = []
for z, p, u in zip(x, probs_x, probs_u):

    if(u < p):
        muestras_distrib.append(z)

print('Eficiencia...', len(muestras_distrib)/n_simul)
```

```
# Gráfica
fig, ax=plt.subplots(1, figsize=(8,3))
w = np.linspace(0.01, 0.99, 100)

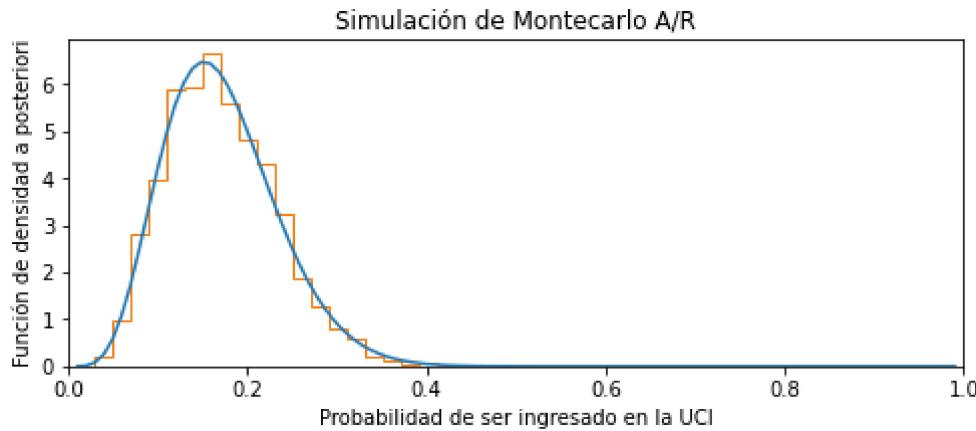
ax.plot(w, scipy.stats.beta.pdf(w, a+k, b+n-k))
ax.hist(muestras_distrib, histtype='step', bins=20, density=True, linewidth=1, label='Montecarlo A/R')
plt.xlim(0, 1)

plt.ylabel('Función de densidad a posteriori')
plt.xlabel('Probabilidad de ser ingresado en la UCI')
plt.title('Simulación de Montecarlo A/R')

plt.show()

print(scipy.stats.kstest(muestras_distrib, 'beta', args=(a+k, b+n-k)))
```

Eficiencia... 0.02433



KstestResult(statistic=0.010107719753527089, pvalue=0.9627216497935536)

Metropolis-Hastings

Existen otros algoritmos de MCMC que realizan lo mismo que hemos contemplado con el algoritmo aceptación y rechazo. Uno de los más famosos es el algoritmo metropolis-hastings, usando cadenas de markov.

Queremos obtener muestras de una distribución $p(x)$, pero no conocemos la distribución, solo conocemos el numerador $f(x)$, de forma que:

$$p(x) = f(x)/c$$

Por lo tanto buscamos conseguir muestras de $p(x)$ únicamente con $f(x)$. Para ello, se usa una cadena de markov que consigue unas primeras muestras poco acertadas al modelo (llamadas burn-in), pero a medida que se generan más, las muestras generadas son como las de $p(x)$. Este punto de roptura en el que se empiezan a tratar las muestras como de la propia distribución $p(x)$ se llama X_b .

Para realizar esto, a M-H se le propone un nuevo candidato a proponer, como en aceptación y rechazo. En Aceptación y rechazo esta propuesta es independiente, pero en M-H se utiliza la media de una distribución normal $\mathcal{N}(x_t, \sigma^2)$, o una más sencilla de extraer muestras, de manera que sea la muestra exacta anterior considerada. De manera que $g(x_{t-1}|x_t)$

El siguiente paso es aceptar este caso $x_t + 1$ con la probabilidad de aceptar esa transición = $A(x_t -> x_t + 1)$

Esto cumple una condición equilibrada detallada que dice: que por cualquier estas: a,b

$$p(a)T(a \rightarrow b) = p(b)T(b \rightarrow a)$$

Siendo $T(a \rightarrow b)$ la probabilidad de transitad de a a b. Con lo mencionado anteriormente llegamos a que, $p(a) = f(a)/C$, y $T(a \rightarrow b)$ es $g(b|a)A(a \rightarrow b)$. Es decir, la probabilidad de la transición es la probabilidad de que el estado b sea propuesto desde a, mas la probabilidad de aceptar el estado b estando en a. Por tanto:

$$\frac{f(a)}{C} * g(b|a) * A(a \rightarrow b) = \frac{f(b)}{C} * g(a|b) * A(b \rightarrow a)$$

Despejando podemos llegar a esta equación para simplificar:

$$\frac{A(a \rightarrow b)}{A(b \rightarrow a)} = \frac{f(b)}{f(a)} \frac{g(a|b)}{g(b|a)}$$

Donde $\frac{f(b)}{f(a)} = Rf = \text{Ratio de } f$

$\gamma \frac{g(a|b)}{g(b|a)} = Rg = \text{Ratio de } g$

La definicion del algoritmo MCMC M-H dice que:

si $Rf * Rg < 1$, entonces $A(a \rightarrow b) = RfRg$ y $A(b \rightarrow a) = 1$

si $Rf * Rg \geq 1$, entonces $A(a \rightarrow b) = 1$ y $A(b \rightarrow a) = \frac{1}{RfRg}$

Todo esto se puede resumir en una equación

$$A(a \rightarrow b) = \min(1, RfRg)$$

Como conclusión, el algoritmo M-H basicamente genera muestras acercandose cada vez mas al punto de mayor densidad de la población desconocida o propuesta, o a un punto que tiene mayor probabilidad en ella. Esto se da por la relacación entre las muestras, estas no son independientes como en el algoritmo aceptación y rechazo. Por ello este metodo es muy poderoso y de los mas famosos hoy en día.

In [155...]

```
# Ejemplo algoritmo metropolis
def metropolis(func, pasos=10000):
    """A very simple Metropolis implementation"""
    muestras = np.zeros(pasos)
    old_x = func.mean()
    old_prob = func.pdf(old_x)

    for i in range(pasos):
        new_x = old_x + np.random.normal(0, 0.5)
        new_prob = func.pdf(new_x)
        aceptacion = new_prob / old_prob
        if aceptacion >= np.random.random():
            muestras[i] = new_x
            old_x = new_x
            old_prob = new_prob
        else:
            muestras[i] = old_x

    return muestras
```

In [156...]

```

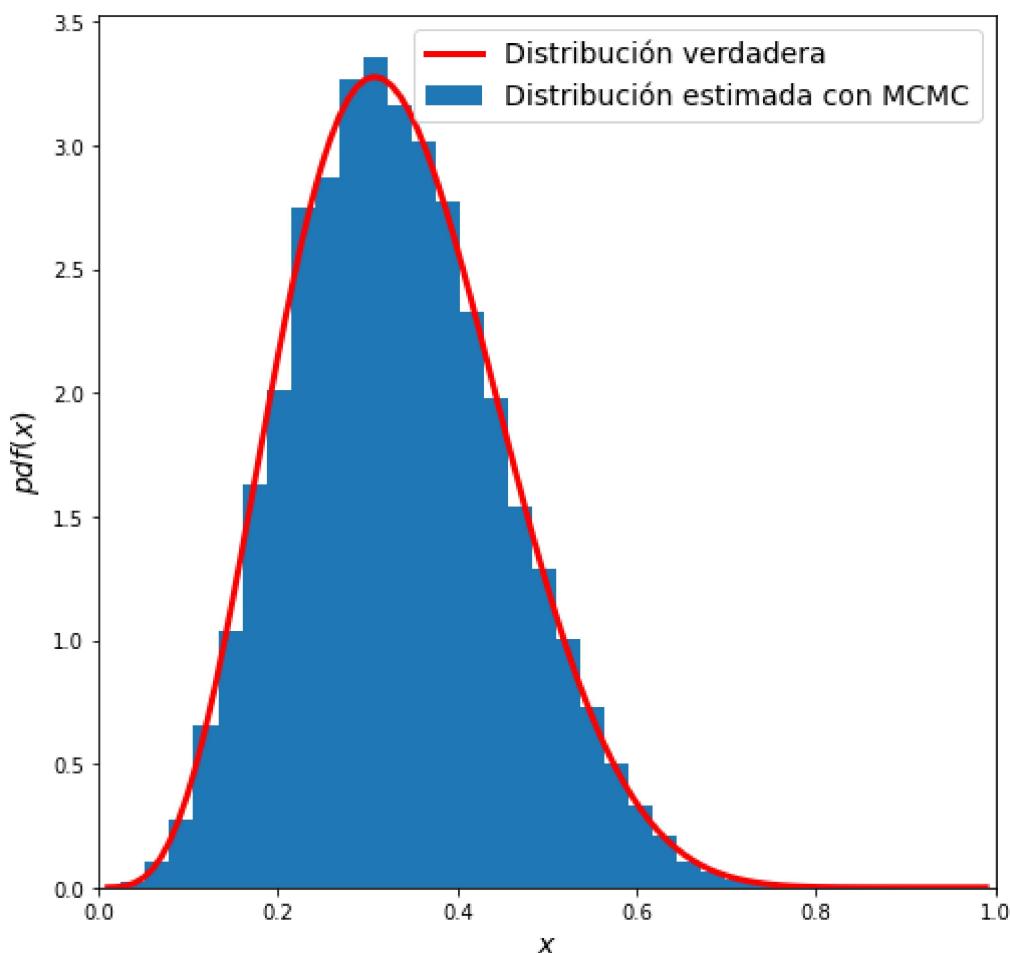
# distribución beta
beta = scipy.stats.beta(a, b)
muestras = metropolis(func = beta, pasos = 100000)
x = np.linspace(0.01, .99, 100)
y = beta.pdf(x)
plt.figure(figsize=(8, 8))
plt.xlim(0, 1)
plt.plot(x, y, 'r-', lw=3, label='Distribución verdadera')
plt.hist(muestras, bins=30, density=True, label='Distribución estimada con MCMC')
plt.xlabel('$x$', fontsize=14)
plt.ylabel('$pdf(x)$', fontsize=14)
plt.legend(fontsize=14)
plt.show()

print(scipy.stats.kstest(muestras, 'beta', args=(a, b)))

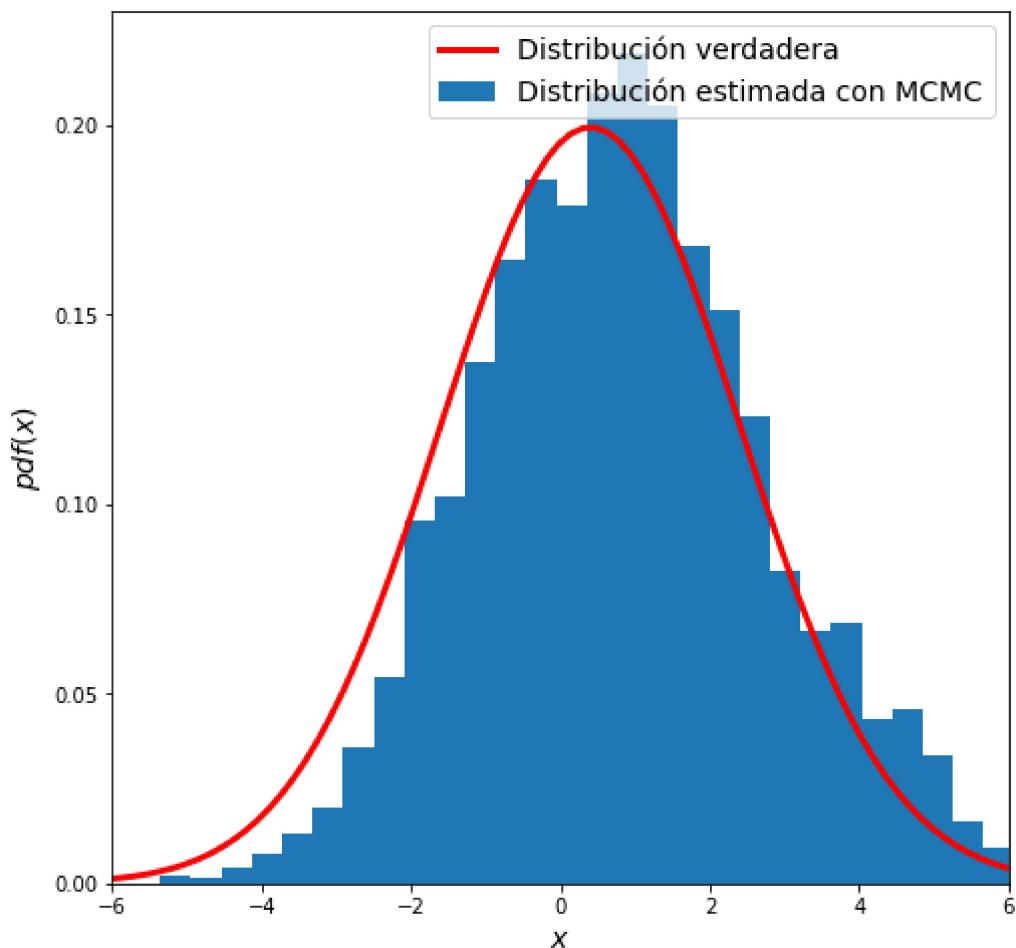
# distribución normal
norm = scipy.stats.norm(0.4, 2)
muestras = metropolis(func = norm)
x = np.linspace(-6, 10, 100)
y = norm.pdf(x)
plt.figure(figsize=(8,8))
plt.xlim(-6, 6)
plt.plot(x, y, 'r-', lw=3, label='Distribución verdadera')
plt.hist(muestras, bins=30, density=True, label='Distribución estimada con MCMC')
plt.xlabel('$x$', fontsize=14)
plt.ylabel('$pdf(x)$', fontsize=14)
plt.legend(fontsize=14)
plt.show()

print(scipy.stats.kstest(muestras, 'norm', args=(0.4, 2)))

```



KstestResult(statistic=0.007328442160809606, pvalue=4.305398546390386e-05)



KstestResult(statistic=0.08291811396860688, pvalue=2.9352119948677536e-60)

Cita para Metropolis-Hastings: <https://relopezbruega.github.io/blog/2017/01/10/introduccion-a-los-metodos-de-monte-carlo-con-python/>