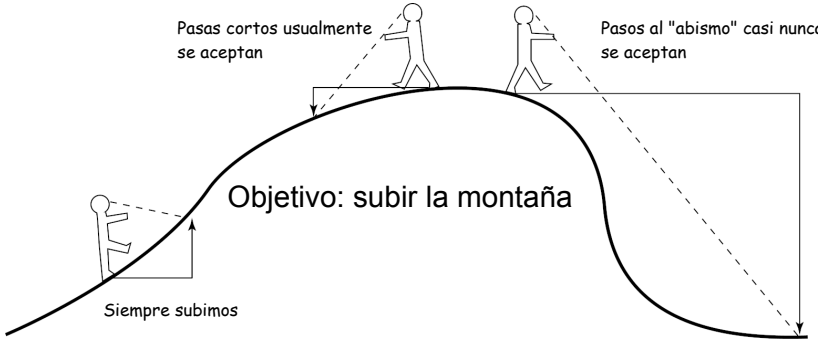
MÉTODOS COMPUTACIONALES PARA OBTENER LA POSTERIOR

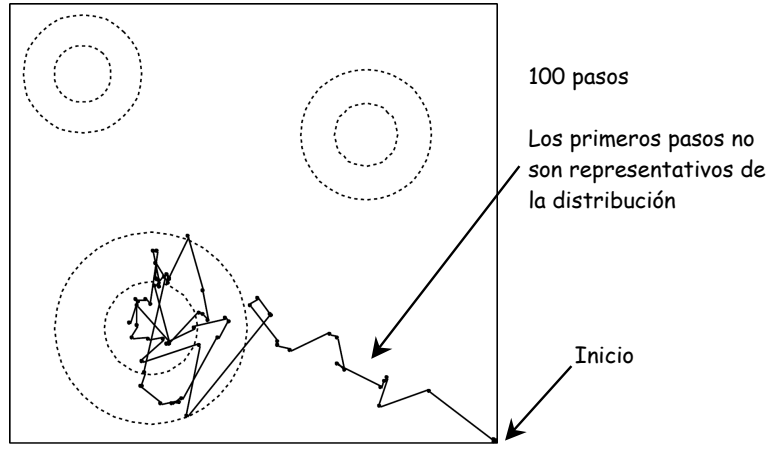
Santiago Alonso-Díaz, PhD
Universidad Javeriana

Ideal: soluciones analíticas (formulas)
Realidad: muy difícil de conseguir
Solución: métodos computacionales (algoritmos)

Robot Sampleador



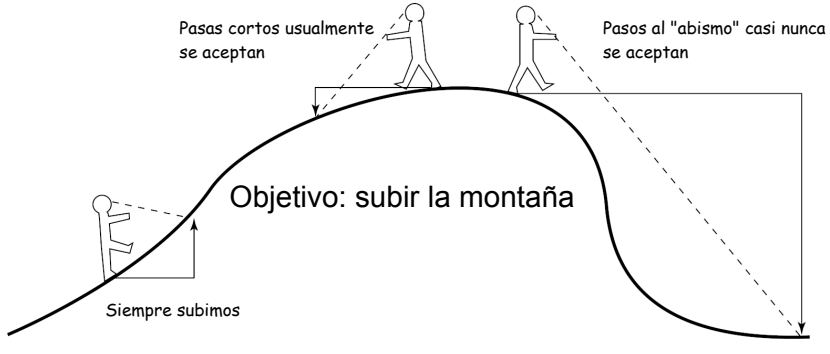
Panorama desde arriba



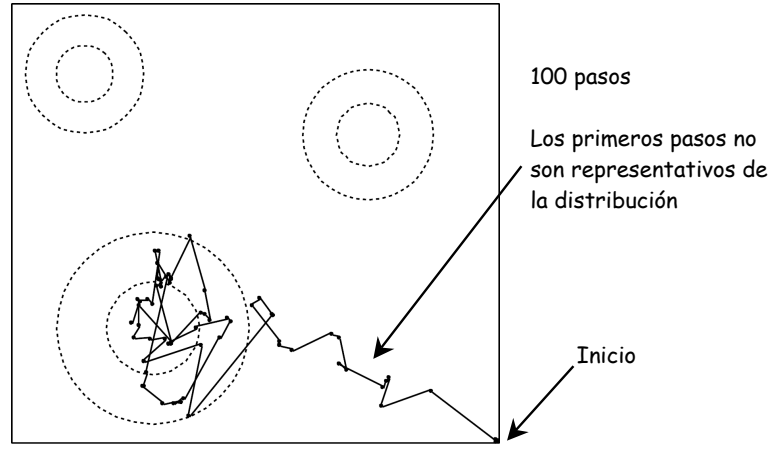
Fuente: Paul O. Lewis

Es una métafora. Muchas preguntas:
¿Qué es la montaña?
¿Qué entiende el algoritmo por "abismo"?
¿Dónde se inicia?
¿Tiene que ir a las otras montañas?

Robot Sampleador



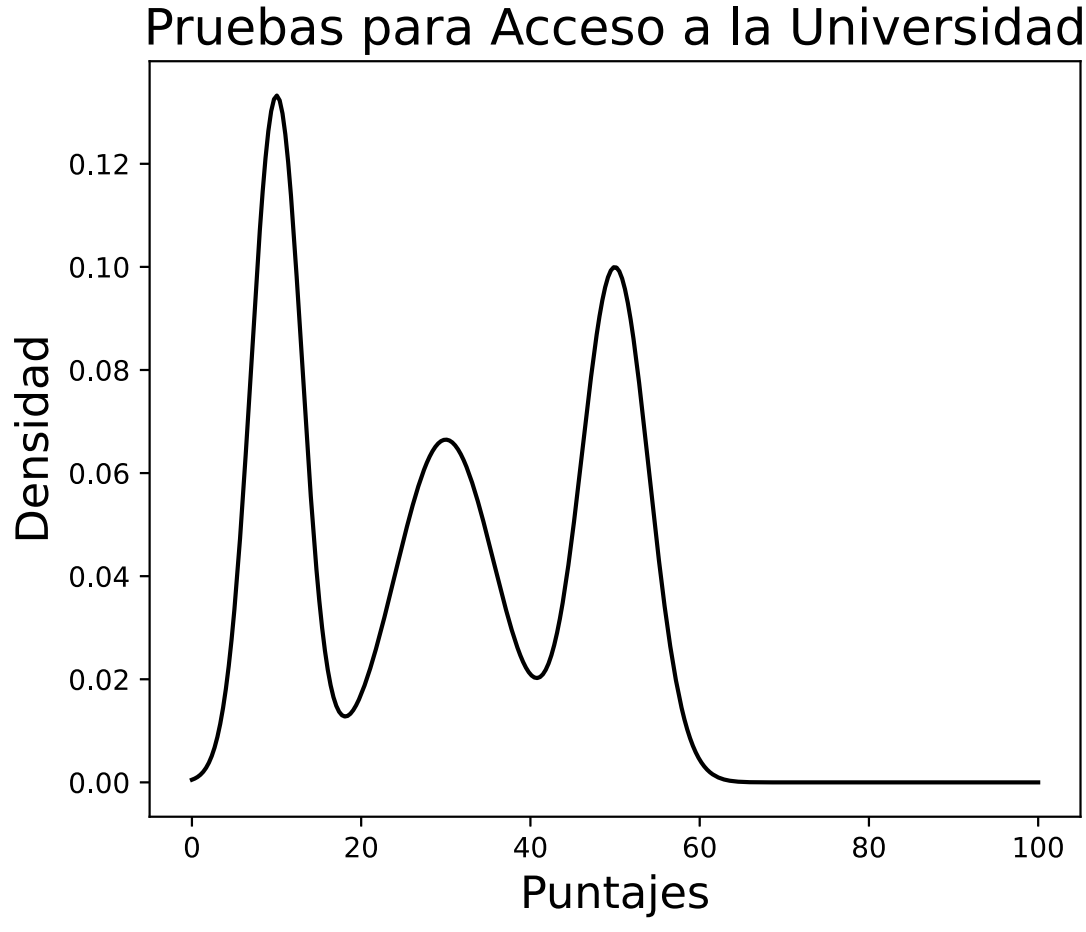
Panorama desde arriba



Fuente: Paul O. Lewis

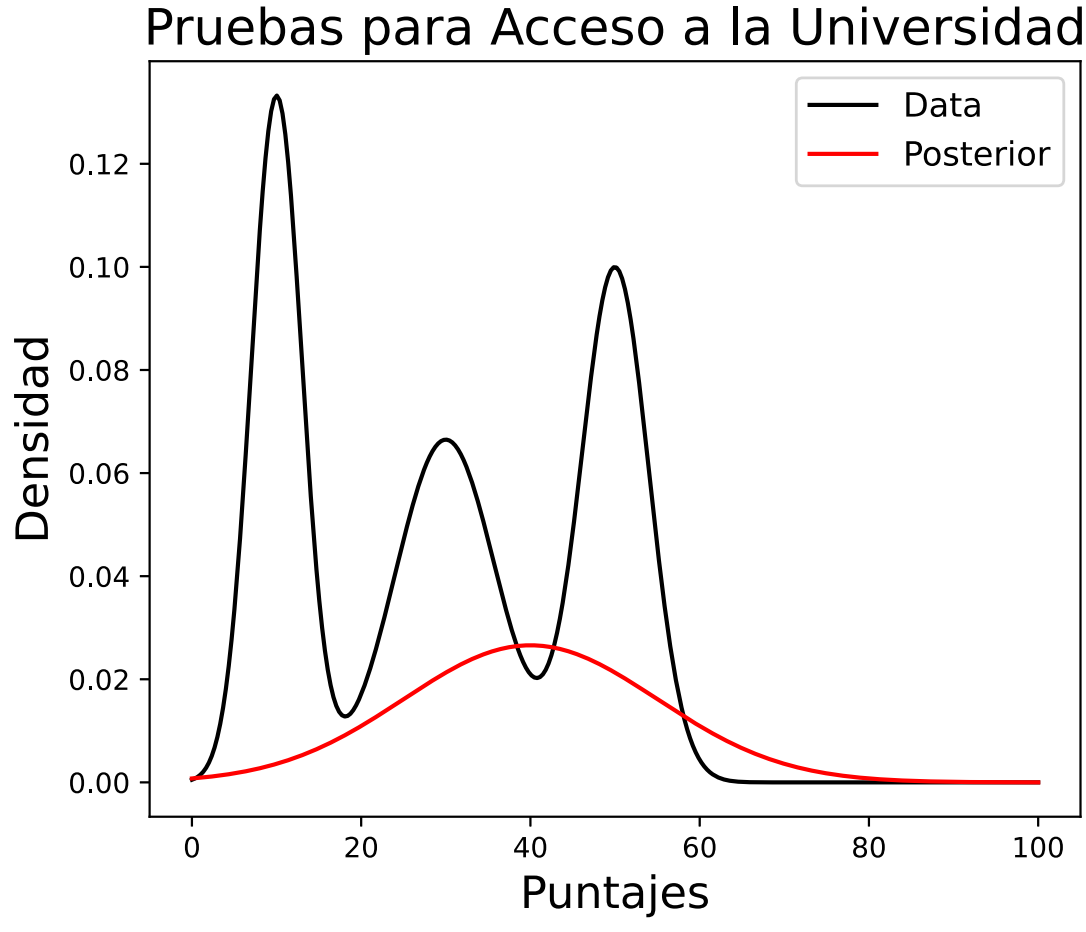
INTRO. MARKOV

La data puede no ser normal. Por ejemplo, con varias modas:



Es data hipótetica, pero ¿qué piensan que puede ser cada moda?

¿Qué pasa si imponemos un modelo normal no jerárquico?



No sirve. Y obtener una expresión matemática puede no ser fácil.

Métodos computacionales nos pueden ayudar a pensar modelos más elaborados.

Vamos a centrarnos en algoritmos `MCMC` .

Primero recordemos que significa que un proceso sea de tipo `Markov`

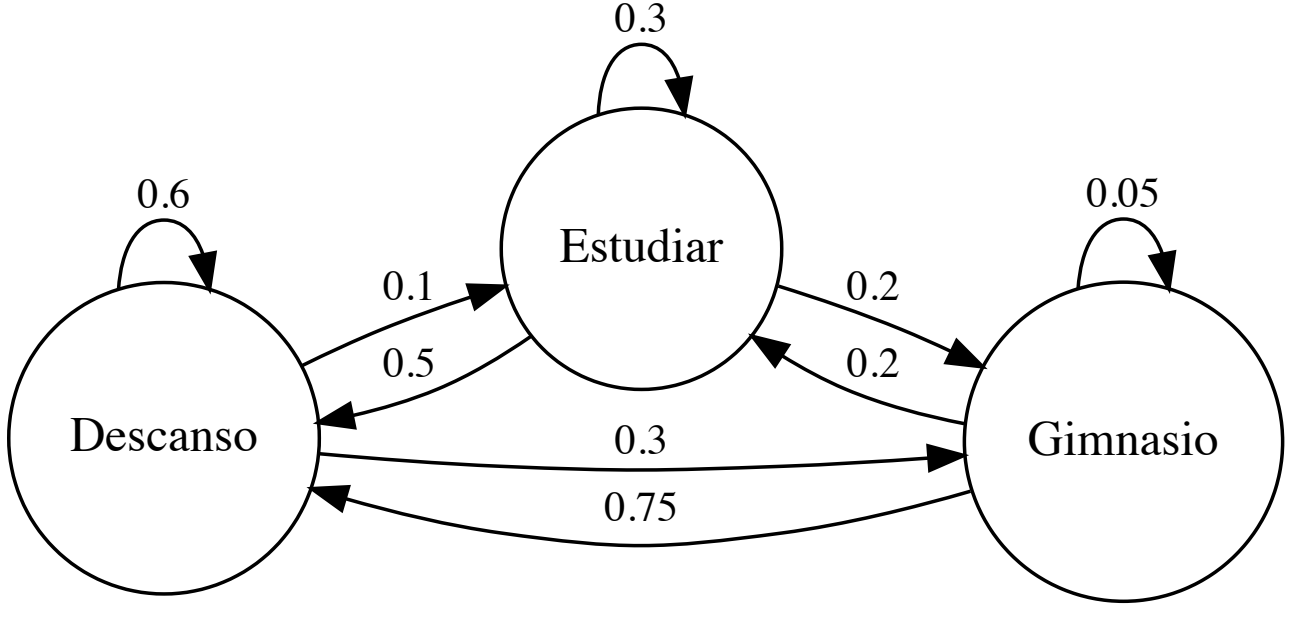
Proceso Markov (s: state):

$$p(s_1, s_2, s_3, \ldots, s_n) = p(s_1)p(s_2|s_1)p(s_3|s_2)\ldots p(s_n|s_{n-1})$$

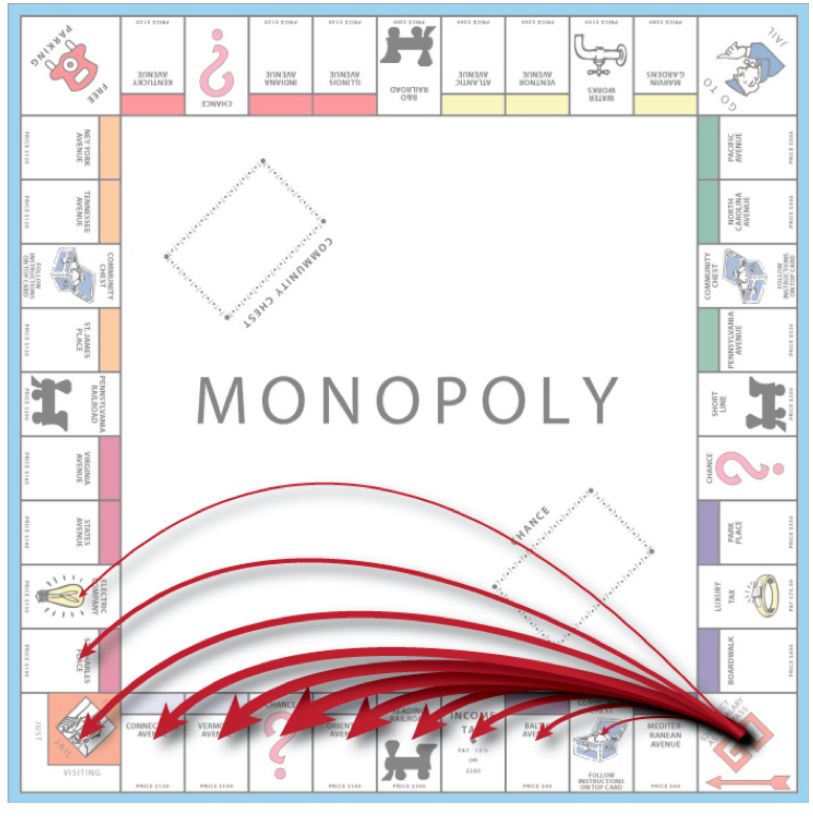
En lenguaje natural:

No hay memoria de toda la secuencia. Solo hay acceso al estado actual para determinar la probabilidad del siguiente estado.

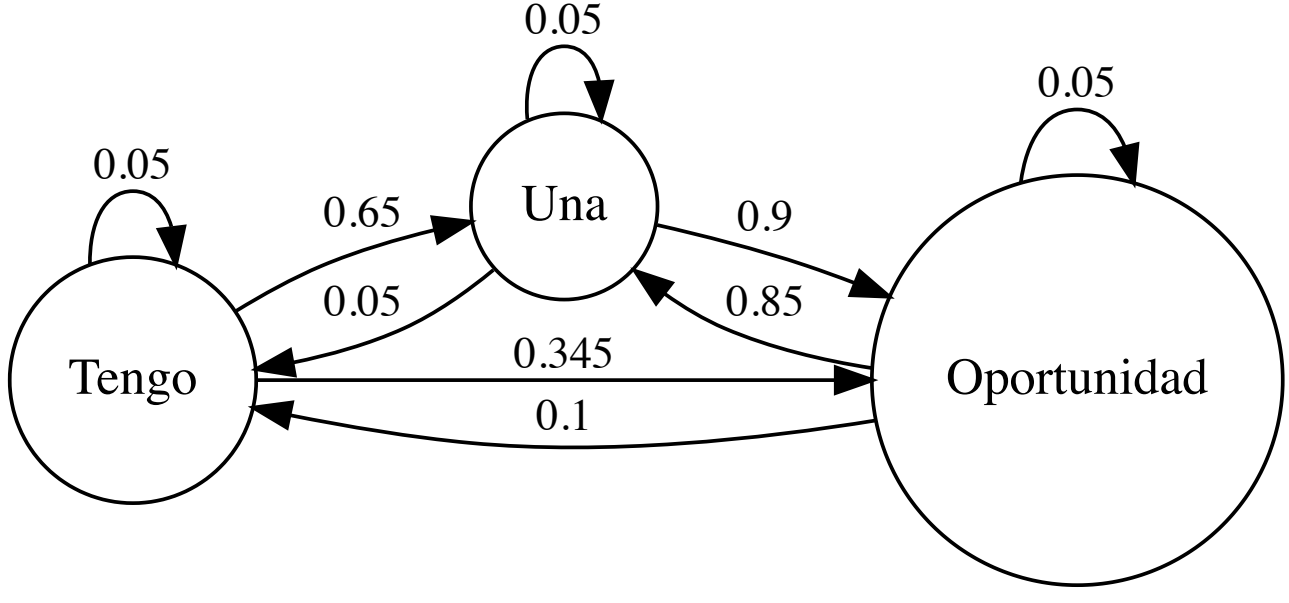
Todo proceso Markoviano tiene transiciones de probabilidad de un estado al otro



Otro ejemplo



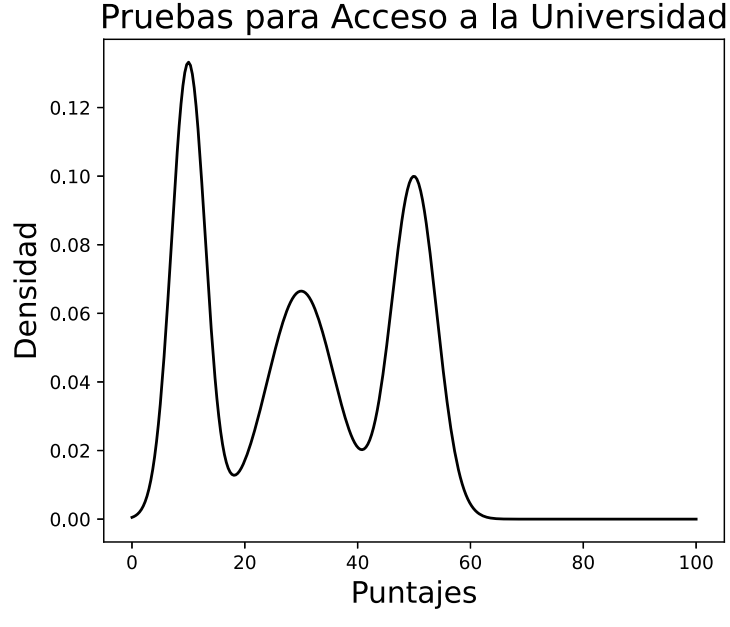
Otro ejemplo ... debate: ¿es el lenguaje Markoviano?



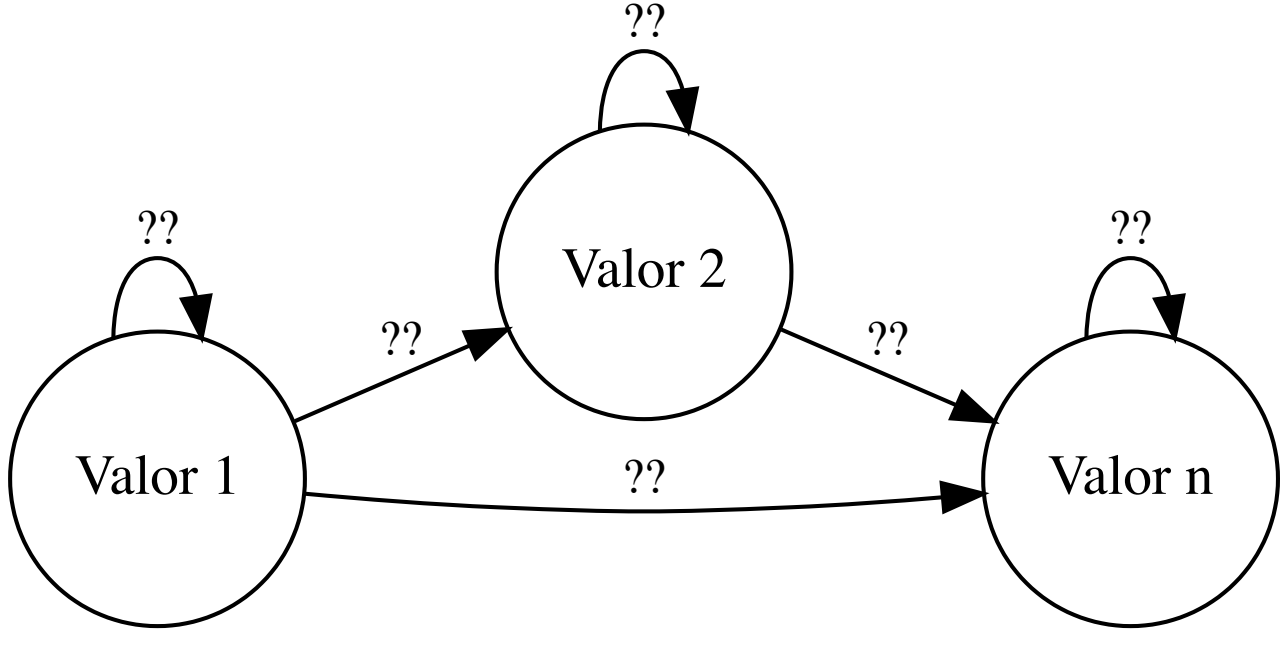
BAYES Y MARKOV (MCMC)

¿Qué tiene que ver esto con una posterior Bayesiana?

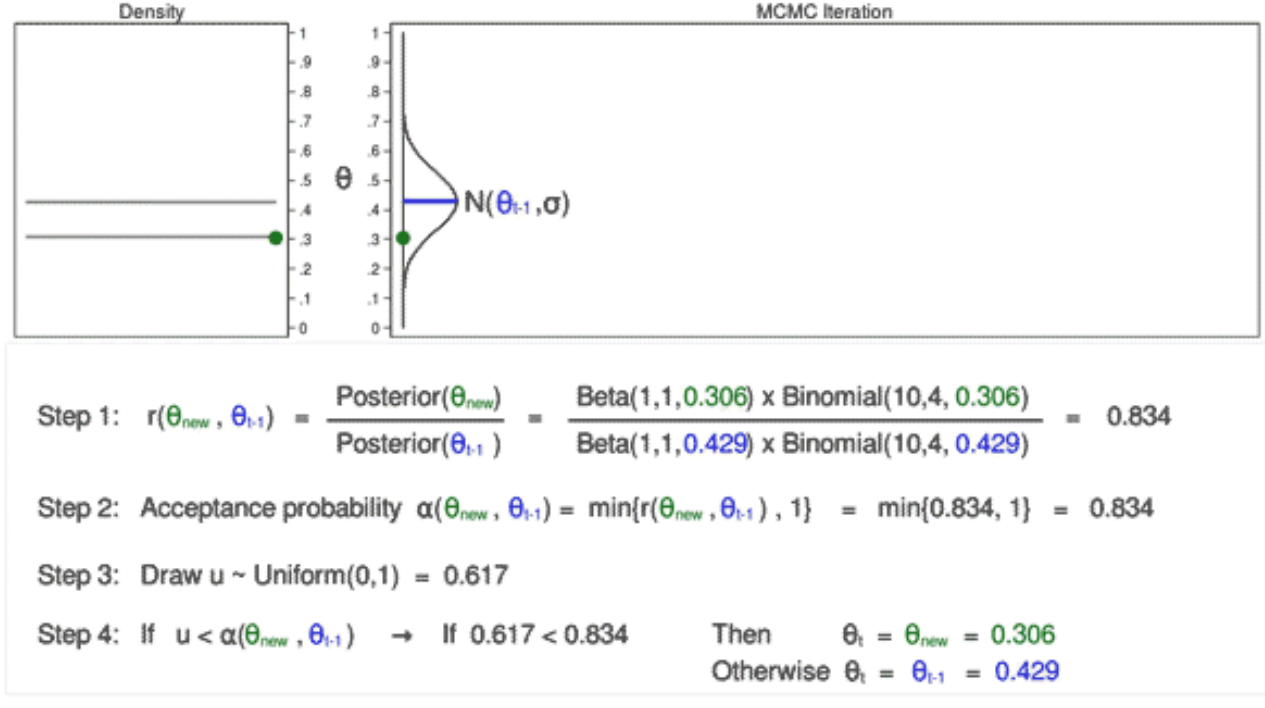
Un modelo de desigualdad en educación en puntajes debería incluir variables latentes escasas (Quarles, 2020): recursos sociales, académicos, emocionales, y económicos.
¿Como tomar una muestra? Muestra: un puntaje, dada las variables.



Problema: tomar muestras (valores) de modelos
Solución: algoritmo MCMC (las probabilidades de transición y aceptar son el algoritmo)



Acá una animación de un algoritmo MCMC en acción.
Idea básica: se propone una muestra (markoviana) y algoritmo acepta o rechaza.



Fuente: https://blog.stata.com/2016/11/15/introduction-to-bayesian-statistics-part-2-mcmc-and-the-metropolis-hastings-algorithm/

Aclaraciones iniciales:

- Podemos samplear posteriors no estandarizados (sin el denominador de Bayes).
- Usamos logaritmos para evitar over y underflows (productos son sumas en logaritmos).
- Pensemos MCMC como una estrategia para visitar "terreno", proporcional a su importancia (siguiente diapositiva)

Metafora del político (Kruschke, 2014).

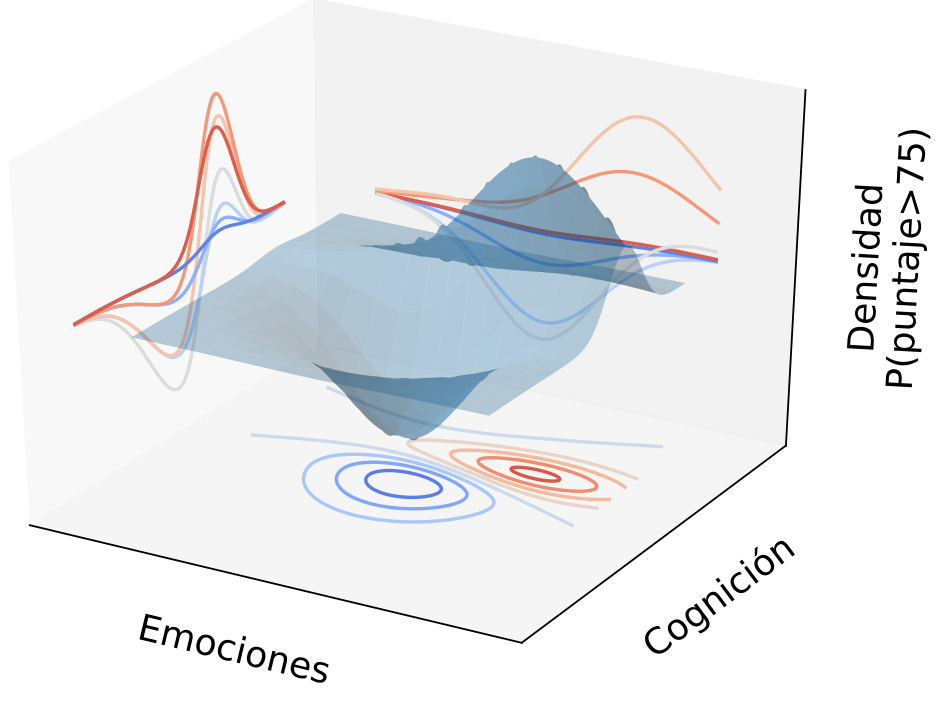
Problema: Dar un discurso en el Estadio Campin. La candidata quiere que el estadio se llene proporcional a la población de las localidades.

Estrategia:

1. Empezar a reclutar en alguna localidad aleatoria.
2. Para ir a otra, lanzar una moneda norte-sur u oriente-occidente. Otro lanzamiento para subdirección (e.g. norte)
3. Si la localidad propuesta tiene más población, ir con certeza. Si tiene menos, ir probabilísticamente. Por ejemplo, si tiene 50% de la población ir con 50% de probabilidad a la nueva localidad, de lo contrario seguir en la actual.
4. De forma general, $Prob_{moverse} = min(\frac{poblacion_{propuesta}}{poblacion_{actual}}, 1)$

Esta heurística es eficiente en el largo plazo: visitamos las localidades proporcional a su población.

Traducción metáfora:
Localidades -> Regiones de la función
Población localidad -> Nivel de la función en una región
Individuo confirmado -> Muestra



Hay muchas versiones de MCMC (Gibbs, Metropolis, Hamiltonian). A nivel general, muchas comparten la siguiente estructura:

1. Posición inicial (muestra actual)
2. Propuesta a donde moverse (muestra propuesta)
3. Aceptar/rechazar propuesta basado en que tanto respeta la data y priors.
4. Si se acepta, moverse a la propuesta. Si se rechaza, seguir en posición actual.
5. Repetir desde paso 1.
6. Repetir 1-5 por iteraciones o muestras requeridas.

PREAMBULO EJEMPLOS

Los lenguajes como Python, R, Julia ya tienen funciones que nos permiten tomar muestras.

Pienselas como ladrillos que nos van a permitir tomar muestras de posteriors complicados.

Actividad: busque en Internet cómo funcionan las funciones .pdf,.cdf,.rvs de scipy.stats:

- scipy.stats.norm.pdf; scipy.stats.norm.cdf; scipy.stats.norm.rvs
- scipy.stats.multivariate_normal.pdf; scipy.stats.multivariate_normal.cdf; scipy.stats.multivariate_normal.rvs
- scipy.stats.beta.pdf; scipy.stats.beta.cdf; scipy.stats.beta.rvs
- scipy.stats.dirichlet.pdf; scipy.stats.dirichlet.cdf; scipy.stats.dirichlet.rvs

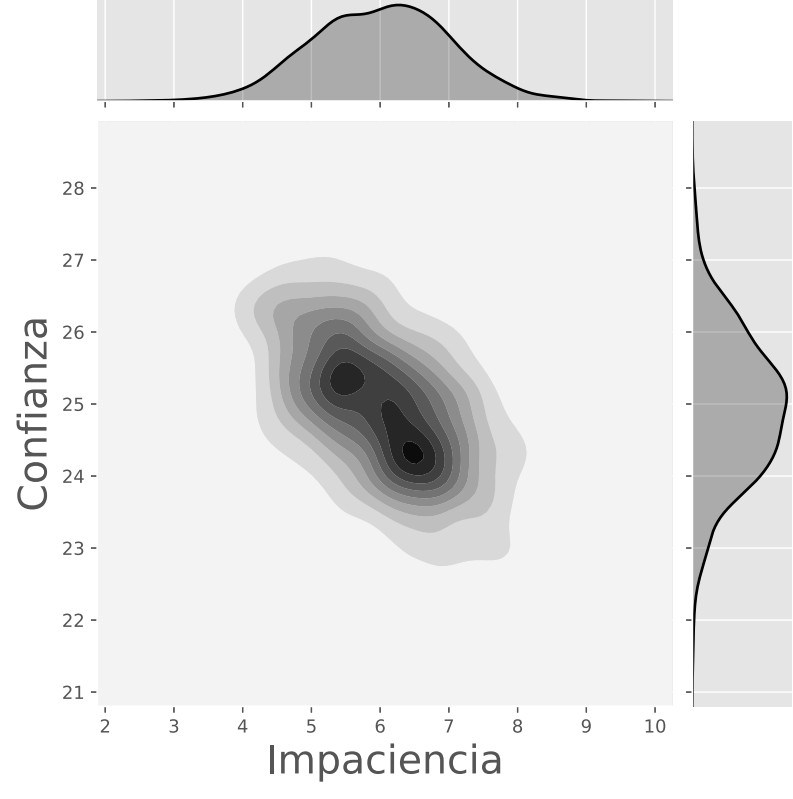
EJEMPLO 1 MCMC: GIBBS SAMPLER

Vamos a ver dos ejemplos: Gibbs y Metropolis.

En la práctica, usaremos rutinas MCMC en paquetes python (PyMC, Edward). Pero vale la pena ver estos dos ejemplos por:

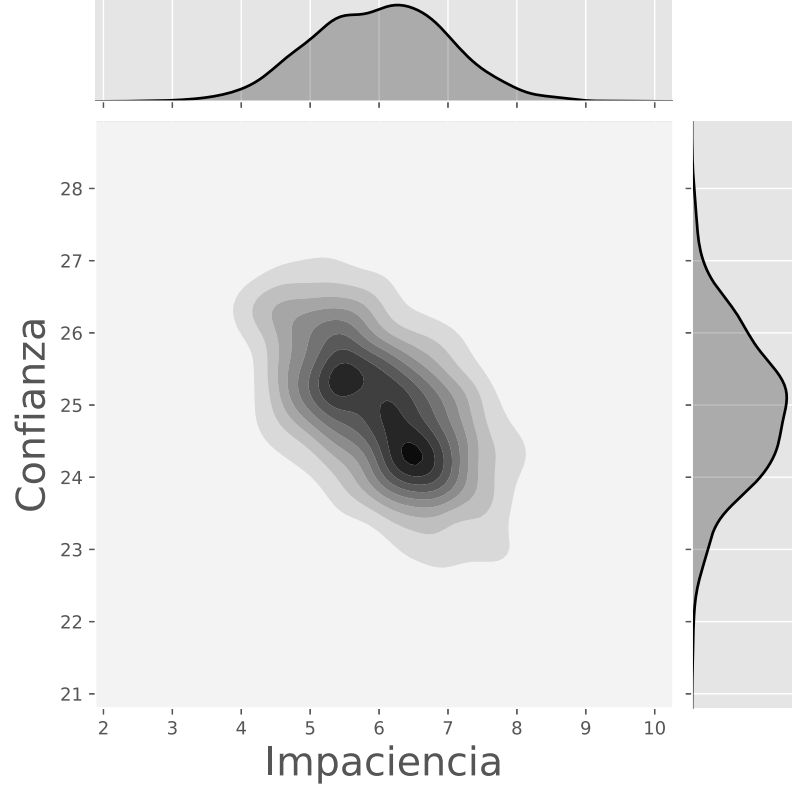
- Clásicos
- Generar intuición de métodos MCMC

La imagen muestra la distribución real de unas variables
El primer método MCMC a ver, para tomar muestras de espacios multidimensionales similares, es Gibbs.
Veamos el caso multivariado normal.



Enfoquemonos en dos dimensiones: p(var1, var1).

Pongamosle nombres a las variables: p(confianza, impaciencia). La imagen muestra que hay correlación negativa (e.g. Kidd, 2013).



La idea básica detrás de Gibbs:

- Suponga que conocemos pero es difícil tomar muestras de la distribución conjunta $p(confianza, impaciencia)$
- Suponga que es fácil tomarlas de las condicionadas: $p(confianza|impaciencia)$ y de $p(impaciencia|confianza)$. Por ejemplo, ambas son normales.
- El algoritmo hace lo siguiente:

    1. Pone valores iniciales a confianza y paciencia.
    2. Toma una muestra aleatoria de $p(confianza|impaciencia)$,
    3. Luego una de $p(impaciencia|confianza)$,
    4. Repite 2 y 3 hasta tener suficientes muestras.

Para Gibbs necesitamos:

- Conjunta:

$$p(conf, impa) \sim N\left(\begin{bmatrix} \mu_{conf} \\ \mu_{impa} \end{bmatrix} \begin{bmatrix} 1 & \rho_{conf,impa} \\ \rho_{conf,impa} & 1 \end{bmatrix}\right)$$

- Condicionales (no lo demostramos acá, pero esta es la formula de las condicionadas para una multivariada normal)

$$p(conf|impa) \sim N(\mu_{conf} + \rho_{conf,impa}(impa - \mu_{impa}), 1 - \rho^2_{conf,impa}))$$
$$p(impa|conf) \sim N(\mu_{impa} + \rho_{conf,impa}(conf - \mu_{conf}), 1 - \rho^2_{conf,impa}))$$

Ahora solo resta aplicar el algoritmo.

In [2]:

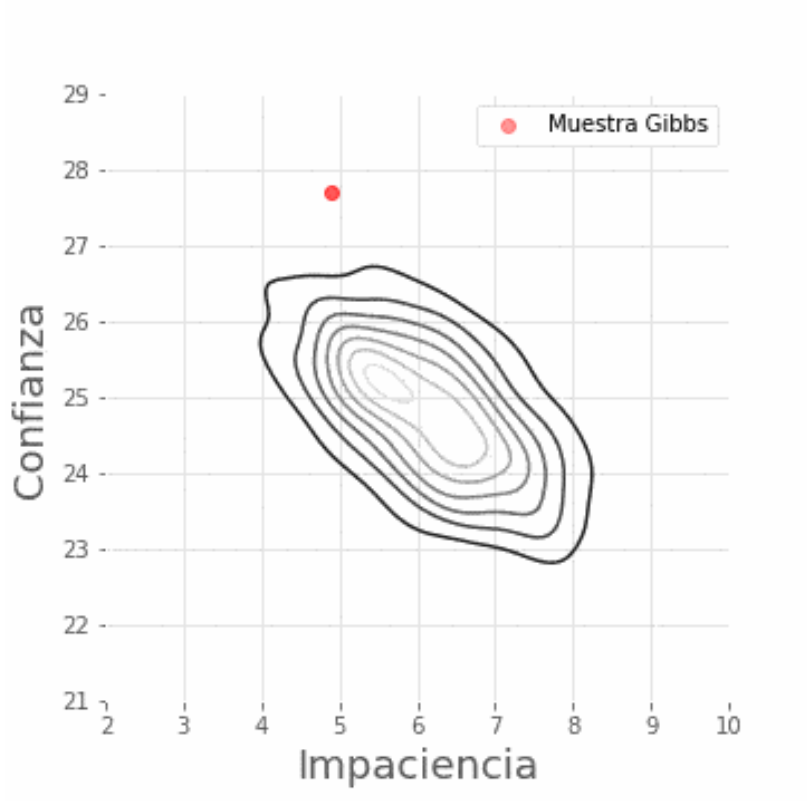
```
#Ejemplo Gibbs sampler para multivariada normal bidimensional
#Parametros joint distribution
mean_joint = [6, 25] #[impaciencia, confianza]
corr = -0.6
cov = [[1, corr], [corr, 1]]

#Paso 1: valores iniciales
conf_samp = st.uniform.rvs(20,30,1)
impa_samp = st.uniform.rvs(0,10,1)
niter = 200 #número de iteraciones
joint = pd.DataFrame({'impaciencia': np.repeat(float('nan'),niter),
                      'confianza': np.repeat(float('nan'),niter)})
for i in range(niter):
    #Paso 2: samplear un valor aleatorio de confianza
    conf_samp = st.norm.rvs(mean_joint[1]+corr*(impa_samp-mean_joint[0]),1-corr**2
,1)

    #Paso 3: samplear un valor aleatorio de impaciencia
    impa_samp = st.norm.rvs(mean_joint[0]+corr*(conf_samp-mean_joint[1]),1-corr**2
,1)

    joint.loc[i,'impaciencia'] = impa_samp
    joint.loc[i,'confianza'] = conf_samp
```

El algoritmo hace un buen recorrido de todo el espacio



En términos generales, con Gibbs:

- Se puede tomar muestras de cualquier distribución conjunta $p(\theta_1, \theta_2, \ldots \theta_n, data)$. No solo normal, lo anterior fue un ejemplo.
- Se necesita conocer las probabilidades condicionales de todas las variables, dado las demás.
- Es markoviano: el siguiente "paso" depende del estado actual (muestra actual).
- Ventaja: siempre se aceptan las propuestas. Desventaja: hay que saber los condicionales

EJEMPLO 2 MCMC: METROPOLIS & METROPOLIS-HASTINGS

Gibbs es un caso particular de Metropolis-Hastings. La diferencia general con Gibbs:

- Necesita una distribución conocida para proponer valores (muestras) que el algoritmo acepta o rechaza.
- En Gibbs esa distribución eran los condicionales.
- En Metropolis-Hastings pueden ser otras. De hecho, muchas versiones ni siquiera usan los condicionales.

Volvamos a nuestro problema favorito: Queremos saber la probabilidad de ocurrencia de un evento binario a partir de la data (e.g. ganar/perder, categoria 1/categoria 2, culpable / inocente, etc.).

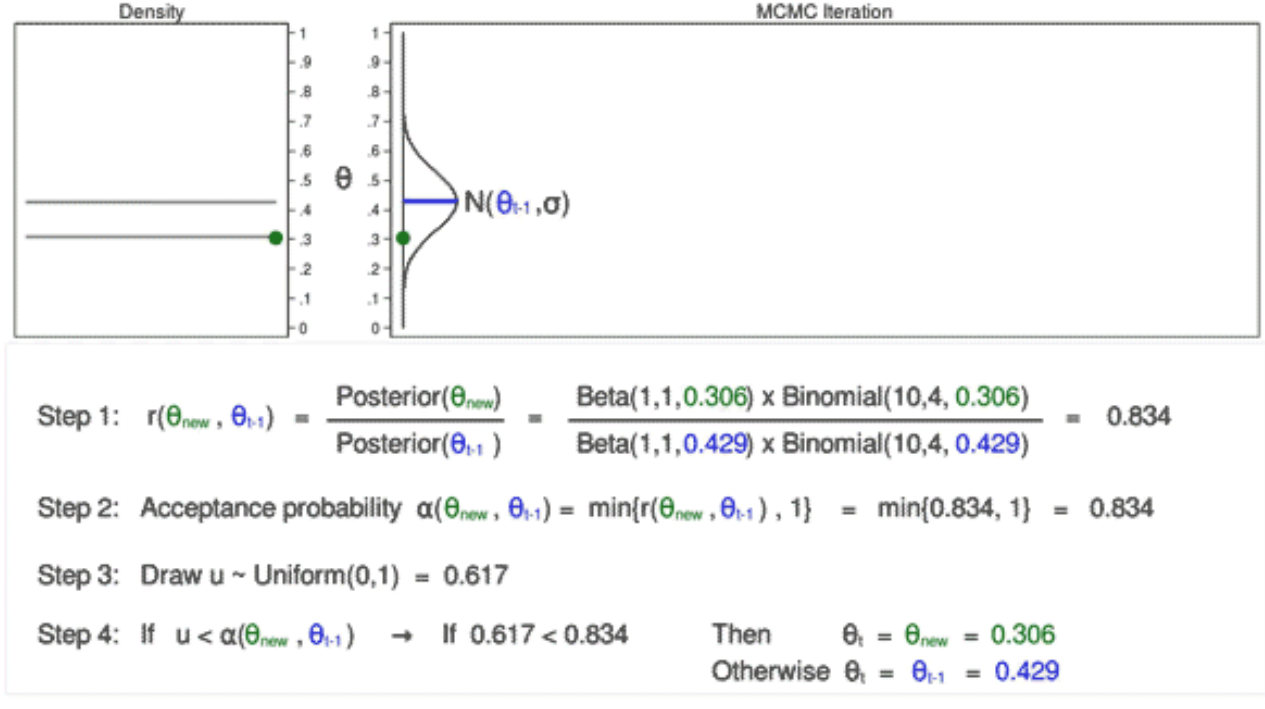
Solución: Bayes.

$$\begin{aligned} p(\theta|data) &\propto p(data|\theta)p(\theta) \\ &\propto \boldsymbol{B}inomial(n, z, \theta)beta(\alpha, \beta) \end{aligned}$$

Donde n es lanzamientos, z éxitos, θ probabilidad de éxito, α, y β parámetros de la prior

En una anterior imagen ya vimos el algoritmo Metropolis (el Metropolis-Hastings en siguientes diapositivas).

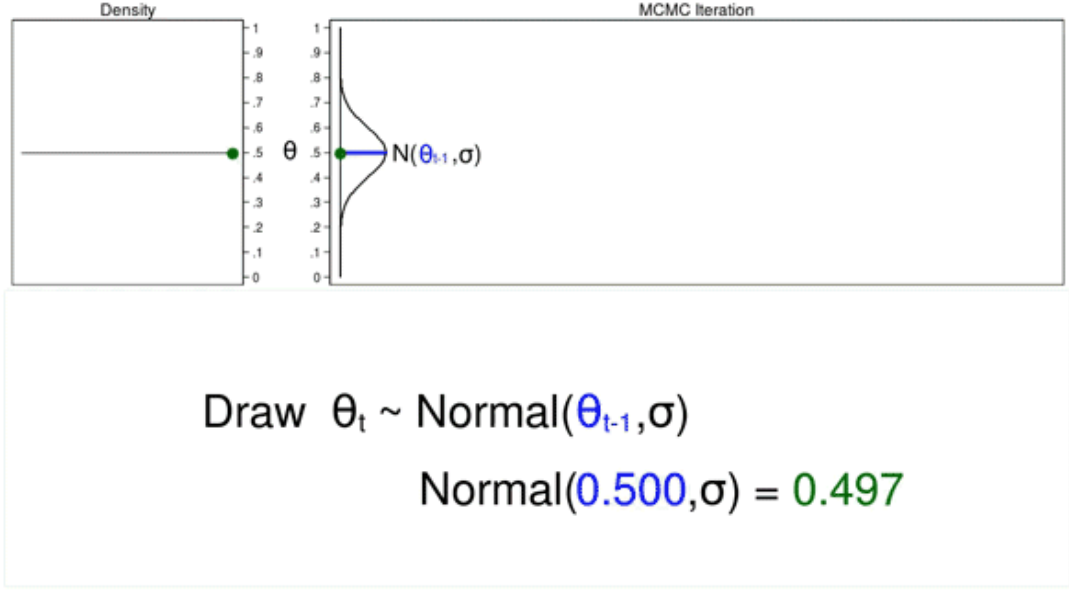
Se proponen valores de una distribución normal que "salta". Se calcula un ratio r de la posterior de la propuesta y la actual (sin estandarizar). Se acepta o rechaza con unos criterios.



Fuente: https://blog.stata.com/2016/11/15/introduction-to-bayesian-statistics-part-2-mcmc-and-the-metropolis-hastings-algorithm/

Sobre la distribución de propuesta q para Metropolis:

- Salta cuando se acepte un valor. En esta visualización, el centro de la normal se mueve a la propuesta θ^* (que fue aceptada).
- Debe ser simétrica: $q(\theta^*|\theta) = q(\theta|\theta^*)$. Evita sesgos (e.g. saltos mayores desde algunos puntos).



Fuente: https://blog.stata.com/2016/11/15/introduction-to-bayesian-statistics-part-2-mcmc-and-the-metropolis-hastings-algorithm/

Metropolis-Hastings (MH) es la version general. No requiere un distribución de propuesta q simétrica. Para compensar, el ratio r de posteriors p se cambia a un ratio de ratios.

Metropolis	**Metropolis-Hastings**
$\frac{p(\theta^* \mid y)}{p(\theta \mid y)}$	$\frac{p(\theta^* \mid y)/q(\theta^* \mid \theta)}{p(\theta \mid y)/q(\theta \mid \theta^*)}$

Para ejemplificar MH, estimemos el promedio y varianza de un proceso gaussiano (e.g. puntajes en una prueba).

El estimativo frecuentista es directo: el promedio y varianza de la data. Queremos el bayesiano: la creencia en forma de distribución de posibles parámetros dado los datos.

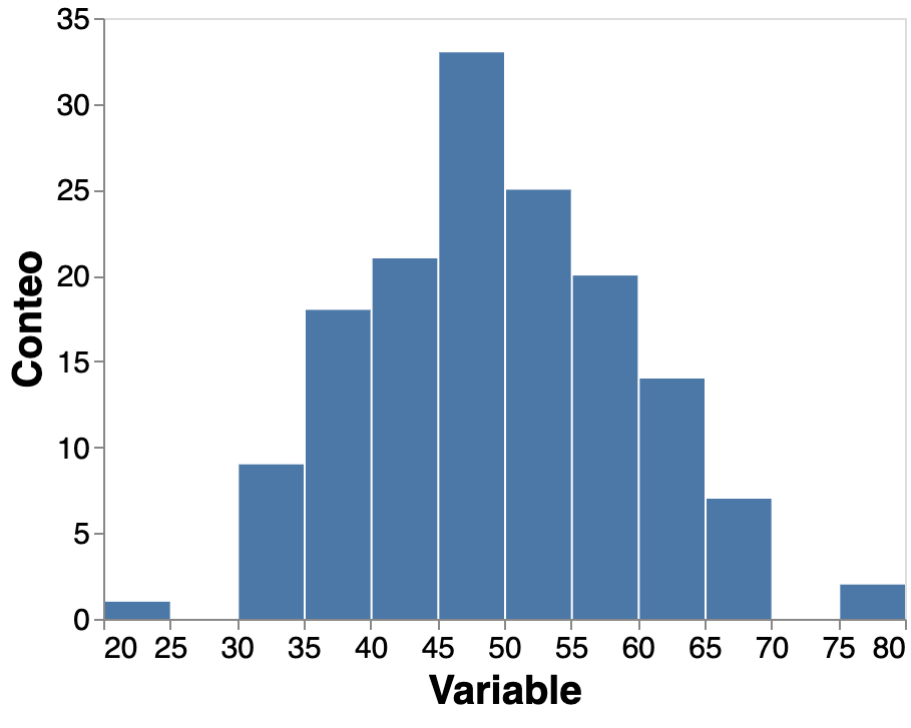
Podemos conseguir la formula dado que es gaussiano, pero tratemos con MH.

Primero generemos unos datos ...

In [3]:
```
# Metropolis-Hastings
# Datos
tamano = 50000
mu_real = 50 #parámetro a inferir, imagine que no los conocemos.
sigma_real = 10 #parámetro a inferir, imagine que no los conocemos.
poblacion = np.random.normal(mu_real,sigma_real,tamano)
n = 150 #solo observamos n datos de la población
data = pd.DataFrame({'muestra': poblacion[np.random.randint(0, tamano, n)]})
```

In [5]:
```
#Conteos y densidad de la data (kernel estimate)
both
```

Out[5]:



Ahora recordemos paso a paso el algoritmo:

1. Defina distribuciones likelihood y prior
2. Defina una distribución de propuesta $q(\theta)$
3. Tome una muestra de $q(\theta)$. Esa muestra es la propuesta θ^*
4. Calcule el ratio de ratios $\frac{p(\theta^*|y)/q(\theta^*|\theta)}{p(\theta|y)/q(\theta|\theta^*)}$
5. Acepte o rechaze θ^*. Si acepta, mueva $q(\theta)$. Si rechaza no se mueva.
6. Repita 1 a 4 por la iteraciones o muestras deseadas.

In [6]:

```
##### Paso 1
#Prior & likelihood
def log_prior(pars):
    #Priors
    #    mean: uniforme
    #    std. dev: uniform (positiva)
    #Supuesto: parámetros son independientes (multiplicar; en log sumar)
    #Input: pars[0] = mean, pars[1] = std. dev
    #Output: log. density
    log_prob_mu = st.uniform.logpdf(pars[0], -100, 200)
    log_prob_sd = st.uniform.logpdf(pars[1], 0, 100)
    return log_prob_sd + log_prob_mu
def log_lik_normal(pars,data):
    #Input: pars[0] = mean, pars[1] = std. dev, data = muestra
    #Output: log. density
    return np.sum(st.norm.logpdf(data, pars[0], pars[1]))
```

In [7]:

```
##### Paso 2 & 3
# Distribución de propuesta
class propuesta:
    #Con MH, no es necesario que sea simétrica.
    #Acá usamos diferentes std y densidades (truncada normal y normal)
    def __init__(self):
        self.scales = [7, 2] #desviaciones estandar
    def rvs(self, pars):
        left_lim = 0 #limite para truncar la normal
        right_lim = 50
        a = (left_lim - pars[1]) / self.scales[1]
        b = (right_lim - pars[1]) / self.scales[1]
        rand_var = [st.norm.rvs(pars[0], self.scales[0], size=1),
                    st.truncnorm.rvs(a=a, b=b, loc=pars[1],
                                     scale=self.scales[1], size=1)]
        return rand_var
    def log_pdf(self, x, pars):
        mu_logpdf = st.norm.logpdf(x[0], pars[0], self.scales[0])
        left_lim = 0 #limite para truncar la normal
        right_lim = 50
        a = (left_lim - pars[1]) / self.scales[1]
        b = (right_lim - pars[1]) / self.scales[1]
        sd_logpdf = st.truncnorm.logpdf(x[1], a=a, b=b, loc=pars[1],
                                        scale=self.scales[1])
        return sd_logpdf + mu_logpdf
```

In [8]:

```
##### Paso 4 & 5
# Ratio de ratios y aceptar
def aceptar(x, x_nuevo, data):
    # Posteriors (sin estandarizar)
    posterior_nuevo = log_prior(x_nuevo)+log_lik_normal(x_nuevo, data)
    posterior_actual = log_prior(x)+log_lik_normal(x, data)
    # Propuesta
    propuesta_nuevo = propuesta().log_pdf(x_nuevo, x)
    propuesta_actual = propuesta().log_pdf(x, x_nuevo)
    # Ratios
    ratio1 = posterior_nuevo - propuesta_nuevo #Resta por que estamos con logaritm
os
    ratio2 = posterior_actual - propuesta_actual
    ratio_de_ratios = ratio1-ratio2 #Resta por que estamos con logaritmos
    # Aceptar
    u = st.uniform.rvs(0, 1, size = 1)
    acceptance_prob = np.min([1, np.exp(ratio_de_ratios)]) #exp para transformar l
ogaritmos
    if u<acceptance_prob:
        return True
    else:
        return False
```

In [9]:

```
##### Paso 6
# Repita
def metropolis_hastings(par_inicial, iteraciones, data):
    x = par_inicial
    aceptado = []
    rechazado = []
    for i in range(iteraciones):
        x_nuevo =  propuesta().rvs(x)
        if aceptar(x, x_nuevo, data):
            x = x_nuevo
            aceptado.append(x_nuevo)
        else:
            rechazado.append(x_nuevo)
    return np.array(aceptado), np.array(rechazado)
```

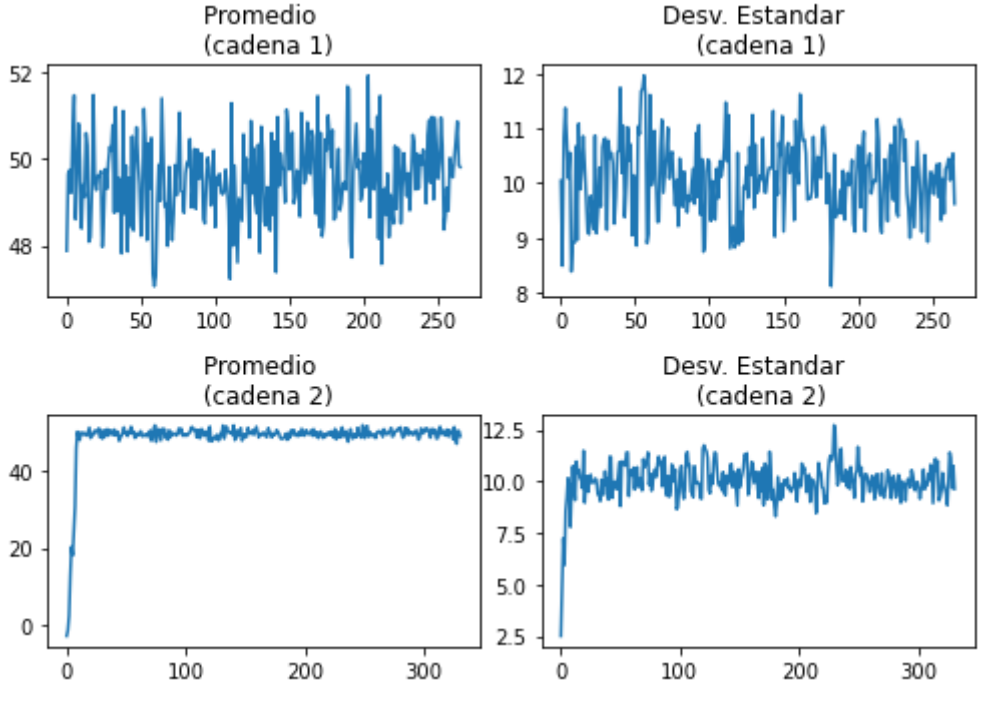
In [10]:
```
##### Resultados
#Cadena 1
par_inicial, iteraciones = [data.mean(), data.std()], 5000
A1, R1 = metropolis_hastings(par_inicial, iteraciones, data)
#Cadena 2
par_inicial, iteraciones = [4, 1], 5000
A2, R2 = metropolis_hastings(par_inicial, iteraciones, data)
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16: RuntimeWa
rning: overflow encountered in exp
  app.launch_new_instance()
```

In [12]:
```
# Convergen ambas cadenas. Hay tests pero se puede ver.
# Las trazas gravitan alredor de los mismos valores (pero baja aceptación)
my_MH_plot(A1, A2) #Note el "burn-in" al comienzo
```

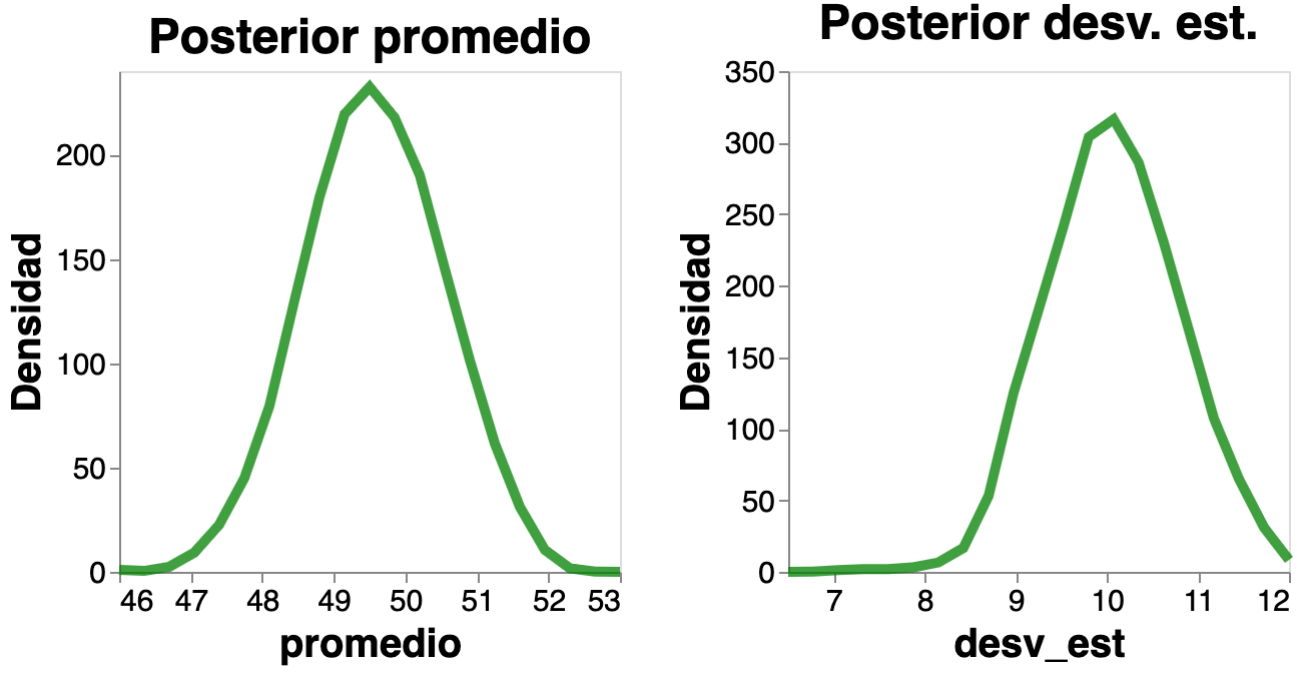
```
Info. cadena 1.
95% HDI promedio: [ 47.69 , 51.41 ]
95% HDI desv. est: [ 8.85 , 11.52 ]
accept: 0.0532

Info. cadena 2.
95% HDI promedio: [ 46.04 , 51.36 ]
95% HDI desv. est: [ 8.67 , 11.46 ]
accept: 0.0662
```



In [14]:
```
# Resultados (en forma de distribución, ambas cadenas)
plot
```

Out[14]:



"Todos los modelos están mal, algunos son útiles" (Box,1976)

¿Predice nuestro modelo la data?
Posterior predictive check (y^{rep}: valor simulado; y: data; θ: parámetros)

$$p(y^{rep}|y) = \int p(y^{rep}|\theta)p(\theta|y)d\theta$$

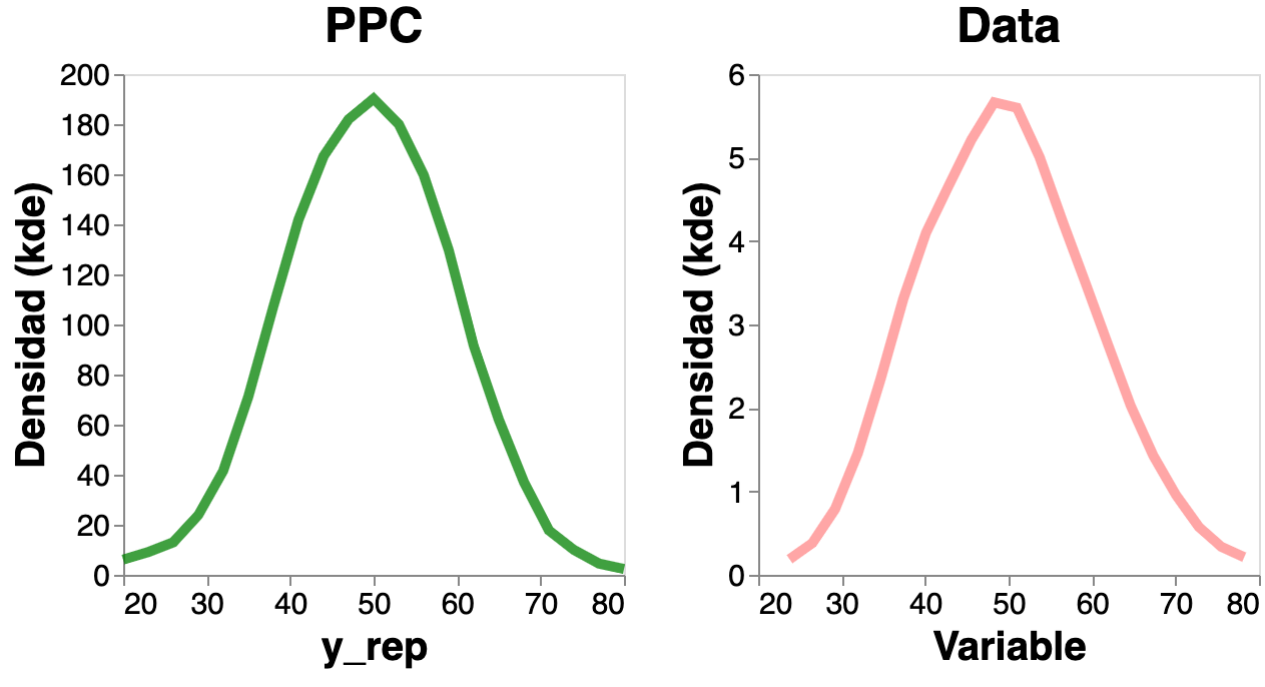
En lenguaje natural (casi):
En la integral, está el likelihood (1er termino) y posterior (2do). La implementamos con valores aleatorios de los parámetros θ.

In [15]:
```
# Posterior predictive check
n_rep = 5000
y_rep = []
for n in range(n_rep):
    #posterior (2do termino integral)
    idx = np.random.randint(data_mcmc.shape[0])
    prom_rep = data_mcmc.loc[idx,'promedio']
    idx = np.random.randint(data_mcmc.shape[0])
    desv_est_rep = data_mcmc.loc[idx,'desv_est']

    #likelihood (1er termino integral)
    y_rep.append(st.norm.rvs(prom_rep, desv_est_rep, size=1))
```

In [17]:
```
plot_ppc_data
```

Out[17]:

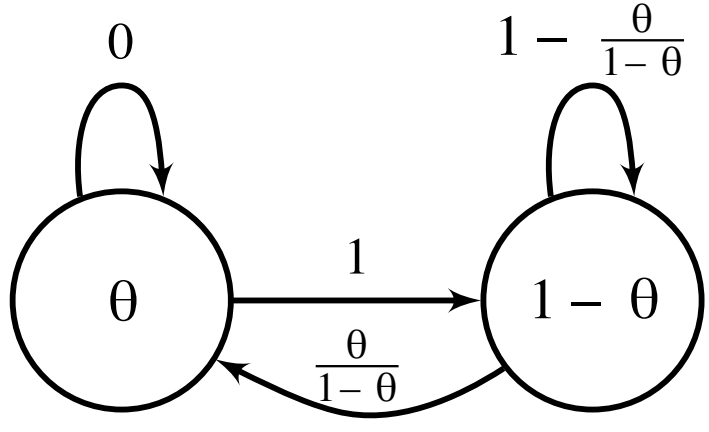


¿Metropolis-Hastings siempre converge a la posterior?

Respuesta corta: sí, el algoritmo converge cuando se cadena se hace bien (aunque puede demorarse). Respuesta larga, ver capitulo 7, Kruschke, 2da edición.

¿Metropolis-Hastings siempre converge a la posterior?

Intuición. Tenemos la posterior $\theta = 0.15$. Esta cadena converge a la posterior.



Fuente: https://people.duke.edu/~ccc14/sta-663/MCMC.html

Converge si la cadena es:

- Irreducible (se puede ir a cualquier estado)
- Aperiodica (no nos quedamos en un loop entre estados)
- Recurrente (el tiempo de volver a un estado dado es finito)

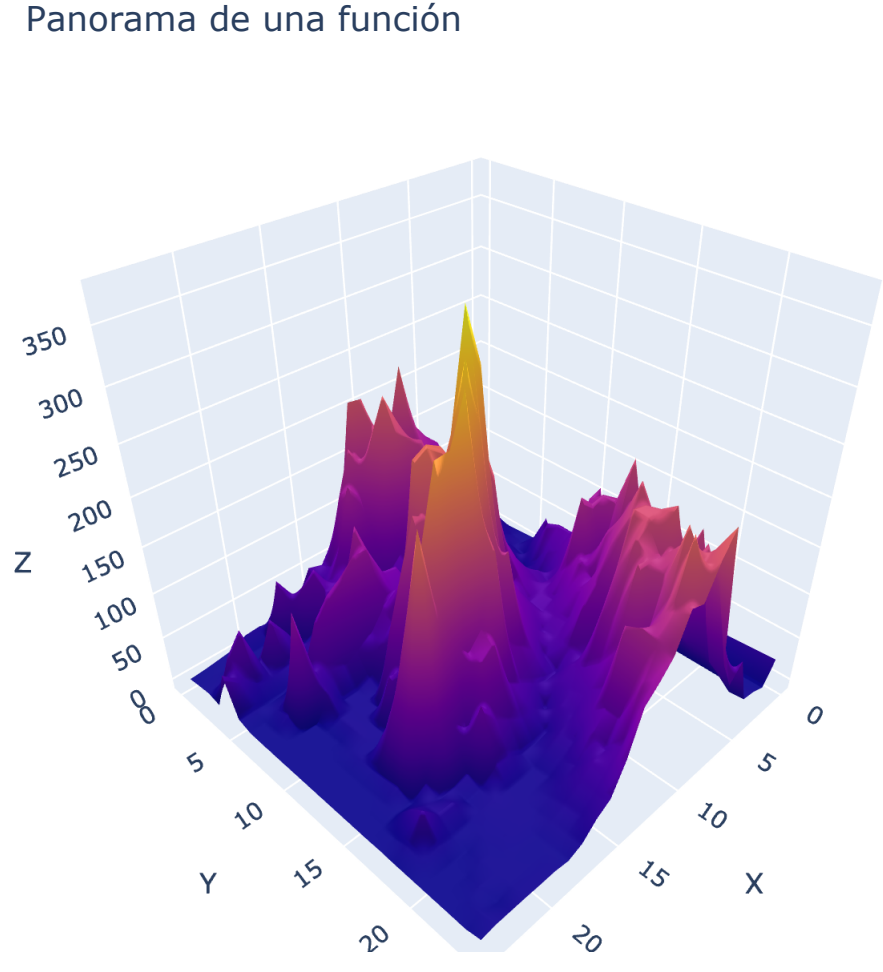
HERRAMIENTA COMPLEMENTARIA: GRADIENT DESCENT

Objetivo: obtener el posterior

Solución anterior: método estocástico (Markov + Montecarlo)

Solución complementaria: seguir un gradiente

¿Qué es gradient descent (ascent)?
Metáfora alpinista (punto más alto) o urbanista (punto más bajo)



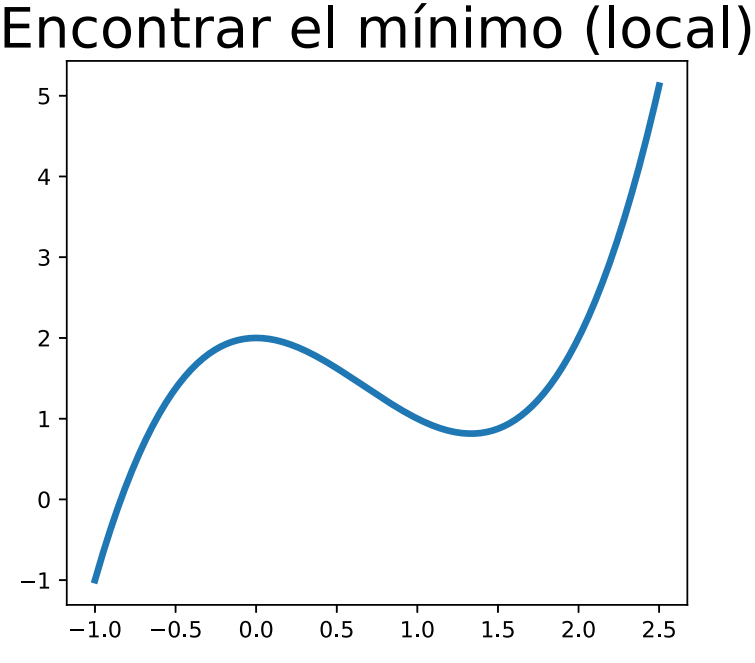
El algoritmo para descent (minimizar) se ve algo así:

- Calcular el gradiente (pendiente) negativo de la función en la posición actual: $\nabla f(x_k)$
- Moverse en dirección del gradiente negativo.

Veamos la implementación de github.com/dtnewman/ con:

$$f(x) = x^3 - 2x^2 + 2$$

In [20]:
```
fig, ax = plt.subplots(1,1, figsize=[5,4.5]);
f = lambda x: x**3-2*x**2+2
x = np.linspace(-1,2.5,1000)
ax.plot(x, f(x), lw = 3);
ax.set_title('Encontrar el mínimo (local)', fontsize=25);
#fig.savefig("img/3_CB/Gradient_Desc.svg"), plt.close();
plt.close()
```

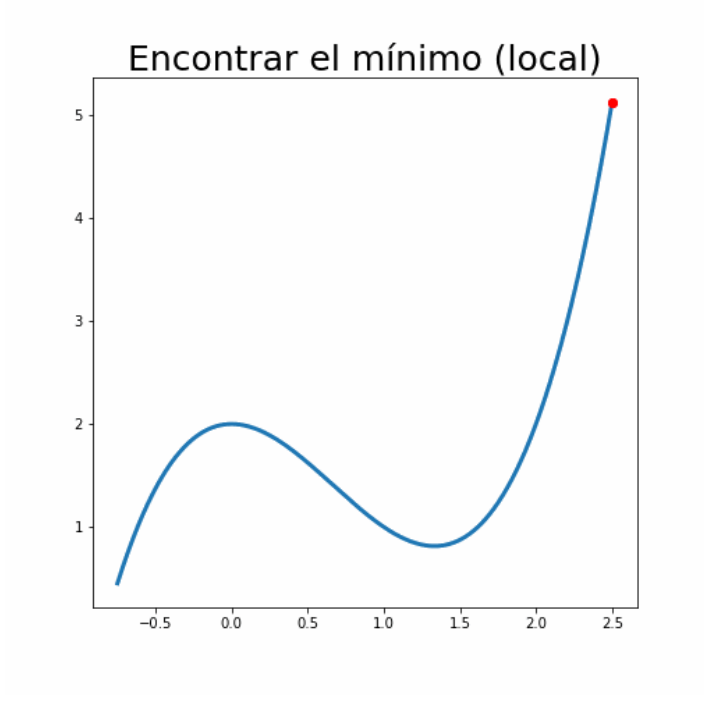


In [21]:

```
# Código 2D gradient descent github.com/dtnewman/stochastic_gradient_descent/

def f_prime(x): #derivada de nuestra función
    return 3*x**2-4*x

x_old,  x_new = 0, 2.5 #posición inicial
n_k = 0.01 #tamaño del paso
precision = 0.001
x_list, y_list = [x_new], [f(x_new)]
while abs(x_new - x_old) > precision:
    x_old = x_new
    s_k = -f_prime(x_old)
    x_new = x_old + n_k * s_k #La "fuerza" del movimiento depende del gradiente
    x_list.append(x_new), y_list.append(f(x_new))
```



¿Podemos acelerar el algoritmo?

Sí. Podemos hacer el paso adaptativo.

En el siguiente ejemplo hacemos "trampa" con una función minimizadora de Python. Lo importante es ver que se puede ser creativo en cómo explorar el espacio de la función.

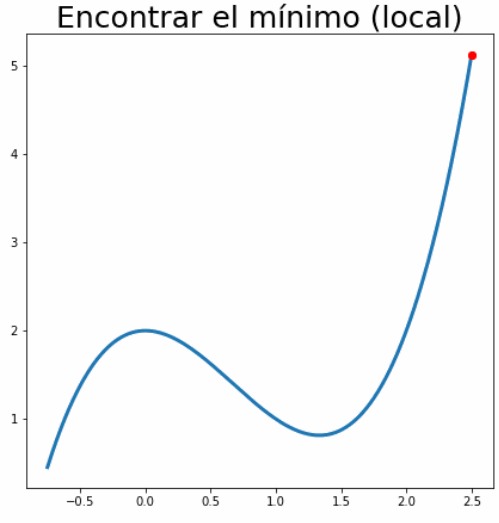
In [23]:

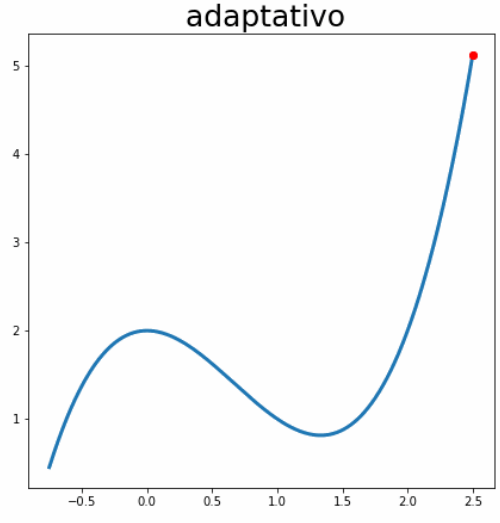
```
# Pasamos esta función a un minimizador (fmin)
def f2(n,x,s):
    #n: paso; fmin cambia este para minimizar y
    #s: gradiente; x: coordenada x
    x = x + n*s #desplazamiento en x
    y = f(x) #y (a minimizar)
    return y

x_old,  x_new = 0, 2.5 #posición inicial
n_k_init = 0.1 #tamaño del paso (valor inicial a fmin)
x_list, y_list = [x_new], [f(x_new)]
while abs(x_new - x_old) > precision:
    x_old = x_new
    s_k = -f_prime(x_old)

    # use scipy fmin function to find ideal step size.
    n_k = fmin(f2,n_k_init,(x_old,s_k), full_output = False, disp = False)

    x_new = x_old + n_k * s_k
    x_list.append(x_new)
    y_list.append(f(x_new))
```


Encontrar el mínimo (local)
5
4
3
2
1
−0.5
0.0
0.5
1.0
1.5
2.0
2.5


Encontrar el mínimo (local)
adaptativo
5
4
3
2
1
−0.5
0.0
0.5
1.0
1.5
2.0
2.5

Gradient descent no asegura encontrar mínimos (máximos) globales, solo locales.

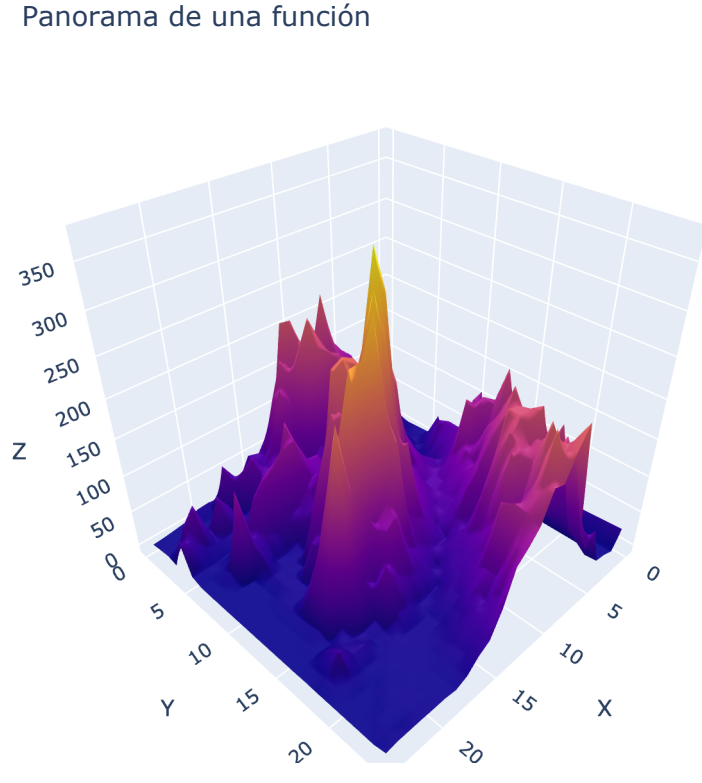
Podemos buscar óptimos (locales) de cualquier función. Por ejemplo, de una función de minimos cuadrados ordinarios (OLS, por sus siglas en inglés):

$$\frac{1}{m}\sum_{i=1}^{m}(h_\beta(x_i)-y_i)^2$$

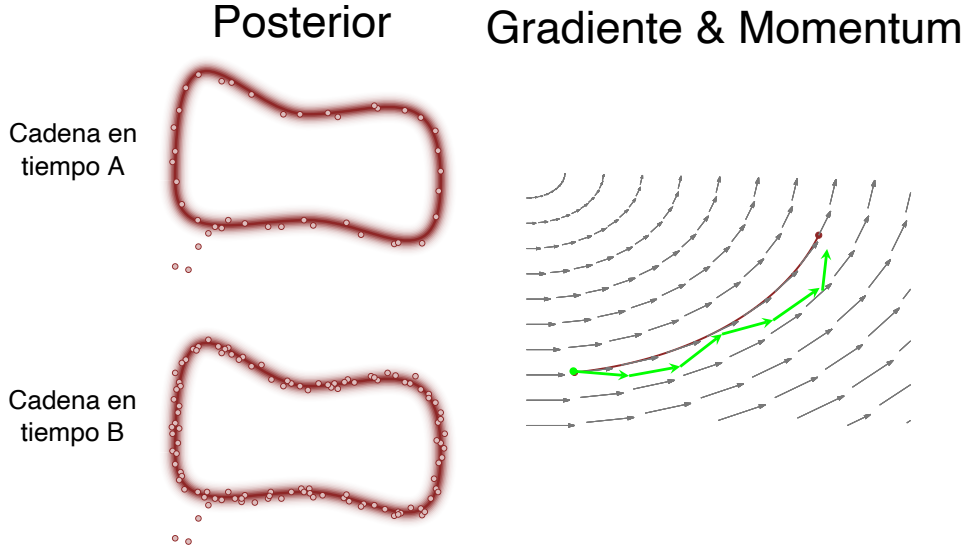
Puede haber varios β. Es el mismo algoritmo pero con derivadas parciales.

¿Qué tiene que ver con Bayes?

La posterior $p(\theta_1, \ldots \theta_n | data)$ es una función multidimensional. Hay algoritmos MCMC que explotan gradientes, por ejemplo Hamiltonian MCMC.



El MCMC hamiltoniano es popular. Esta es la intuición: recorre el posterior usando gradientes y momentum.



Fuente: Betancourt (2018)

VARIATIONAL INFERENCE

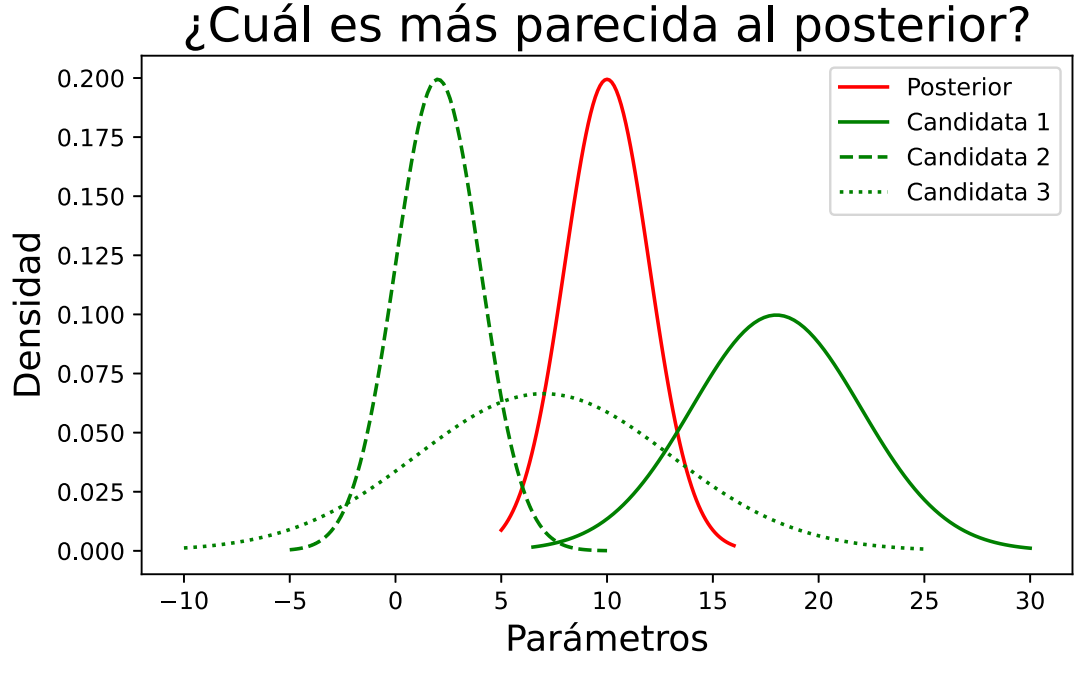
Metáfora pintor (minimizar distancia)

Fuente: https://displate.com/tonycenteno/displates

Problema: algunas posterior son difíciles de analizar matemáticamente o recorrer con MCMC.

Solución de variational inference: Escoger una distribución lo suficientemente parecida y que sea fácil de samplear. Esa distribución la llaman variational.

¿Qué significa que dos distribuciones sean parecidas?



Varios criterios. Usaremos divergencia Kullback–Leibler (KL)

$$D_{KL}(q(z)||p(z|x)) = \int_z q(z) log\left(\frac{q(z)}{p(z|x)}\right) dz$$

$q(z)$: Candidata (variational)
$p(z|x)$: Posterior

Positiva y asimétrica:

$$D_{KL}(q(z)||p(z|x)) \neq D_{KL}(p(z|x)||q(z))$$

Es el promedio de $log\left(\frac{q(z)}{p(z|x)}\right)$, ponderando por $q(z)$

$$D_{KL}(q(z)||p(z|x)) = \int_z q(z) log\left(\frac{q(z)}{p(z|x)}\right) dz$$

$log\left(\frac{q(z)}{p(z|x)}\right) = log(p(x)) - log(p(z|x))$. Es decir, es el promedio de la diferencia de (log) probabilidades de z.

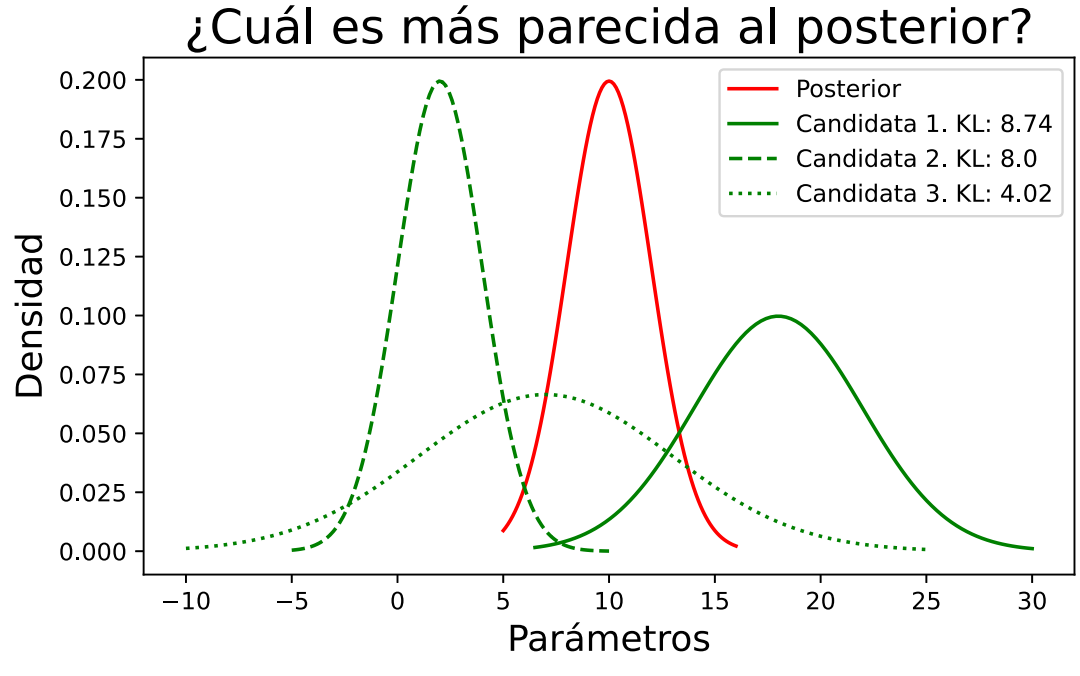
Mide cuanta información se pierde cuando usamos p para aproximar q.

Objetivo: minimizar esta cantidad (e.g. con gradient descent).

In [25]:

```
#Ejemplo simple
def KL(x, p, q):
    return p(x) * np.log( p(x) / q(x) )
p = st.norm(10,2).pdf
q = st.norm(18,4).pdf
kl_int, err = integrate.quad(KL,-30,30,args=(q,p))
print("La divergencia KL entre q y p es: ", round(kl_int,4))
```

```
La divergencia KL entre q y p es:  8.7396
```



Queremos minimizar la divergencia entre la posterior p y la candidata q, que es más fácil de samplear.

Pero primero, escribamos la divergencia KL de otra forma.

$$D_{\mathrm{KL}}\left(q(\boldsymbol{z})||p(\boldsymbol{z}|\boldsymbol{x})\right) = \int_{\boldsymbol{z}} q(\boldsymbol{z}) \log\left[\frac{q(\boldsymbol{z})}{p(\boldsymbol{z}|\boldsymbol{x})}\right] d\boldsymbol{z}$$

Apliquemos propiedad de logaritmos y dividamos la integral

$$= \int_{\boldsymbol{z}} \Big[q(\boldsymbol{z}) \log q(\boldsymbol{z})\Big] d\boldsymbol{z} - \int_{\boldsymbol{z}} \Big[q(\boldsymbol{z}) \log p(\boldsymbol{z}|\boldsymbol{x})\Big] d\boldsymbol{z}$$

Son promedios, ponderados por $q(z)$

$$= \mathbb{E}_q\Big[\log q(\boldsymbol{z})\Big] - \mathbb{E}_q\Big[\log p(\boldsymbol{z}|\boldsymbol{x})\Big]$$

Apliquemos la definición de probabilidad condicional

$$= \mathbb{E}_q\Big[\log q(\boldsymbol{z})\Big] - \mathbb{E}_q\Big[\log p(\boldsymbol{z}|\boldsymbol{x})\Big]$$
$$= \mathbb{E}_q\Big[\log q(\boldsymbol{z})\Big] - \mathbb{E}_q\left[\log\left[\frac{p(\boldsymbol{x}, \boldsymbol{z})}{p(\boldsymbol{x})}\right]\right]$$

Propiedad de logaritmos

$$= \mathbb{E}_q\Big[\log q(\boldsymbol{z})\Big] - \mathbb{E}_q\Big[\log p(\boldsymbol{x}, \boldsymbol{z})\Big] + \mathbb{E}_q\Big[\log p(\boldsymbol{x})\Big]$$

$\mathbb{E}_q\Big[\log p(\boldsymbol{x})\Big] = \log p(\boldsymbol{x})$ i.e. el prior de x (i.e. data) no depende de q(z) por definición

$$= \mathbb{E}_q\Big[\log q(\boldsymbol{z})\Big] - \mathbb{E}_q\Big[\log p(\boldsymbol{x}, \boldsymbol{z})\Big] + \log p(\boldsymbol{x})$$

Tenemos entonces que la divergencia KL es:

$$D_{\mathrm{KL}}\left(q(\boldsymbol{z})||p(\boldsymbol{z}|\boldsymbol{x})\right) = \mathbb{E}_q\left[\log q(\boldsymbol{z})\right] - \mathbb{E}_q\left[\log p(\boldsymbol{x}, \boldsymbol{z})\right] + \log p(\boldsymbol{x})$$

Ahora definamos evidence lower bound (ELBO) como una reorganización de los dos primeros términos:

$$\boldsymbol{ELBO} = \mathbb{E}_q\left[\log p(\boldsymbol{x}, \boldsymbol{z})\right] - \mathbb{E}_q\left[\log q(\boldsymbol{z})\right]$$

La divergencia KL queda así:

$$D_{\mathrm{KL}}\left(q(\boldsymbol{z})||p(\boldsymbol{z}|\boldsymbol{x})\right) = -\boldsymbol{ELBO} + log(p(x)$$

$\log p(\boldsymbol{x})$ es una constante, no afecta la minimización. Es decir, *maximizar ELBO* es lo mismo que minimizar la divergencia KL

$$D_{\mathrm{KL}}\left(q(\boldsymbol{z})||p(\boldsymbol{z}|\boldsymbol{x})\right) = -ELBO + log(p(x)$$

Se llama evidence lower bound por que es un limíte inferior del logaritmo de la evidencia (D_{KL} siempre es positivo)

$$\begin{aligned} log(p(x) &= ELBO + D_{\mathrm{KL}}\left(q(\boldsymbol{z})||p(\boldsymbol{z}|\boldsymbol{x})\right) \\ &\geq ELBO \end{aligned}$$

La idea es maximizar ELBO respecto a los parámetros z.

$$ELBO = \mathbb{E}_q\left[\log p(\boldsymbol{x}, \boldsymbol{z})\right] - \mathbb{E}_q\left[\log q(\boldsymbol{z})\right]$$

$p(x, z)$ la tenemos. ¿Cómo escogemos $q(z)$? Mitad ciencia mitad arte.

Fuente: https://displate.com/tonycenteno/displates

Una posibilidad es con mean-field variational family.

$$q(z_1, \ldots z_m) = \prod_{j=1}^{m} q(z_j)$$

Es decir, cada parámetro z_j tiene su propia distribución $q(z_j)$ (e.g. normal) y se asumen independientes.

Recordar que q es una aproximación a la posterior. En la verdadera posterior los parámetros sí se relacionan: la posterior es nuestro modelo generativo de los datos.

En suma variational inference es:

- Un método para encontrar parametros de una distribución fácil de samplear que se parece a la posterior
- Necesitamos la posterior p y la candidata (variational) q
- Podemos minimizar la divergencia con métodos computacionales (e.g. gradient descent, o expectation maximization; este último no lo vimos).

CONCLUSIÓN

- Obtener expresiones matemáticas para la posterior es difícil (incluso imposible)
- Usamos metodos computacionales. Vimos dos:

MCMC	Variational Inference
Insesgada	Sesgada
Lenta	Rápida