Primer Parcial Inferencia II

Santiago Robatto

Ejercicio 1: (Algoritmo de Metropolis-Hastings: cambio del modelo de propuesta)

Antes de comenzar con la resolución del ejercicio, haremos un breve repaso respecto al funcionamiento del algoritmo de Metrópolis Hastings y de otros conceptos previos. A modo introductorio, el algortimo de Metropolis Hastings, es un metodo para construir Monte Carlo Markov Chains (MCMC), o, en otras palabras, que permite simular una distribución a partir de cadenas de Markov.

¿Qué es un método de Monte Carlo? Son un tipo de algoritmo computacional que utilizan muestreo aleatorio repetido, basado en cierta distribución de probabilidad elegida por el usuario para obtener la probabilidad de que ocurra una serie de resultados. De otra manera, una simulación de Montecarlo construye un modelo de posibles resultados aprovechando una distribución de probabilidad, como pueden ser la uniforme o normal, para cualquier variable que tenga incertidumbre inherente. A continuación, vuelve a calcular los resultados una y otra vez. En un experimento típico de Montecarlo, puede repetirse miles de veces para producir un gran número de resultados probables.

¿Qué es una Markov Chain? Una cadena de Markov es una serie de eventos, en la cual la probabilidad de que ocurra un evento depende del evento inmediato anterior. Es decir que las cadenas de este tipo tienen memoria, "recuerdan" el ultimo evento y esto condiciona las posibilidades de los eventos proximos. Esta dependencia respecto al evento anterior distingue a las cadenas de Markov de las series de eventos independientes, como tirar una moneda al aire o un dado.

Entonces, **¿Qué son los MCMC?** Son metodos que combinan Monte Carlo (muestreo aleatorio) con cadenas de Markov (dependencia del estado previo) para obtener muestras de distribuciones complejas, asegurando que la distribución límite de la cadena sea la deseada. En otras palabras, Los MCMC evolucionan la simulacion de Monte Carlo clasica, de modo tal que en lugar de ser muestras independientes cada una se obiente a partir de la anterior, generando asi dependencia entre estas (por ello es que es una cadena de Markov).

Aclarados estos conceptos previos, podemos conocer y entender **el algoritmo de Metropolis Hastings**.

El algoritmo realiza los siguientes pasos:

1. Dado un valor actual de μ, propone un candidato μ', a partir de una distribución propuesta $p(\mu' \mid \mu)$. Esta distribucion es denominada proposal. Cuando el proposal es simetrico decimos que el metodo es un "Metropolis Algorithm" y tiene propiedades bondadosas que simplifican las reglas de aceptacion de moverse o no a la nueva locacion propuesta.
2. Calcula la probabilidad de aceptar el nuevo valor, bajo la siguiente regla: $\alpha = \min\left\{1, \frac{f(\mu'|y)}{f(\mu|y)}\right\}$
3. Cuando $f(\mu' \mid y) > f(\mu \mid y)$ entonces el minimo es 1 y nos movemos automaticamente a la nueva locacion.

   Cuando $f(\mu' \mid y) < f(\mu \mid y)$ podriamos aceptarlo o no. Para decidirlo, realizaremos un sorteo aleatorio con probabilidad α de aceptacion y $1 - \alpha$ de rechazar.

La funcion dada para realizar una unica iteracion de Metropolis - Hastings es la siguiente, y la modificaremos de modo que en vez de realizar el calculo del proposal con el modelo uniforme, se realice con la distribucion normal:

```
one_mh_iteration <- function(w, current){
 # STEP 1: Calculo del proposal con el modelo uniforme
 proposal <- runif(1, min = current - w, max = current + w)

 # STEP 2: Decide whether or not to go there
 proposal_plaus <- dnorm(proposal, 0, 1) * dnorm(6.25, proposal, 0.75)
 current_plaus  <- dnorm(current, 0, 1) * dnorm(6.25, current, 0.75)
 alpha <- min(1, proposal_plaus / current_plaus)
 next_stop <- sample(c(proposal, current),
                     size = 1, prob = c(alpha, 1-alpha))

 # Return the results
 return(data.frame(proposal, alpha, next_stop))
}
```

Como se observa en el STEP 1, el proposal sigue una distribucion uniforme en el intervalo (current - w, current + w).

Entonces, debemos modificar la manera que se genera dicho proposal para que sea desde una normal. Se realizara con la funcion rnorm, a la que le pasaremos los parametros current (media inicial) y s(desvio estandar).

¿Qué cambia al usar un proposal normal?

A contraposicion de la distribucion uniforme que limita su recorrido al intervalo [current - w, current + w], al utilizar un proposal normal liberamos el rango de valores posibles para μ a todos los reales, dado que este es el recorrido de cualquier distribucion normal independientemente de sus parametros.

Por otro lado,simular el proposal con una Normal nos da más control sobre cómo se mueve la cadena. Si elegimos un desvío muy chico, los valores propuestos quedarán pegados al actual y la cadena avanzará despacio, parecido a lo que pasaba con la uniforme. En cambio, con un desvío más grande aparecen saltos más largos que permiten explorar mejor el espacio, aunque a costa de más rechazos. Aun así, estos saltos seguiran teniendo cierto "control", porque los valores mas cercanos al estado actual siguen siendo los más probables, dado que la normal es simetrica entorno a la media.

En otras palabras, dejamos atrás la equiprobabilidad de la uniforme, donde todos los puntos dentro del intervalo eran igual de probables. Con la normal, la cadena se mueve la mayor parte del tiempo de a pasos cortos, pero cada tanto puede dar saltos más largos. Esto en el modelo uniforme era sumamente dificil de lograr, dado que dependiamos en gran medida del tamano del intervalo elegido.

La **funcion modificada** es la siguiente:

```
one_mh_iteration_normal <- function(s, current){
 # STEP 1: Proponer la siguiente locacion con el modelo normal
 proposal <- rnorm(1, mean=current, sd=s)

 # STEP 2: Decide whether or not to go there
 proposal_plaus <- dnorm(proposal, 0, 1) * dnorm(6.25, proposal, 0.75)
 current_plaus  <- dnorm(current, 0, 1) * dnorm(6.25, current, 0.75)
 alpha <- min(1, proposal_plaus / current_plaus)
 next_stop <- sample(c(proposal, current),
                     size = 1, prob = c(alpha, 1-alpha))

 # Return the results
 return(data.frame(proposal, alpha, next_stop))
}
```

Los datos del ejercicio son:

1. El prior es: $\mu \sim \mathcal{N}(0, 1)$
2. La verosimilitud es: $Y \mid \mu \sim \mathcal{N}(\mu, 0.75^2)$, con un dato observado $y = 6.25$.

A partir de ellos, utilizando la funcion summarize_normal_normal, deducimos que el posterior teorico es:

$$\mu \mid Y \sim \mathcal{N}(4.0, 0.36)$$

Simulaciones para distintos parametros

Simulacion con s=0.01 y current=3:

proposal	alpha	next_stop
2.993736	0.9826955	2.993736

Observamos que el valor del proposal es sumamente cercano a 3 (Valor de la media que cargamos para realizar la iteracion). Es tan cercano dado que utilizamos una desviacion estandar sumamente pequena.

El intervalo que acumula 95% de probabilidad se concentra de la siguiente manera, lo que hara que la mayor parte de las veces los valores sean inmediatos a 3.

Table 2: Intervalo de probabilidad 95% en Normal(3,0.01^2)

Percentil	Valor
2.5%	2.9804
97.5%	3.0196

La desviacion tan pequena genera que el proposal sea muy parecido al current, por ende α sera siempre 1 o muy cercano a 1. Esto hara que la mayor parte de las veces aceptemos el valor del proposal, dado que en caso de realizarse el sorteo se hara con probabilidad muy alta de ser verdadera. De este modo, no nos estancaremos en ningun valor, pero a cambio recorreremos muy lentamente la cadena.

Simulacion con s=0.5 y current=3

En este caso se decidio primero mirar los cuartiles Q1 y Q3:

Table 3: Intervalo de probabilidad 50% en Normal(3,0.5^2)

Percentil	Valor
25%	2.662755
75%	3.337245

Es decir que el valor estara con 50% de probabilidad entre 2.66 y 3.37. Se observa claramente que si s=0.5 repartimos de mejor manera la probabilidad que le vamos a otorgar a los valores del recorrido.

proposal	alpha	next_stop
2.686773	0.3655544	3

El valor del proposal es considerablemente menor que 3, yendo hacia el lado contrario de las observaciones y del verdadero posterior, lo que genera que la probabilidad de aceptacion α disminuya. Se realizo el sorteo con α=0.365 y en este caso en particular se rechazo el valor del proposal.

s=0.5 nos dio un valor para el proposal que es muy interesante: el sorteo tiene probabilidad considerable de ser verdadero o falso. Es decir que antes de realizar el sorteo no podemos tener tanta intuicion al respecto de lo que va a pasar, como si podiamos tener si cuando realizamos el sorteo con α=0.98.

Simulacion con s=1 y current=3

En este caso en primer lugar miraremos los percentiles 10% y 90% para saber que podemos esperar:

Table 5: Intervalo de probabilidad 80% en Normal(3,1)

Percentil	Valor
10%	1.718448
90%	4.281552

El intervalo que acumula 80% de probabilidad es considerablemente amplio, por lo que no tenemos certeza de donde estara nuestro valor del proposal.

proposal	alpha	next_stop
2.373546	0.1017526	3

El valor del proposal fue 2.37. Esto genera un $\alpha = 0.101$, el cual es pequeno y poco probable en el sorteo. Al alejarse del verdadero posterior mas que el valor actual (3), la plausibilidad de este valor es menor que la plausibilidad de 3.

En cambio, si hubiesemos tomado el proposal=3.63, que es el simetrico entorno a 3 del proposal actual, el valor hubiese sido mas plausible que el actual (3) porque esta mas cerca de los datos observados y por ende α=1.

A continuacion lo demostraremos, modificando la funcion de modo que tome el proposal igual a 3.63:

```
one_mh_iteration_3.63 <- function(current){
 # STEP 1: Propose the next chain location
 proposal <- 3.63

 # STEP 2: Decide whether or not to go there
 proposal_plaus <- dnorm(proposal, 0, 1) * dnorm(6.25, proposal, 0.75)
 current_plaus  <- dnorm(current, 0, 1) * dnorm(6.25, current, 0.75)
 alpha <- min(1, proposal_plaus / current_plaus)
 next_stop <- sample(c(proposal, current),
                     size = 1, prob = c(alpha, 1-alpha))

 # Return the results
 return(data.frame(proposal, alpha, next_stop))
}
s4<-one_mh_iteration_3.63 (current=3 )
kable(s4)
```

proposal	alpha	next_stop
3.63	1	3.63

Tal como esperabamos, α es igual a 1 y por ende la simulacion decide cambiar al nuevo valor, dado que este es mas plausible.

Ejercicio 2: (Recorrido de Metropolis-Hastings con propuestas Normales)

Emplearemos la funcion *mh_tour*, y la modificaremos de modo que involucre a la funcion creada en el item anterior.

La funcion es:

```
mh_tour <- function(N, w, current){
  # Inicializamos la simulacion, creando un vector de ceros de tamano N.
  mu <- rep(0, N)

#Simulamos N veces en la cadena.

  for(i in 1:N){
    # Simulamos una iteracion con la funcion que definimos antes, utilizando el proposal uni
    sim <- one_mh_iteration(w = w, current = current)

    #Avanzar a la siguiente locacion.
    mu[i] <- sim$next_stop

    #Reseteamos el valor de current para volver a iterar.
    current <- sim$next_stop
  }

  # Retornamos los valores iterados.
  return(data.frame(iteration = c(1:N), mu))
}
```

Esta funcion repite N veces la iteracion que definimos en el ejercicio anterior. Por ende, la modificacion a realizar es cambiar one_mh_iteration por one_mh_iteration_normal que ya definimos previamente.

```
mh_tour_normal <- function(N, s, current){
  # Inicializamos la simulacion, creando un vector de ceros de tamano N.
  mu <- rep(0, N)

#Simulamos N veces en la cadena.

  for(i in 1:N){
    # Simulamos una iteracion con la funcion que definimos para la normal
    #que en el ejercicio anterior.
    sim <- one_mh_iteration_normal(s = s, current = current)
```

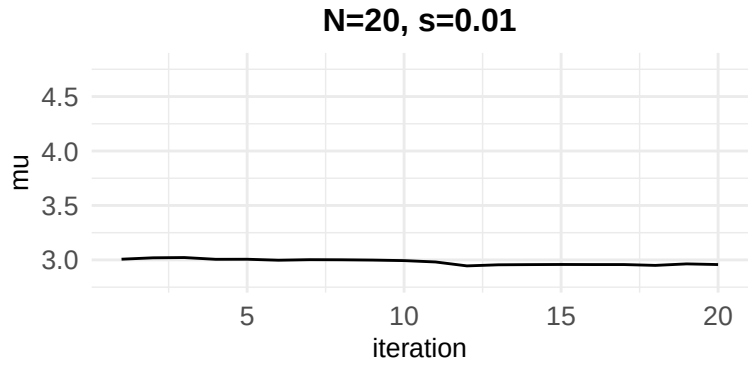
```
    #Avanzar a la siguiente locacion.
    mu[i] <- sim$next_stop

    #Reseteamos el valor de current para volver a iterar.
    current <- sim$next_stop
  }

  # Retornamos los valores iterados.
  return(data.frame(iteration = c(1:N), mu))
}
```

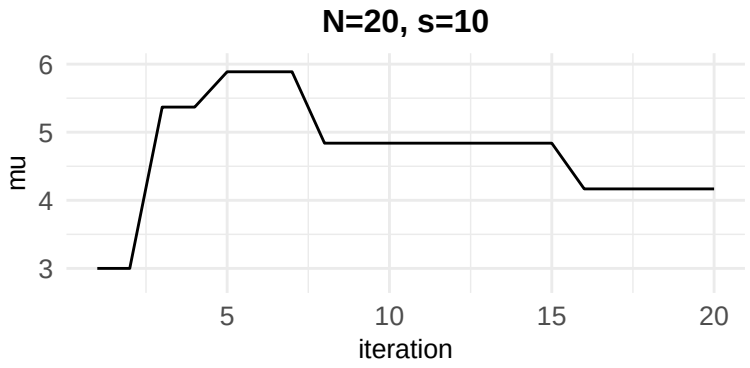
Parte B: Correr mh_tour.

Se procedera a correr la funcion con distintos valores para s y N. En este primer caso se grafico s=0.01 y N=20.



Cuando s=0.01 todos los valores generados son muy cercanos al actual, es decir al valor de *current* que le pasamos y es por ello que le cuesta mucho alejarse de los mismos, dado que los saltos que da son muy pequenos. En este caso, como cargamos **current=3**, todos los valores son cercanos a 3.

El resultado de la varianza tan pequena, es una cadena sumamente estable, pero que tiene un problema: le cuesta mucho recorrer el espacio parametrico. Esto implica que necesitaremos un N considerablemente grande para poder recorrerla de manera correcta, siendo ineficiente computacionalmente por una cuestion de recursos empleados.



En contraposicion, cuando s=10 la situacion es opuesta. Observamos una cadena inestable, que pega saltos considerables y tiene mesetas donde se estanca. Esto ocurre dado que cuando la varianza es tan grande en comparacion a la media, los valores propuestos con rnorm (en este caso) se pueden alejar mucho de la misma, obteniendo verosimilitudes muy chicas y por ende el α muy cercano a cero, lo que nos llevara a realizar sorteos con probabilidad muy baja de ser verdaderos.

De todos modos, esta cadena tiene menor dificultad para salir de los estancamientos que cuando usamos runif, porque los valores se producen centrados entorno a μ, siendo este el valor mas probable para ser simulado. Es decir, por mas grande que sea la varianza, siempre los valores mas probables seran los mas cercanos a la media, en mayor o menor medida dependiendo de que tan grande sea la varianza, lo que le reduce la dificultad a la hora de salir de los estancamientos.

Resulta interesante comparar el metodo de MH con proposal uniforme versus el metodo con proposal normal. Para ello deduciremos el valor de w que nos brinde la misma varianza que s=10 en la normal.

En el modelo normal s representa el desvio estandar, por ende $\sigma^2 = \text{Var}(X) = 10^2 = 100$

Despejaremos w para que la varianza del proposal uniforme sea la misma, es decir 100. Sabemos que $X \sim U(3-w,\, 3+w)$, por ende:

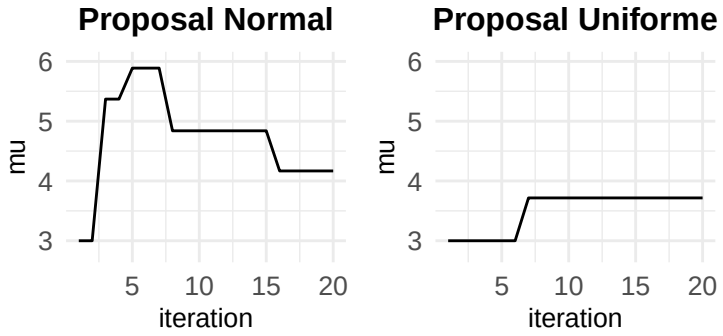
$$a = 3 - w, \quad b = 3 + w$$

$$b - a = (3 + w) - (3 - w) = 2w$$

$$\text{Var}(X) = \frac{(b-a)^2}{12} = \frac{(2w)^2}{12} = \frac{w^2}{3}$$

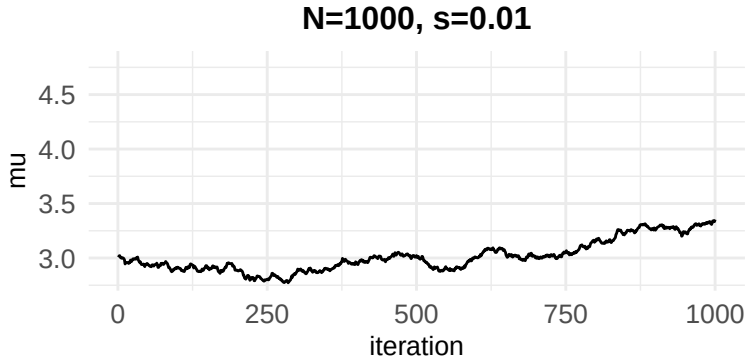
$$\frac{w^2}{3} = 100 \;\Rightarrow\; w^2 = 300 \;\Rightarrow\; w = \sqrt{300} \approx 17.32$$

Al comparar los trace_plot, se observa que la iteracion con proposal normal realiza 6 cambios de valor, mientras que la iteracion con proposal uniforme solo 1. Por ende, podemos pensar que nuestra creencia inicial de que el proposal normal es menos susceptible a una configuracion incorrecta de parametros es correcta.



Analisis cuando *N=1000 y s=0.01*

Volviendo al ejercicio, procederemos a simular, graficar y comparar que pasa con las cadenas de distinto desvio si agrandamos el valor de N:



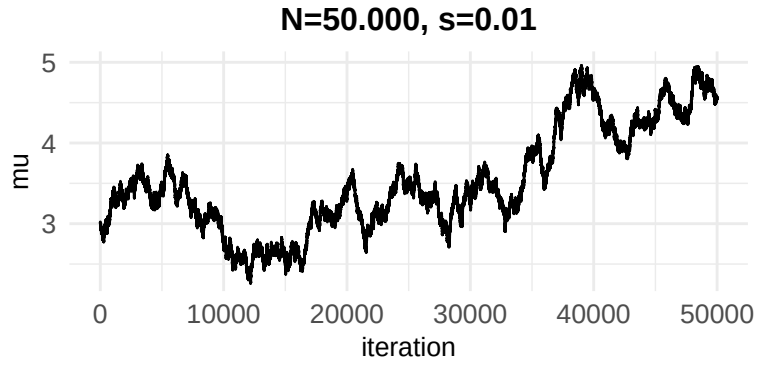
Observamos que la cadena generada con desvio *s=0.01* es sumamente estable, lo que le impide recorrer el espacio parametrico de manera correcta. En mas de 1000 observaciones no logra superar la barrera de 3.5.

Teniendo en cuenta que sabemos que el verdadero valor de μ es 4, concluimos a simple que resulta mala la simulacion con *s=0.01*.

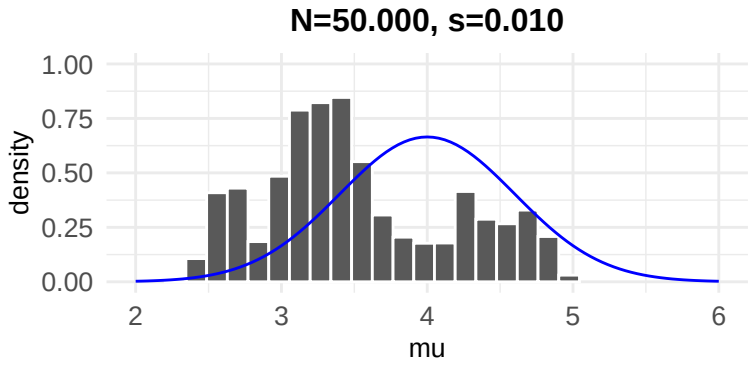
Por ejemplo, si sacrificamos eficiencia agrandando a *N=50.000*, recien ahi obtenemos una cadena que logra recorrer de mejor manera los valores plausibles de μ.

De todos modos esta no es una cadena con la forma deseada, dado que no oscila alrededor de la media del posterior. Tampoco presenta saltos medianos que generen la aparencia ruidosa, y por ultimo y mas importante, no converge entorno al valor del posterior.

Solo presenta dos propiedades bondadosas, y es que cubre el espacio parametrico y ademas no presenta estanciamientos.

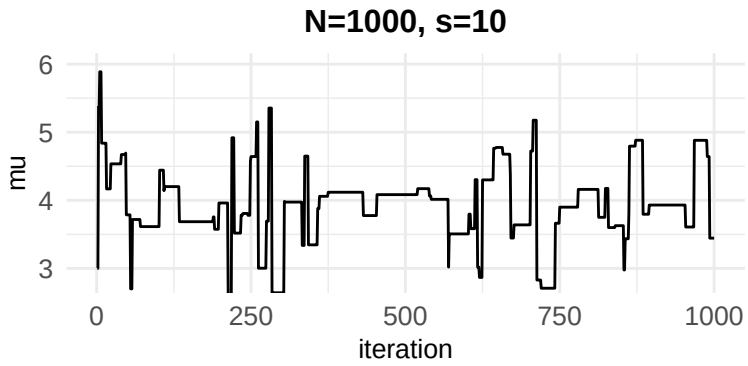


Si generamos el historgrama y lo comparamos con el posterior teorico, vemos que lo representa de mala manera dado que no esta centrado en la media y tampoco tiene la forma caracteristica de un modelo normal. En conclusion, la cadena subestima considereblemente a μ y por ende no sirve.



Analisis para *n=1000 y s=10*

Tal como nos esperabamos, se observan muchisimos saltos grandes y estancamientos largos:

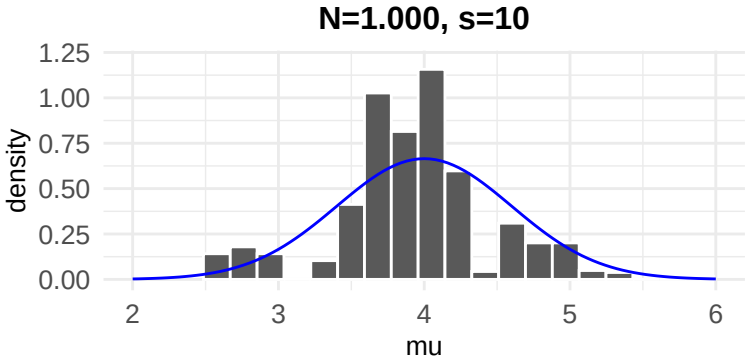


Como aspectos positivos, se destaca que los valores se encuentran relativamente centrados entorno al posterior verdadero (4), que cubre todo el espacio parametrico y que muestra algunos saltos medianos y largos.

Por otra parte, como aspectos negativos se destacan los estancamientos, la falta de apariencia ruidosa y la falta de estabilidad.

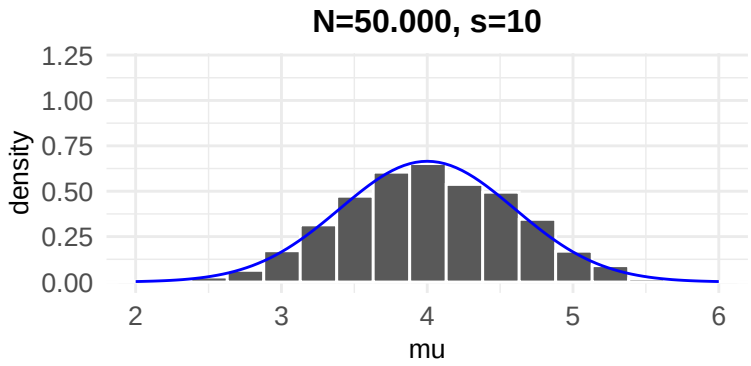
Si generamos el histograma, observamos que simula de manera bastante centrada al posterior, lo cual es destacable. Sin embargo, al entrar en detalle, tambien subestima levemente a μ. A su vez, no representa de manera correcta a los valores extremos y le asigna mas probabilidad a los valores entre 3.5 y 4.

De todos modos, a pesar de haber iterado 49.000 veces menos la aproximacion con s=10 y n=1000 es mejor que la aproximacion con s=0.01 y n=50.000.



Si volvemos a generar el histograma, pero esta vez con *N=50.000*, se observa que se corrigen los defectos mencionados anteriormente, y ajusta de manera correcta a una normal.

No se observan picos por encima de la probabilidad y esta perfectamente centrada, por lo que no subestima ni sobreestima a μ. Tambien cabe destacar que ajusta de manera correcta a los valores mas extremos.



De esta manera, **concluimos que en el modelo con prposal normal se ajusta mejor s=10 antes que s=0.01**, dado que implica menos reiteraciones ajustar el modelo de manera correcta.

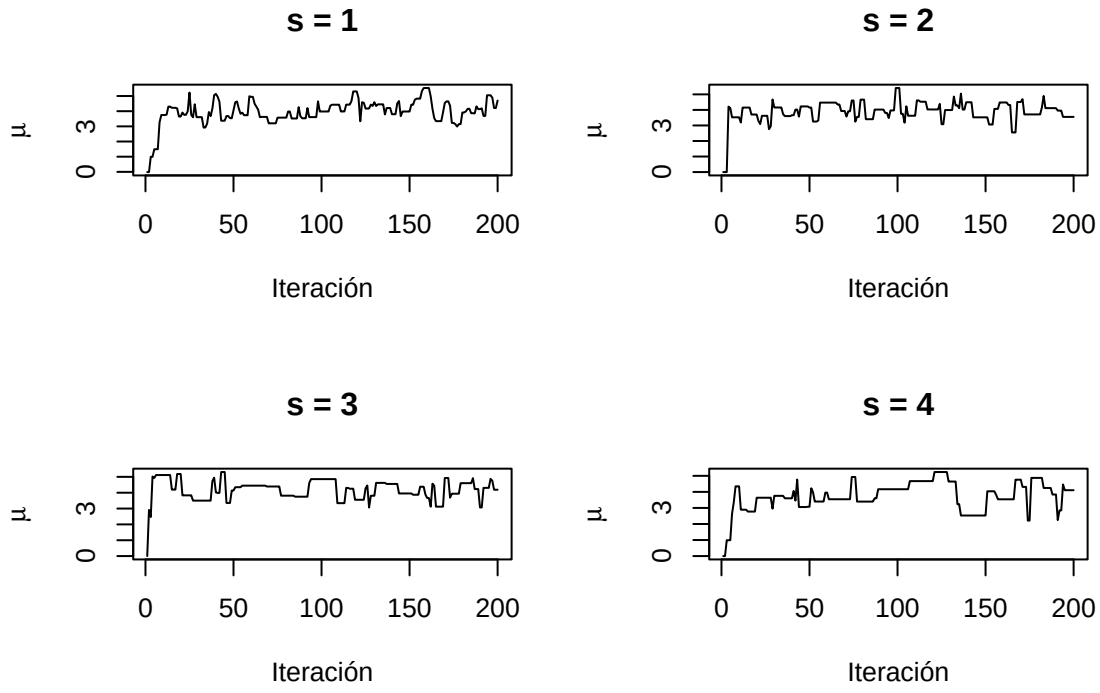
Generalizando esta conclusion, sera preferible sobreestiamar s antes que subestimarlo, dado que el costo de subestimarlo requiere infinitamete mas recursos computacionales.

Parte D: Busqueda del ajuste perfecto.

Como se vio en la parte anterior, cuando el proposal es normal, con un N lo suficientemente grande, podremos compensar un desvio demasiado grande. Pero esto no es eficiente en cuestion de recursos, por lo que buscaremos el valor que necesite menor tamano para ajustar correctamente. Tambien debemos recordar que si este valor es demasiado pequeno, como lo era *s=0.01*, el valor se estancara muy rapidamente. Es por ello que sabemos que el valor que buscamos pertenece al intervalo [0.01 ; 10].

Para encontrarlo de manera rapida, con la ayuda de ChatGpt, se creo una nueva funcion que simula y grafica la trasa automaticamente para los valores de s que le pase.

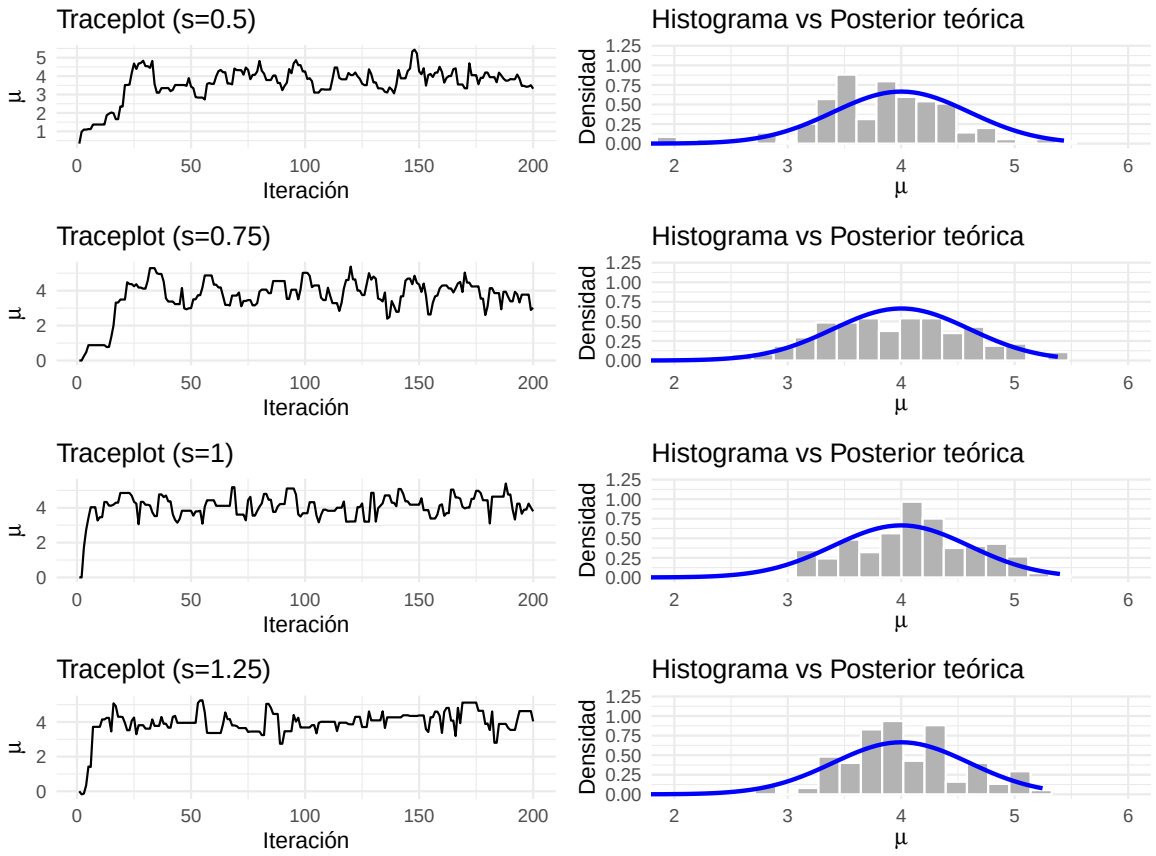
Correremos esta funcion con para una secuencia de valores de 1 a 4 separados de a 1, es decir para *s=1,s=2, s=3 y s=4*



Facilmente se observa en esta secuencia que el que mejor ajusta es 1, dado que todos los valores de s mayores a 1 presentan saltos grandes y estancamientos. La traza cuando *s=1* practicamente no presenta mesetas, tiene saltos pero no son extremos y esta centrada correctamente alrededor de 4.

Con esto ya nos hicimos una idea de donde debemos buscar: alrededor de 1. Repetiremos el experiento para *s=0.5, s=0.75, s=1 y s=1.25* , esta vez graficando a izquierda la traza y a la derecha su correspondiente histograma.

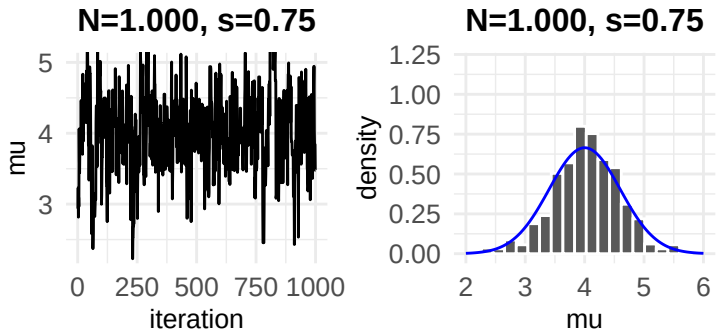
Seguiremos trabajando con un N bajo (N=200) dado que buscamos el mejor valor, es decir el mas eficiente computacionalmente hablando. Repetiremos luego el experimento agrandando el tamano de iteraciones, y probablemente funcione correctamente para todos estos valores de s.



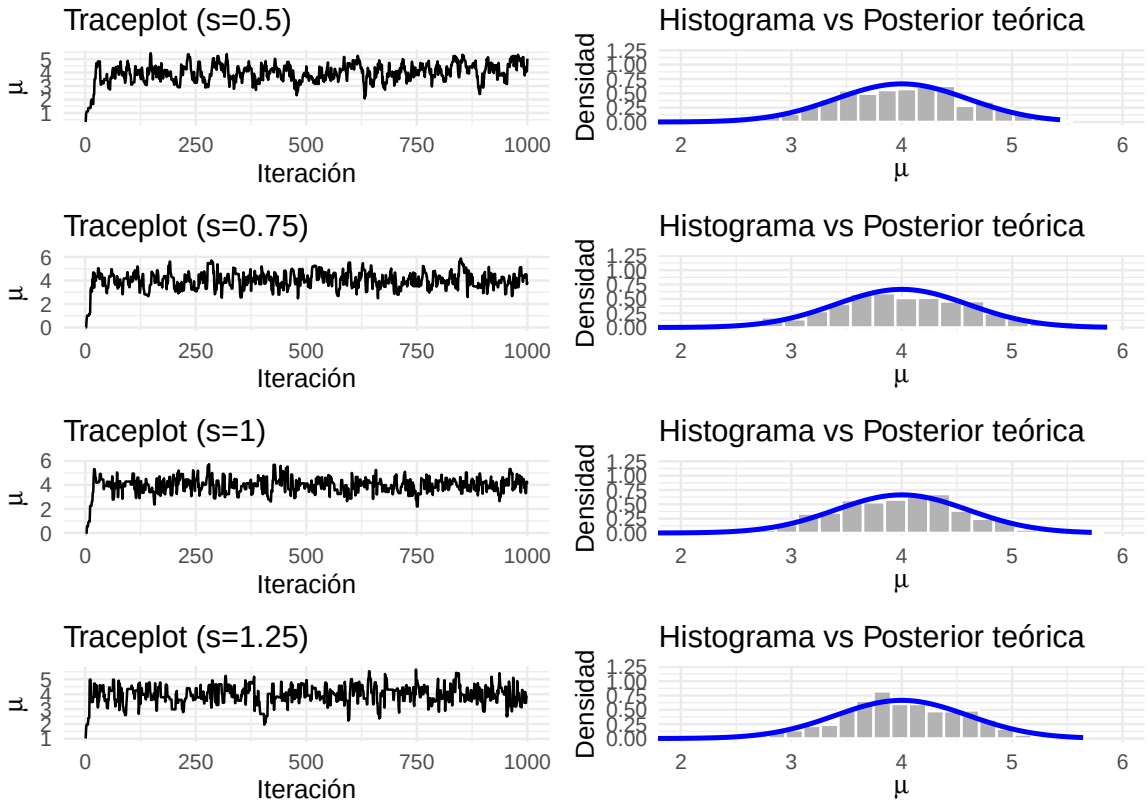
A simple vista los trace_plots son sumamente similares y todos estan bastante correctos. Solamente podemos descartar el grafico de traza de *s=1.25*, dado que tiene algunas mesestas pequenas, pero que no seran problema al agrandar n. Al mirar los histogramas se observa que *s=0.5* subestima levemente a μ.

Es por ello que el mejor valor de s estae entre 0.75 y 1. Ambos histogramas ajustan relativamente bien, aunque cuando *s=1* subestima levemente. En este caso, con esta semilla (84735) yo elegiria 0.75 como valor para s.

Si graficamos con s=0.75 y N=1000, el grafico de traza obtenido es perfecto, se ve el ruido centrado en 4, que es exactamente lo que queriamos. A su vez, el histograma es realmente bueno, dado que ajusta entorno a 4 sin subestimar ni sobreestimar. Solo podriamos reclamarle que no le da tanto peso a los valores extremos, pero en realidad no es algo que nos afecte dado que le da mas importancia al verdadero valor (4).



De todos modos, como mencionabamos anteriormente repetiremos el experimiento agrandando N a 1000. Se observa a simple vista que ajuste es igual de bueno para los distintos valores de s, por lo que cuando el proposal es normal pierde relevancia el valor del desvio, siempre y cuando sea elegido con coherencia.



Ejercicio 3: Modelo Normal-Normal con RStan

Aunque el posterior es analítico en el caso de modelo Normal–Normal, utilizamos RStan para ilustrar y compararemos con la solución cerrada. RStan es la interfaz de Stan en R. Stan es un lenguaje de programacion centrado en inferencia bayesiana que utiliza modelos de MCMC. Entre otros, utiliza una adaptacion del modelo hamiltoniano de Monte Carlo denominado NUTS.

El primer paso es definir el modelo. Utilizaremos la estructura clasica de Stan: definir el recorrido de los datos, los parámetros y por último el modelo:

1. Para definir los datos utilizaremos $(Y_1, Y_2, Y_3, Y_4, Y_5) = (-10.1, 5.5, 0.1, -1.4, 11.5)$, que representa el vector de observaciones Y, dado por la letra.
2. Para los parámetros utilizaremos μ y lo definiremos en $\mathbb{R}^+$, dado que es el recorrido de la distribución Normal.
3. Finalmente definiremos la distribución de Y y μ, con el modelo normal-normal. Sabemos que los datos siguen una distribución Normal $(\mu, 8^2)$ y que $\mu \sim \text{Normal}(-14, 2^2)$.

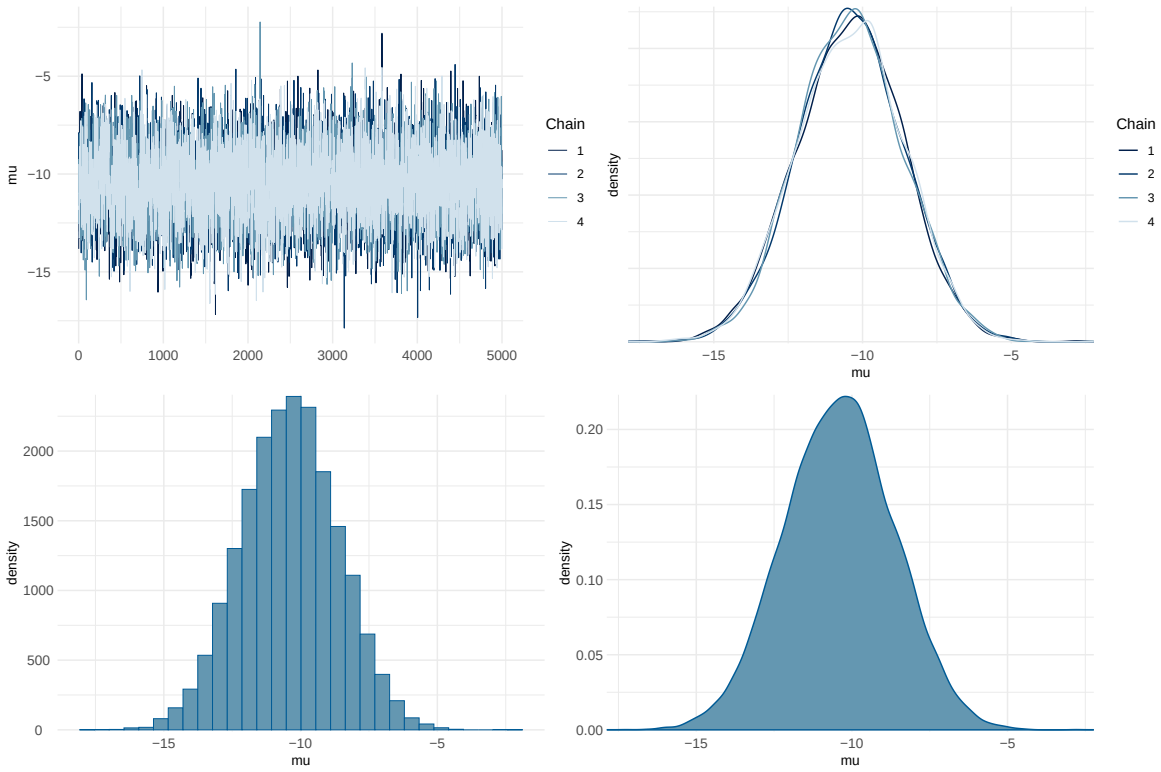
La **definición del modelo** es:

```
y <- c(-10.1, 5.5, 0.1, -1.4, 11.5)
nn_model <- "
  data {
   int<lower=0> N;
   array[N] real Y;
  }
  parameters {
    real mu;
  }
  model {
    mu ~ normal (-14,2) ;
    Y ~ normal (mu, 8);
  }
"
```

El siguiente paso sera utilizar la función **stan** para simular las 4 cadenas y 10.000 iteraciones. Debemos tener en cuenta que el algoritmo descarta la primer mitad de iteraciones realizadas. Es decir que si iteramos 10.000 veces, el algoritmo automaticamente solo se queda con las ultimas 5000. Esto lo realiza para asegurarse que la cadena este ajustada correctamente, es decir que elimina un posible sesgo inicial.

Luego de corridas las cadenas, utilizamos las funciones mcmc_trace para graficar la traza, densoverlay para ver todas las densidades superpuestas, mcmc_hist para calcular el histograma y mcmc_dens para calcular la función de densidad.

A continuación observamos los resultados:



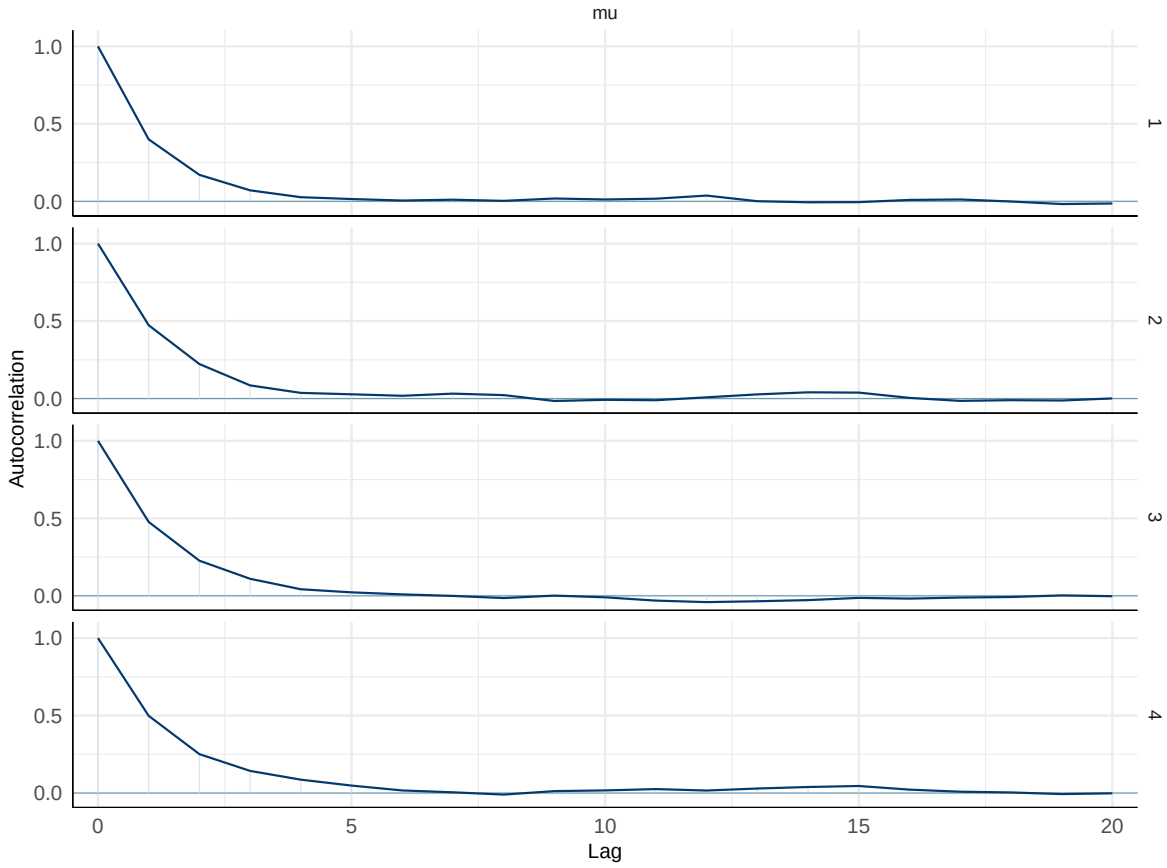
Al observar el **gráfico de trazas**, se aprecia que todas las cadenas se concentran alrededor de -10, con la mayor densidad de valores en el rango aproximado entre -15 y -5. Las trayectorias resultan muy similares entre sí, sin diferencias relevantes más allá de algunos valores atípicos que aparecen de manera ocasional. Dado que estos casos extremos ocurren de forma excepcional, no tienen un impacto significativo en la densidad posterior.

En el conjunto de cadenas, puede afirmarse que estas presentan un comportamiento adecuado: convergen hacia resultados prácticamente idénticos, se centran en torno a -10 con una dispersión muy similar, y sobretodo exhiben el patrón de "ruido" característico de un muestreo MCMC sano.

Si observamos la **densidad**, tal como era de esperar vemos que todas las cadenas generaron distribuciones prácticamente idénticas, con forma aproximadamente normal y centradas en torno a -10, superponiéndose unas con otras. Al analizar en detalle se aprecian pequeñas

diferencias, por ejemplo en la altura máxima de las curvas, pero se trata de variaciones esperables dentro de un contexto de simulación MCMC. En general, el comportamiento de las cadenas sugiere una buena convergencia y da confianza en que la estimación de μ es estable.

A su vez, se realizara una breve revision de las cadenas utilizando la funcion **mcmc_amf**, que grafica la autocorrelacion de las cadenas. Esto es, cuanto se parecen los valores actuales a los valores de hace k pasos atrás (lag k). En otras palabras, sirve para medir la independencia entre valores, y buscamos que sea lo mas cercana a 0 posible. Una autocorrelacion cercana a 1 seria indicio que las nuevas muestras no aportan nueva informacion nueva, por lo que la cadena avanzaria lento.



En el momento 0 la autocorrelacion siempre es igual 1 (cada valor se autorelaciona consigo mismo). Esto ocurre siempre para todas las cadenas posibles. Para 1 lag de 1, la autocorrelacion es menor a 0.5 en todos los casos, lo que es un indicio fuerte de que la cadena no tiene autorrelacion. En todas las cadeas para un lag de 4 vale practicamente 0, lo que es evidencia mas que suficiente de que las cadenas se mezclaron de manera correcta. Podemos interpretarlo como que la cadena ya se olvido casi completamente lo que paso 4 momentos atras.

Si en vez de visual preferimos usar evidencia numerica, podemos calcular el **neff_ratio** y el **rhat**, que son medidas usuales para medir la eficiencia y el funcionamiento de las cadenas.

El **neff_ratio** es una medida que mira la independencia entre valores. Tiene que ver con la autocorrelacion de la cadena, y buscamos que sea lo mas alta posible. Ademas es util para estimar cuantos de mis valores simulados son efectivamente independientes.

En este caso no contamos explicitamente con el neff_ratio, sino que lo calcularemos calculando el ess_bulk sobre n. El **ess_bulk** es la estimacion la cantidad de valores que fueron muestreados de forma independiente:

$$\text{ESS ratio} = \frac{N_{\text{eff}}}{N} = \frac{7162}{20000} = 0.3581$$

Se observa que **neff_ratio**=0.3581. Es decir que, se estima que casi el 36% de los valores son muestreados de manera independiente, o viendolo en numeros, 7612 valores aprox fueron aleatorios. Usualmente se dice que: Valores superiores a 0.3 implican una convergencia razonable, por lo que la cadena aprueba el test.

Si hubiesemos obtenido valores menores a 0.1 deberiamos preocuparnos, implicaria que menos del 10% de los valores son aleatorios y por ende la cadena se estaria moviendo muy lento.

El **rhat** es una medida que compara la convergencia entre cadenas. Basicaemtne compara la convergencia dentro de cada una de las cadenas con la variabilidad entre ellas. Si todas las cadenas convergieron a lo mismo, que es lo que queremos, el rhat sera aproximadamente 1.

Una regla usual es:

- $\hat{R} \approx 1.00 \quad \Rightarrow \quad$ convergencia buena.
- $\hat{R} > 1.01 \quad \Rightarrow \quad$ sospecha de que las cadenas no convergieron.
- $\hat{R} > 1.1 \quad \Rightarrow \quad$ definitivamente problemático.

En nuestro caso el rhat es muy cercano a 1, por lo que confirma que la convergencia es buena.

Table 8: Rhat y ess_bulk para mu

variable	rhat	ess_bulk
mu	1.000706	7162.391

Parte C: Busqueda del valor más plausible

En el grafico a simple vista se observa que la mediana es apenas menor que -10. Podriamos pensar que es aproximadamente -10.5. Por ende, por ser el valor con mas densidad podemos decir que este sera el valor mas plausible para μ.

Si convertimos los valores obtenidos en la cadena en un dataframe, luego podremos calcular las medidas de resumen con la funcion summarise, obteniendo lo siguiente:

Table 9: Resumen de los datos generados con las cadenas

variable	mean	median	sd
mu	-10.40401	-10.38745	1.766311

De esta manera, se observa que las cadenas otorgan un valor mas probable para μ que es aproximadamente -10.4, con un desvio estandar igual a 1.76.

Este valor es relativamente cercano a nuestro prior, el cual tenia media en -14. Se aleja bastante de la verosimilitud , cuya media de los datos es:

Table 10: Media de los datos observados

x
1.12

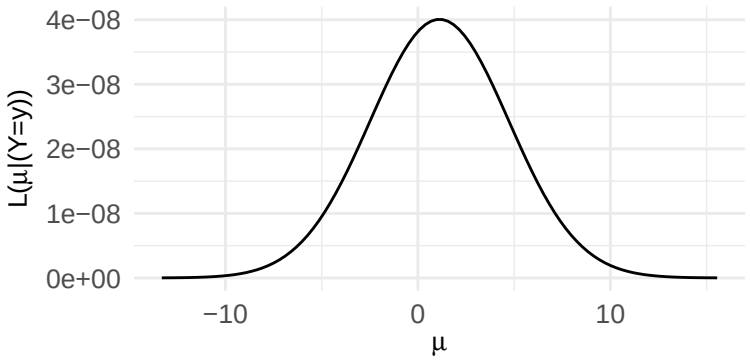
Profundizaremos este analisis en la siguiente seccion.

Parte D: Comparacion entre posterior teorica y aproximacion MCMC.

Realizaremos el calculo del posterior teorico con la funcion summarize_normal_normal, del paquete bayesrules, que a partir del prior y la verosimilitud, devuelve la distribucion a posterior.

En el caso del modelo normal normal, en la verosimilitud no cargaremos el vector de observaciones como hicimos con el modelo beta binomial, sino que le pasaremos la media de los datos observados, el desvio y el total de observaciones. El total de observaciones es 5, el desvio es un dato de letra y la media ya fue calculada en el ejercicio anterior (mean(y)=1.12). Por ultimo, tambien le debemos pasar la informacion del prior, que tambien es un dato de letra del ejercicio.

La verosimilitud se visualiza de la siguiente manera, estando centrada en 1.12, tal como esperabamos.



Definimos la funcion para calcular el posterior de la siguiente manera:

```
postnn<-summarize_normal_normal(
  mean = -14, sd = 2,    # prior
  sigma = 8,             # desviación dada por letra de los datos
  y_bar = mean(y),       # media muestral
  n = length(y)          # tamaño de muestra
)
```

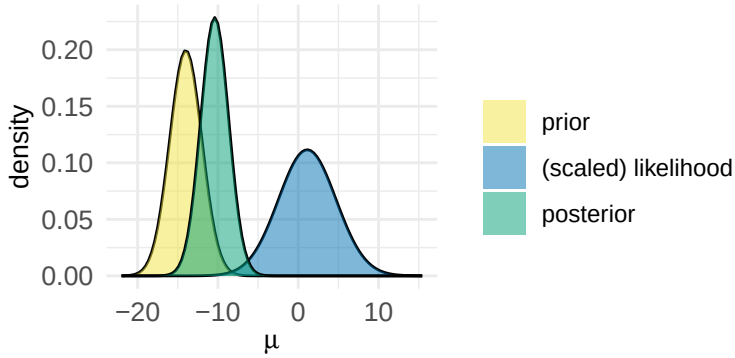
De esta manera el posterior es:

```
kable(postnn, digits = 3)
```

model	mean	mode	var	sd
prior	-14.0	-14.0	4.000	2.000
posterior	-10.4	-10.4	3.048	1.746

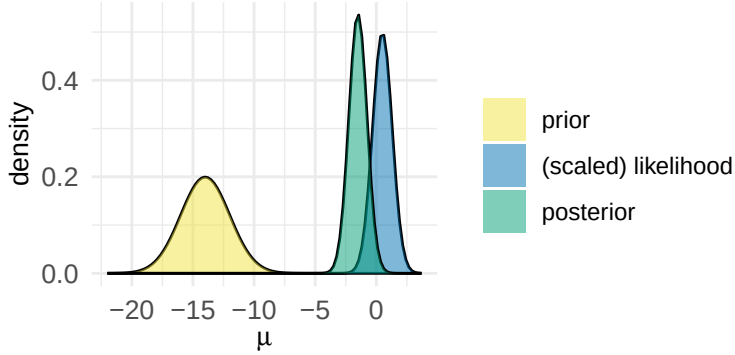
El posterior que simulamos distribuye igual que el posterior teorico. Tanto la media como la varianza convergen a la correcta.

Podemos visualizar todo junto con la funcion plot_normal_normal:



Se observa que el prior, al ser mas concentrado que la verosimilitud, tiene mas peso sobre el posterior. Sumado a esto estamos trabajando con N=5, que es bajo, por lo que los datos toman menos peso que la distribucion a priori.

Por ejemplo, si repetimos el experimento con la misma media y N=100, obtenemos un posterior mucho mas cercano a la verosimilitud dado que estos toman mayor relevancia:



Ejercicio 4: Modelo Beta Geometrico

Parte A: Deduccion del modelo

En estadistica, mas precisamente en el campo del conteo, la **distrbucion geometrica** es una de las distribuciones fundamental. En particular, se utiliza cuando nos interesa el número de intentos necesarios hasta obtener el primer éxito en una secuencia de Bernoulli's independientes con probabilidad de éxito θ. Si bien existen dos definiciones validas para el modelo geometrico, utilizaremos "Y= *Numero de fracasos hasta el primer exito*", en lugar de usar la definicion que refiere a los fracasos *antes* del exito. A su vez, debemos recordar que la distribucion geometrica es el caso particular de 1 exito en la **distribucion binomial negativa**.

Dado que y representa el numero de intentos hasta el primer exito, entonces:

$$P(Y = y \mid \theta) = (1-\theta)^{y-1}\,\theta$$

Donde:

- $(1-\theta)^{y-1}$ representa la probabilidad de observar y fracasos iniciales,
- θ es la probabilidad de éxito en cada ensayo.

De esta manera, en el contexto bayesiano, consideramos el parametro θ desconocido y le asignamos una distribucion a priori.

Al igual que en el modelo *beta-binomial*, el modelo beta-geometrico sera util para modelar la probabilidad de exito en un ensayo Bernoulli. Lo utilizaremos cuando los datos, en vez de venir como numero de exitos en n pruebas, vengan como numero de pruebas hasta un exito. Ademas de su diferencia conceptual, en el modelo beta-binomial el tamano de los datos es fijo y varia la cantidad de exitos, mientras que en el modelo beta-geometrico se fijan los exitos (1) y lo que varia es el tamano muestral.

Dada la flexibilidad que otorga el modelo beta para modelar proporciones, dado que vive en el intervalo [0,1] y dado que la distribucion beta es conjugada de la binomial negativa, elegiremos la **distribucion beta como prior** para la geometrica.

Entonces:

- Prior:

$$\pi(\theta) = \frac{1}{B(\alpha,\beta)}\,\theta^{\alpha-1}(1-\theta)^{\beta-1}, \quad 0<\theta<1$$

$$\pi(\theta) \propto \theta^{\alpha-1}(1-\theta)^{\beta-1}$$

- Verosimilitud

$$L(\theta \mid y) = \prod_{i=1}^{n} P(Y_i = y_i \mid \theta) = \prod_{i=1}^{n} (1-\theta)^{y_i - 1}\theta = \theta^n (1-\theta)^{\sum_{i=1}^{n}(y_i - 1)} \propto \theta^n (1-\theta)^{\sum y_i - n}$$

- Recordando el Teormea de Bayes:

$$\pi(\theta \mid y) \propto \pi(\theta) \cdot L(\theta \mid y)$$

- Entoncesel kernell obtenido es:

$$\pi(\theta \mid y) \propto \pi(\theta)\, L(\theta \mid y) = \left[\theta^{\alpha}(1-\theta)^{\beta}\right]\left[\theta^n (1-\theta)^{\sum y_i - n}\right] = \theta^{\alpha+n}\,(1-\theta)^{\beta + \sum y_i - n}.$$

- Y corresponde a la distribucion:

$$\theta \mid y \sim \text{Beta}\left(\alpha + n,\ \beta + \sum_{i=1}^{n} y_i - n\right)$$

De esta manera, concluimos que el modelo beta es conjugado de la geometrica, dado que ambos pertenecen a la misma familia de distribuciones (beta).

Parte B: Ejemplo con prior $\theta \sim$ Beta$(2, 5)$ y datos $(3, 5, 1, 3, 4, 5)$

En primer lugar definiremos en R el siguiente vector de observaciones:

$$(Y_1, Y_2, Y_3, Y_4, Y_5, Y_6) = (3, 5, 1, 3, 4, 5)$$

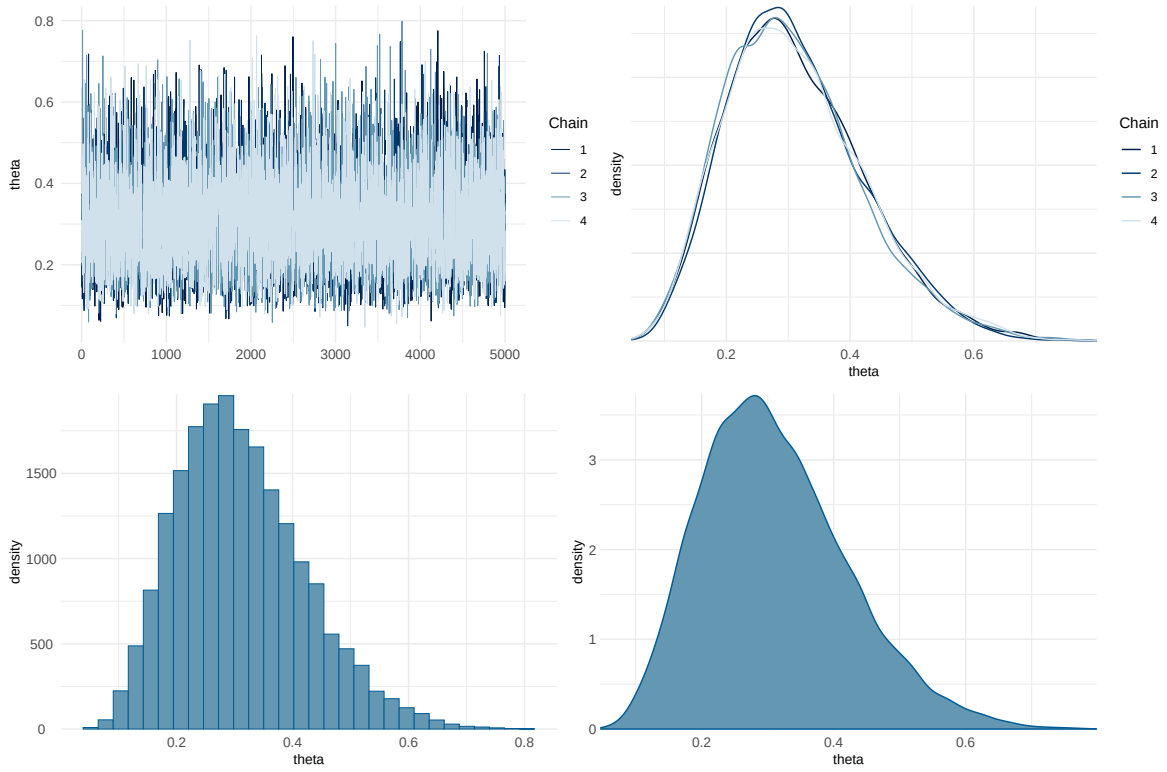
Luego, al igual que hicimos en el ejercicio anterior, definiremos el modelo de rstan. Dado que en Stan no existe una funcion para la geometrica, utilizaremos su version binomial negativa para 1 exito.

```
n <- length(y)
bg_model <- "
data {
  int<lower=0> N;
   array[N] int<lower=0> Y;
  }
parameters {
  real<lower=0,upper=1> theta;
}
model {
  Y ~ neg_binomial (1 , theta);
  theta ~ beta(2, 5);
}
"
```

El siguiente paso es utilizar la función stan para simular las 4 cadenas.

Utilizamos las funciones mcmc_trace para graficar la traza, densoverlay para ver todas las densidades superpuestas, mcmc_hist para calcular el histograma y mcmc_dens para calcular la función de densidad.

A continuación observamos los resultados de la simulación:



El grafico de traza muestra claramente que se mezclaron de buena manera los valores: Las trazas oscilan de forma estable. Asimismo, es casi imposible distinguir las cadenas, ya que están todas superpuestas. Esto es ideal, ya que todas las cadenas independientes nos dan resultados parecidos, a pesar de haber empezado de puntos diferentes.

Al mirar la densidad, las graficas de las cuatro cadenas estan superpuestas. Esto confirma que, lo que vimos en el gráfico de las trazas, es correcto: La distribución final es casi igual para las cuatro cadenas. Analizando el gráfico concluimos que el valor más plausible de θ es aproximadamente 0.28.

Utilizaremos la funcion summarise, del paquete posterior, que es muy utilizado en inferencia bayesiana, para calcular las principales medidas de tendencia centra, el intervalo 90% de confianza para θ, el rhat, el ess_bulk, entre otros.

Table 12: Resumen para theta

variable	mean	median	sd	mad	q5	q95	rhat	ess_bulk	ess_tail
theta	0.3109342	0.2982743	0.110618	0.1097559	0.1509181	0.5126082	1.000315	7581.106	9202.841

A modo de comentario, reafirmamos la correcta convergencia de las cadenas: el **rhat** (muy cercano a 1) y el **ess_bulk** (ESS ratio $= \frac{N_{\text{eff}}}{N} = \frac{7162}{20000} = 37.5$) es superior a 0.3.

Mirando las medidas de tendencia central, vemos que la media y mediana confirman nuestras sospechas respecto a que el valor mas plausible para θ ronda 0.28. A su vez, que la media y mediana sean tan similares son un indicio que no hay colas pesadas hacia ninguno de los dos extremos.

Profundizando sobre el **intervalo de confianza 90%** vemos que, con 90% de probabilidad, el valor de θ, para este ejemplo, estara en el intervalo [0.146 , 0.500]. De esta manera, el intervalo [0 , 0.5] acumula el 95% de probabilidad, y si recordamos el significado de θ, podemos afirmar con una confianza de 95% que el verdadero valor de θ es menor a 0.5, es decir que muy probablemente la proporcion de exitos es peuena

Profundizando en el intervalo de confianza 90%, observamos que, con un 90% de probabilidad a posteriori, el valor de θ se encuentra en el intervalo [0.146, 0.500]. Esto implica que los valores de θ superiores a 0.5 son muy poco plausibles bajo los datos y el modelo. De hecho, el intervalo [0, 0.5] concentra aproximadamente el 95% de probabilidad a posteriori.

Recordando que θ representa la probabilidad de éxito en un ensayo Bernoulli, esta evidencia sugiere con alta confianza que θ es relativamente pequeña. En otras palabras, la proporción de éxitos es baja, y es probable que se requieran mas de un intento antes de observar un éxito.

Calculo del posterior Teorico.

Dado que definimos el modelo en la parte A), podemos calcular el posterior de manera teorica para luego poder graficarlo y compararlo con las cadenas simuladas por RStan.

Recordemos que:

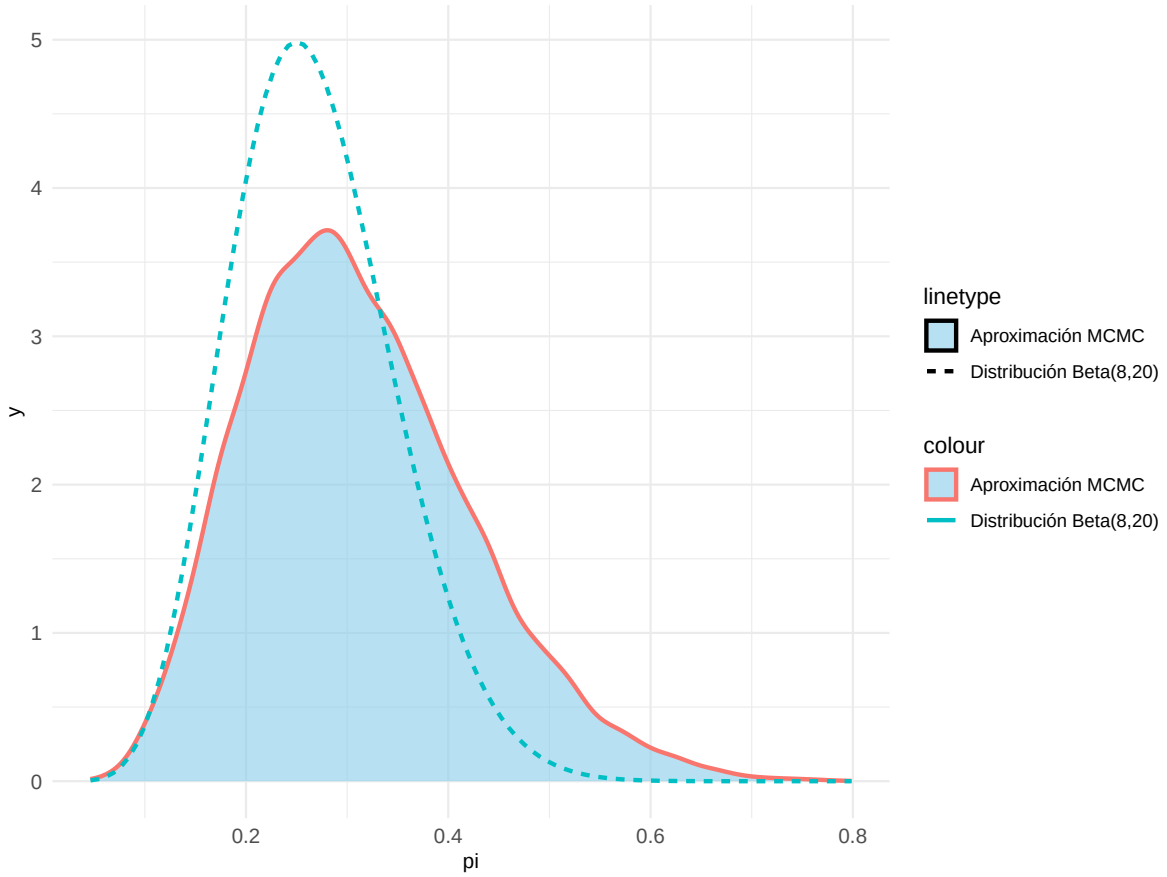
- Prior: $\theta \sim \text{Beta}(2, 5)$
- $n = 6$
- Verosimilitud: $(Y_1, Y_2, Y_3, Y_4, Y_5, Y_6) = (3, 5, 1, 3, 4, 5)$
- Entonces: $\sum y_i = 3 + 5 + 1 + 3 + 4 + 5 = 21$

Por ende

$$\theta \mid y \sim \text{Beta}\left(\alpha + n,\ \beta + \sum_{i=1}^{n} y_i - n\right) \sim \text{Beta}(2+6, 5+21-6) \sim \text{Beta}(8, 20)$$

Extraeremos los valores de las cadenass y graficaremos la densidad obtenida. Lo realizaremos con ggplot para poder superponer la distrbucion teorica.

Si bien la aproxumacion simulada es buena, en este caso no es perfecta. Como caracteristica positiva se destaca que el posterior simulado esta correctamente centrado al posterior teorico. A pesar de esto, tal como se ve en el grafico siguiente, se observa que el las cadenas MCMC sobreestiman levemente a θ. A su vez, las mismas son levemente mas dispersas, en especial la cola de la derecha.



De esta manera, la aproximación MCMC reproduce bien la forma general de la posterior teórica y está centrada en torno al valor esperado.

Fuentes:

https://www.ibm.com/es-es/topics/monte-carlo-simulation

https://www.ingenieria.unam.mx/javica1/ingsistemas2/Simulacion/Cadenas_de_Markov.htm

https://rpubs.com/ROARMarketingConcepts/1063733

https://pmc.ncbi.nlm.nih.gov/articles/PMC9788645/

Anexo

Simulación de las cadenas de markov (ejercicio 3):

```
y <- c(0,1,0)

gp_sim <-stan(
  model_code = nn_model,
  data = list(N = length(y), Y=y),
  chains = 4,
  iter = 5000*2,
  seed = 84735)
```

```
SAMPLING FOR MODEL 'anon_model' NOW (CHAIN 1).
Chain 1:
Chain 1: Gradient evaluation took 3e-06 seconds
Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.03 seconds.
Chain 1: Adjust your expectations accordingly!
Chain 1:
Chain 1:
Chain 1: Iteration:    1 / 10000 [  0%]  (Warmup)
Chain 1: Iteration: 1000 / 10000 [ 10%]  (Warmup)
Chain 1: Iteration: 2000 / 10000 [ 20%]  (Warmup)
Chain 1: Iteration: 3000 / 10000 [ 30%]  (Warmup)
Chain 1: Iteration: 4000 / 10000 [ 40%]  (Warmup)
Chain 1: Iteration: 5000 / 10000 [ 50%]  (Warmup)
Chain 1: Iteration: 5001 / 10000 [ 50%]  (Sampling)
Chain 1: Iteration: 6000 / 10000 [ 60%]  (Sampling)
Chain 1: Iteration: 7000 / 10000 [ 70%]  (Sampling)
Chain 1: Iteration: 8000 / 10000 [ 80%]  (Sampling)
Chain 1: Iteration: 9000 / 10000 [ 90%]  (Sampling)
Chain 1: Iteration: 10000 / 10000 [100%]  (Sampling)
Chain 1:
Chain 1:  Elapsed Time: 0.019 seconds (Warm-up)
Chain 1:                0.021 seconds (Sampling)
Chain 1:                0.04 seconds (Total)
Chain 1:

SAMPLING FOR MODEL 'anon_model' NOW (CHAIN 2).
Chain 2:
Chain 2: Gradient evaluation took 4e-06 seconds
```

```
Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.04 seconds.
Chain 2: Adjust your expectations accordingly!
Chain 2:
Chain 2:
Chain 2: Iteration:    1 / 10000 [  0%]  (Warmup)
Chain 2: Iteration: 1000 / 10000 [ 10%]  (Warmup)
Chain 2: Iteration: 2000 / 10000 [ 20%]  (Warmup)
Chain 2: Iteration: 3000 / 10000 [ 30%]  (Warmup)
Chain 2: Iteration: 4000 / 10000 [ 40%]  (Warmup)
Chain 2: Iteration: 5000 / 10000 [ 50%]  (Warmup)
Chain 2: Iteration: 5001 / 10000 [ 50%]  (Sampling)
Chain 2: Iteration: 6000 / 10000 [ 60%]  (Sampling)
Chain 2: Iteration: 7000 / 10000 [ 70%]  (Sampling)
Chain 2: Iteration: 8000 / 10000 [ 80%]  (Sampling)
Chain 2: Iteration: 9000 / 10000 [ 90%]  (Sampling)
Chain 2: Iteration: 10000 / 10000 [100%]  (Sampling)
Chain 2:
Chain 2:  Elapsed Time: 0.019 seconds (Warm-up)
Chain 2:                0.019 seconds (Sampling)
Chain 2:                0.038 seconds (Total)
Chain 2:

SAMPLING FOR MODEL 'anon_model' NOW (CHAIN 3).
Chain 3:
Chain 3: Gradient evaluation took 3e-06 seconds
Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0.03 seconds.
Chain 3: Adjust your expectations accordingly!
Chain 3:
Chain 3:
Chain 3: Iteration:    1 / 10000 [  0%]  (Warmup)
Chain 3: Iteration: 1000 / 10000 [ 10%]  (Warmup)
Chain 3: Iteration: 2000 / 10000 [ 20%]  (Warmup)
Chain 3: Iteration: 3000 / 10000 [ 30%]  (Warmup)
Chain 3: Iteration: 4000 / 10000 [ 40%]  (Warmup)
Chain 3: Iteration: 5000 / 10000 [ 50%]  (Warmup)
Chain 3: Iteration: 5001 / 10000 [ 50%]  (Sampling)
Chain 3: Iteration: 6000 / 10000 [ 60%]  (Sampling)
Chain 3: Iteration: 7000 / 10000 [ 70%]  (Sampling)
Chain 3: Iteration: 8000 / 10000 [ 80%]  (Sampling)
Chain 3: Iteration: 9000 / 10000 [ 90%]  (Sampling)
Chain 3: Iteration: 10000 / 10000 [100%]  (Sampling)
Chain 3:
Chain 3:  Elapsed Time: 0.02 seconds (Warm-up)
```

```
Chain 3:                0.022 seconds (Sampling)
Chain 3:                0.042 seconds (Total)
Chain 3:

SAMPLING FOR MODEL 'anon_model' NOW (CHAIN 4).
Chain 4:
Chain 4: Gradient evaluation took 3e-06 seconds
Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 0.03 seconds.
Chain 4: Adjust your expectations accordingly!
Chain 4:
Chain 4:
Chain 4: Iteration:    1 / 10000 [  0%]  (Warmup)
Chain 4: Iteration: 1000 / 10000 [ 10%]  (Warmup)
Chain 4: Iteration: 2000 / 10000 [ 20%]  (Warmup)
Chain 4: Iteration: 3000 / 10000 [ 30%]  (Warmup)
Chain 4: Iteration: 4000 / 10000 [ 40%]  (Warmup)
Chain 4: Iteration: 5000 / 10000 [ 50%]  (Warmup)
Chain 4: Iteration: 5001 / 10000 [ 50%]  (Sampling)
Chain 4: Iteration: 6000 / 10000 [ 60%]  (Sampling)
Chain 4: Iteration: 7000 / 10000 [ 70%]  (Sampling)
Chain 4: Iteration: 8000 / 10000 [ 80%]  (Sampling)
Chain 4: Iteration: 9000 / 10000 [ 90%]  (Sampling)
Chain 4: Iteration: 10000 / 10000 [100%]  (Sampling)
Chain 4:
Chain 4:  Elapsed Time: 0.019 seconds (Warm-up)
Chain 4:                0.019 seconds (Sampling)
Chain 4:                0.038 seconds (Total)
Chain 4:
```