MÉTODOS COMPUTACIONALES PARA OBTENER LA POSTERIOR

Santiago Alonso-Díaz, PhD
Universidad Javeriana

En esta sesión veremos programación probabílistica.

Programación probabílistica se refiere a algoritmos que permiten hacer inferencia con distribuciones (no son programas inciertos).

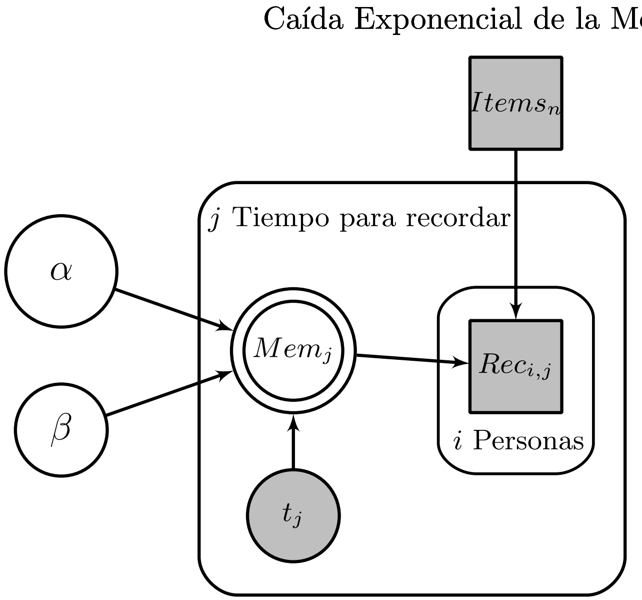
Con probabilistic programming se puede samplear posteriors elaboradas

Figuras:

- Cuadrado -> discreto
- Circulo -> continuo
- Frontera doble -> determinista
- Platos/grupos -> repeticiones

Colores:

- Blanco -> latente
- Gris -> observables



$\alpha \sim Beta(1,1)$

$\beta \sim Beta(1,1)$

$Mem_j \sim min\left(1, e^{-\alpha t_j} + \beta\right)$

$Rec_{i,j} \sim Binomial(Mem_j, Items_n)$

Stan	PyMC	Edward

Nota: PyMC3 ... PyMC4 está en pre-release.

Basado en: Davidson-Pilon, C. (2015). Bayesian methods for hackers: probabilistic programming and Bayesian inference. https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers [(https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers)](https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers)

PERO PRIMERO UN CORTO TUTORIAL DE DOT

Se pueden hacer modelos gráficos con Python usando la sintaxis de dot

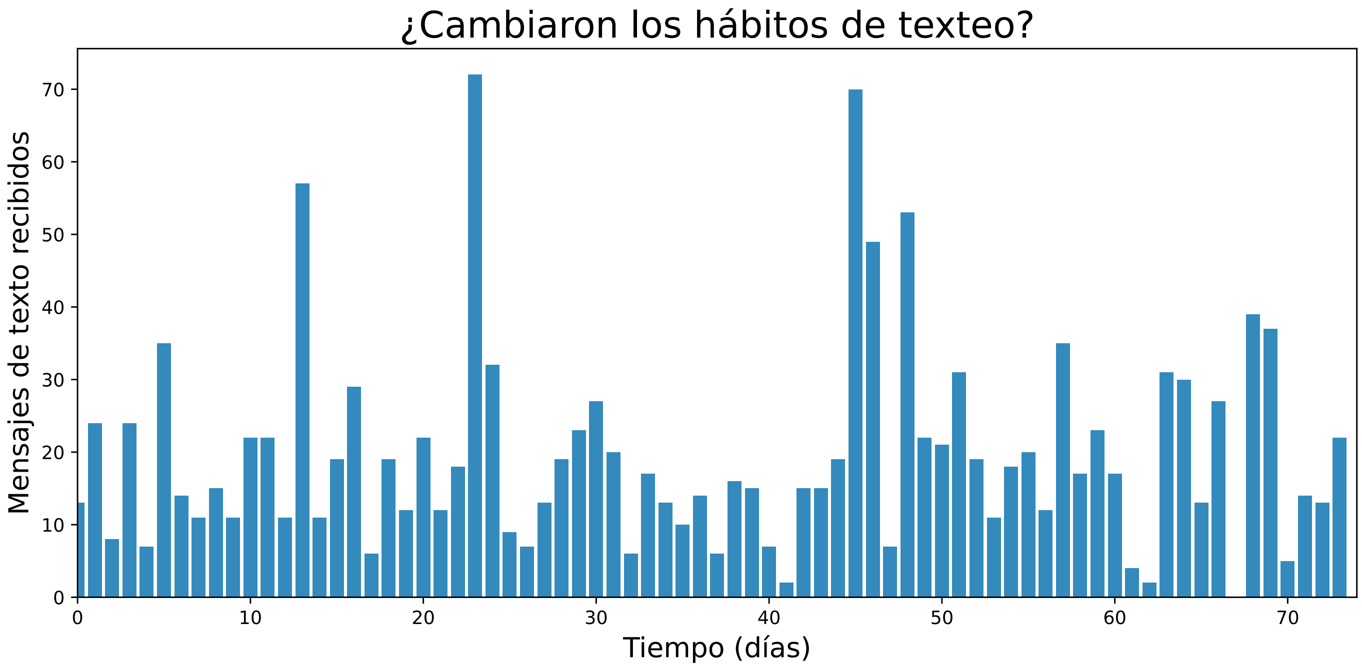
```
In [2]: #Primero definamos la gráfica
dot_text = 'digraph G {\
                    node[margin=0.2, shape=rectangle,\
                         style = rounded,\
                         width=0.8, height = .7];\
                    compound=true;\
                    newrank=true;\
                    d -> e [label=" mi texto "];\
                    subgraph cluster0{\
                       label = " ";\
                       texlbl = "$\\theta_i$";\
                       labeljust = "l";  color = "red";\
                       f;\
                       g;\
                    };\
                    e -> f [lhead = cluster0, label="  "];\
                    f -> g;\
                    g -> h [ltail = cluster0];\
                    { rank=same; f; e};\
                    d [label = "Nodo", fixedsize=true, width=0.5,\
                                shape = circle];\
                    e [texlbl = "$\\mu$", shape = square,\
                                 width = 0.6 ];\
                    f [label = "VI", shape = plaintext];\
                    g [texlbl = "$\\sum_i x_i$"];\
                    h [texlbl = "$\\int \\beta_i di$",\
                                 shape = circle, peripheries = 4];
\
                    }'
#Tip: use single quote at start and end; double quotes for labels
```

In [3]:

```
#Ahora salvarla
#Para que se vea bien en tex
tex = d2t.dot2tex(dot_text, format='tikz', preproc = True)
#crop: tamaño de página igual al modelo
tex = d2t.dot2tex(dot_text, texmode = 'verbatim', crop=True)
diagram_tex = open('img/4_CB/tutorial_dot.tex', 'w')
diagram_tex.write(tex)
diagram_tex.close()

# this builds a pdf-file inside a directory
pdf = build_pdf(tex)
#convertir a svg y pulir/editar posiciones en inkscape
pdf.save_to('img/4_CB/tutorial_dot.pdf')
```

Analicemos mensajes de texto de una persona (Fuente: Davidson-Pilon, 2015)

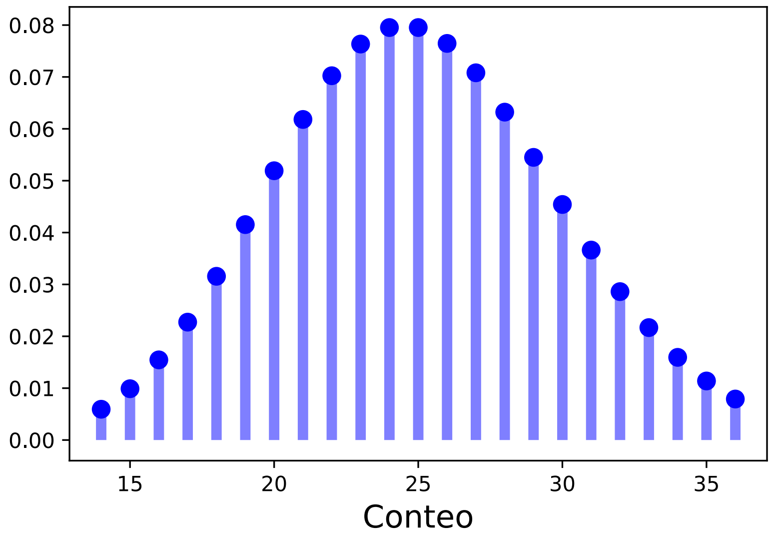
In [4]:
```
count_data = np.loadtxt("data/4_CB/txtdata.csv")
n_count_data = len(count_data)
dias = np.linspace(1, n_count_data, num = n_count_data, dtype = int)
```



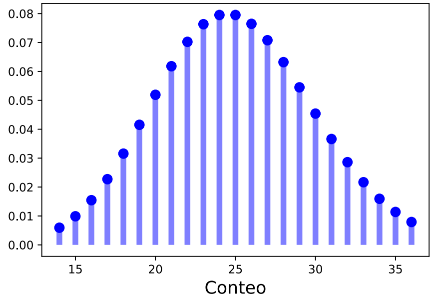
¿Cómo modelamos el conteo de mensajes por día?

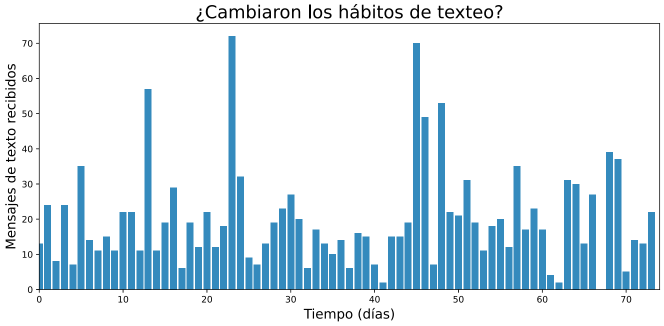
$$Conteo_{\text{día}} \sim Poisson(\lambda)$$

Nota: El parámetro λ es el promedio y varianza al mismo tiempo.



Pero ... se ven diferentes





Un histograma es una guía visual (depende de los bins)

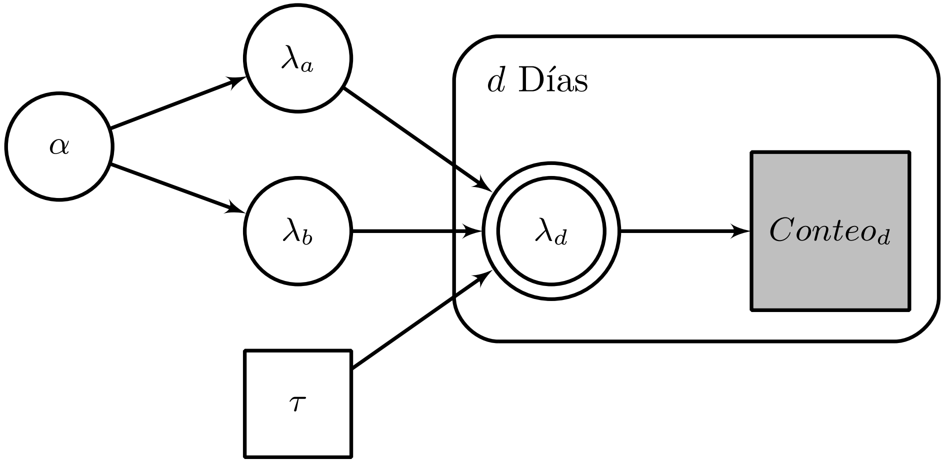
Pueden ser dos Poisson diferentes, después de un día τ

Esta es la propuesta:

$$\lambda = \begin{cases} \lambda_a & \text{día} < \tau \\ \lambda_b & \text{día} \geq \tau \end{cases}$$

Esta es la propuesta bayesiana completa:

Mensajes de Texto por Día



$$\alpha = \frac{1}{E[Conteo_d]}$$

$$\lambda_b \sim Exponential(\alpha)$$

$$\lambda_a \sim Exponential(\alpha)$$

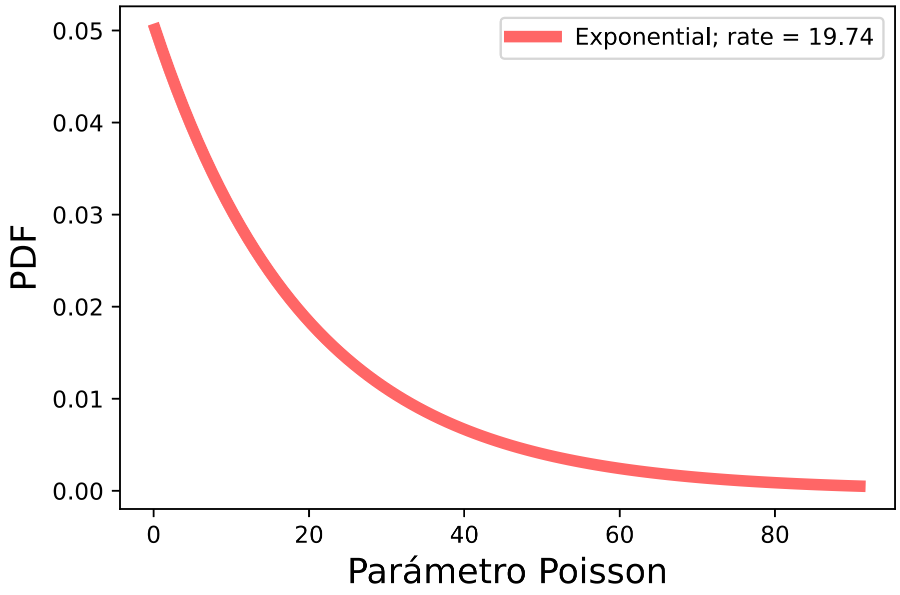
$$\tau \sim Uniform(0, N_{\text{días}})$$

$$\lambda = \begin{cases} \lambda_a & \text{día} < \tau \\ \lambda_b & \text{día} \geq \tau \end{cases}$$

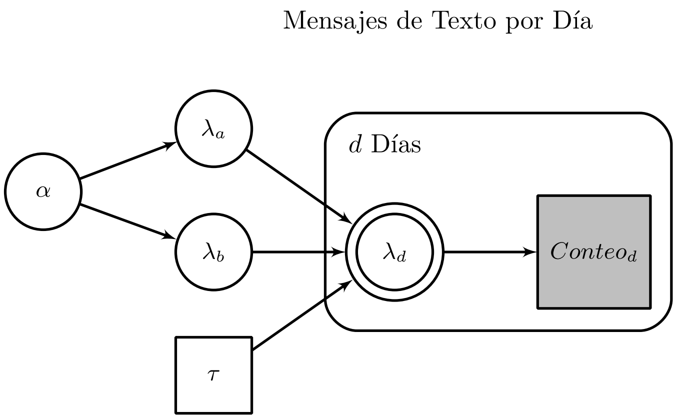
$$Conteo_d \sim Poisson(\lambda)$$

¿Por qué esas formas y distribuciones para los parámetros?

Parte ciencia (e.g. conjugate prior) y arte (conocimiento del área). Por ejemplo, para los λ_a y λ_b escogimos una exponencial con rate $\alpha = \frac{1}{\mathbb{E}[Conteo_d]}$. No es conjugate, pero hace algo deseable: castiga conteos altos.



Ahora tomemos muestra del modelo con PyMC



$$\alpha = \frac{1}{E[Conteo_d]}$$

$$\lambda_b \sim Exponential(\alpha)$$

$$\lambda_a \sim Exponential(\alpha)$$

$$\tau \sim Uniform(0, N_{\text{días}})$$

$$\lambda = \begin{cases} \lambda_a & \text{día} < \tau \\ \lambda_b & \text{día} \geq \tau \end{cases}$$

$$Conteo_d \sim Poisson(\lambda)$$

PyMC guarda el modelo en la clase `Model`. Veamos el código.

In [5]:
```python
# Primero definamos constantes y variables deterministicas
alpha = 1.0/count_data.mean()  # count_data holds our text counts.

# PyMC utiliza la clase `Model` para guardar el modelo
with pm.Model() as modelo_mensajes:

    ### Latentes
    lambda_antes = pm.Exponential("lambda_antes", lam = alpha)
    lambda_despues = pm.Exponential("lambda_despues", lam = alpha)
    tau = pm.DiscreteUniform("tau", lower=0, upper=n_count_data)
    #Siguiente línea:
    #if primer argumento true, then tercero, else segundo
    lambda_usado = pm.math.switch(tau < dias,
                                  lambda_despues, lambda_antes)

    ### Likelihood
    conteos = pm.Poisson("conteos", lambda_usado,
                         observed=count_data)
```

In [6]:

```
#La variable model tiene toda la información
#RV: random variables
print(modelo_mensajes.basic_RVs) #Todas
print(modelo_mensajes.free_RVs) #Latentes / no observables
print(modelo_mensajes.observed_RVs) #Observables

#En prob. program. las variables son aleatorias (ejemplo)
rvs = modelo_mensajes.lambda_antes.random
print(np.round(rvs(),2), np.round(rvs(),2), np.round(rvs(),2))
modelo_mensajes.lambda_antes
```

```
[lambda_antes_log__, lambda_despues_log__, tau, conteos]
[lambda_antes_log__, lambda_despues_log__, tau]
[conteos]
11.43 7.44 25.75
```

Out[6]: $\text{lambda\_antes} \sim \text{Exponential}(lam = 0.05065023956194388)$

¡Ahora sí a tomar muestras de nuestro modelo!

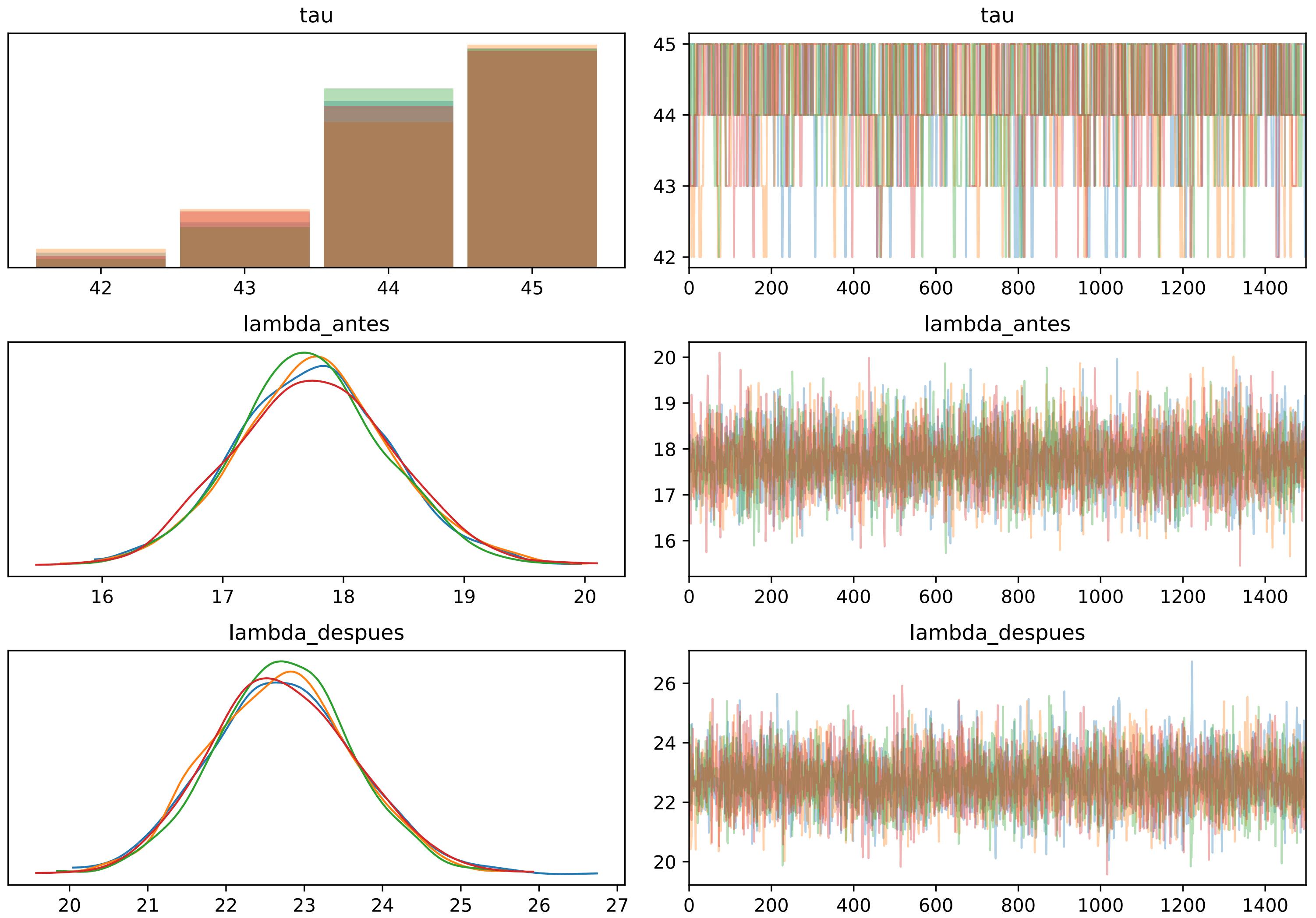
In [7]:
```python
with modelo_mensajes:
    trace = pm.sample(10000, chains = 4, cores = 2)
    #posterior_predictive = pm.sample_posterior_predictive(trace)
```

```
Multiprocess sampling (4 chains in 2 jobs)
CompoundStep
>NUTS: [lambda_despues, lambda_antes]
>Metropolis: [tau]
```

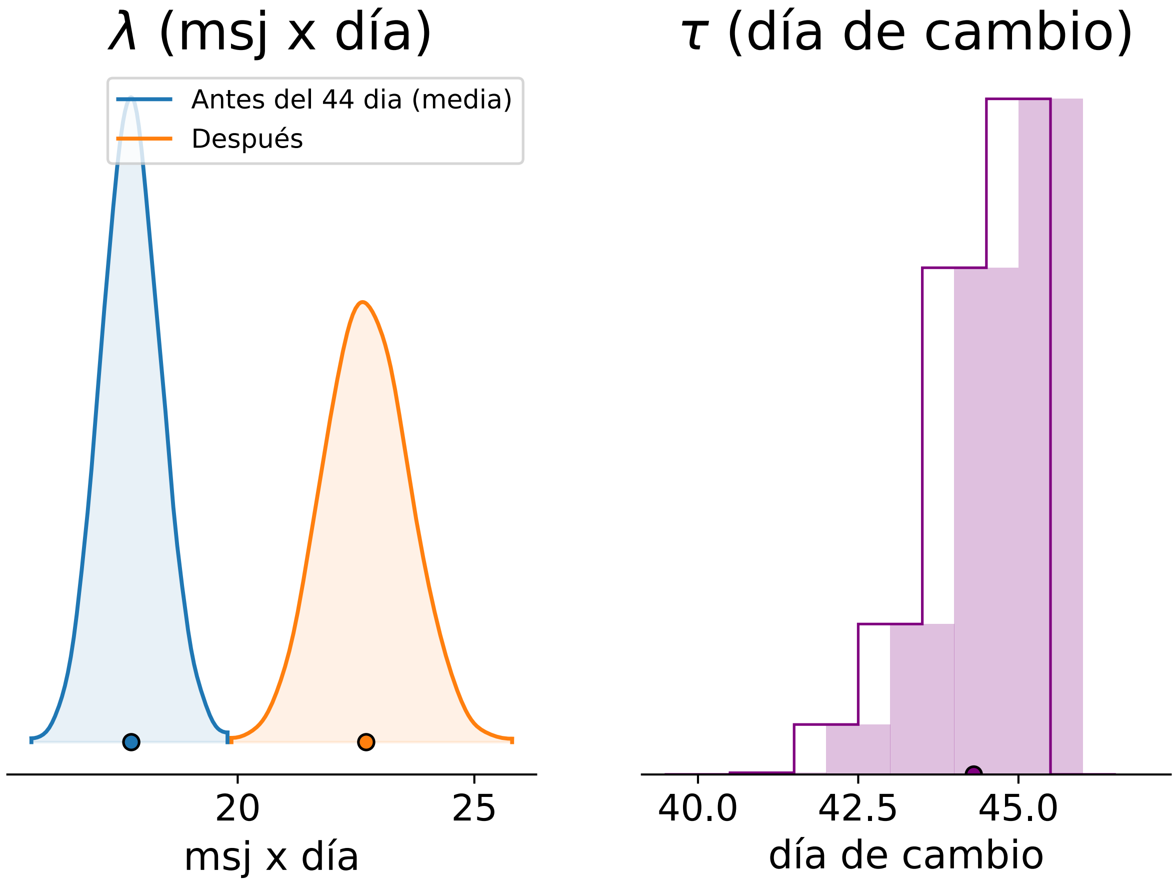
100.00% [44000/44000 00:30<00:00 Sampling 4 chains, 0 divergences]

```
Sampling 4 chains for 1_000 tune and 10_000 draw iterations (4_000 + 40_000 dr
aws total) took 31 seconds.
The number of effective samples is smaller than 25% for some parameters.
```

El modelo convergio. Hay tests, pero visualmente combina bien (plots derechos)



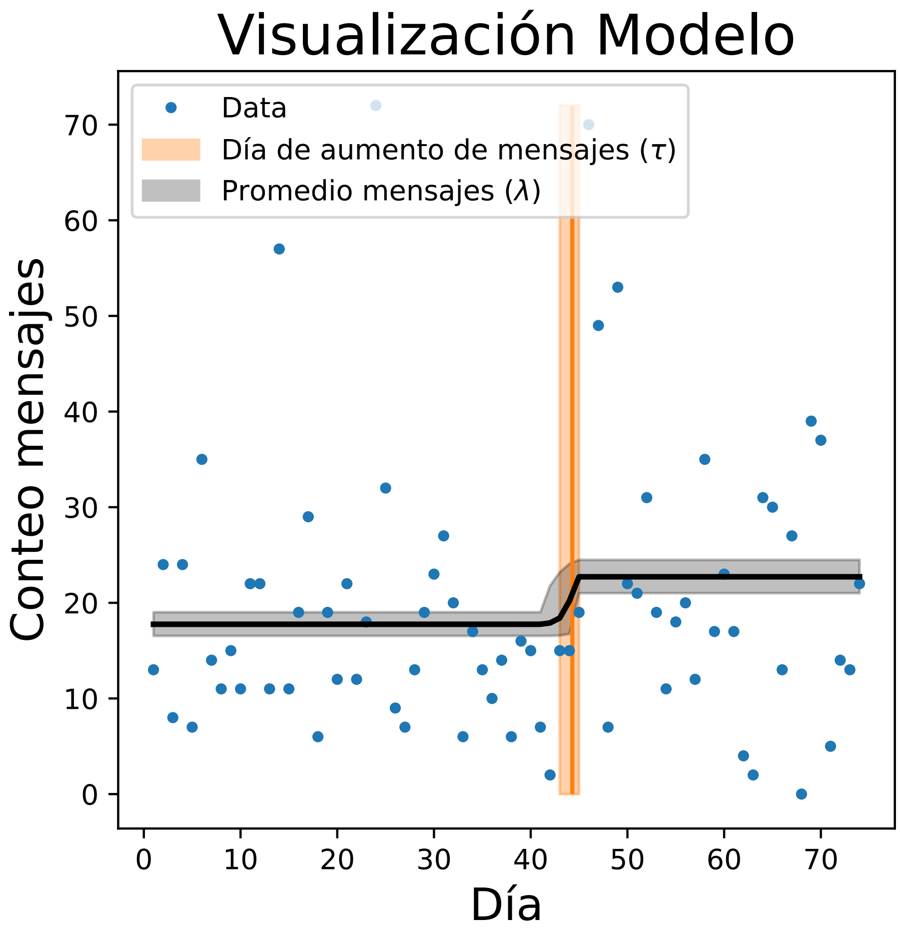
Aún cuando escogimos prior exponenciales (lambda) y uniformes (tau), las posterior se ven diferentes. La data modificó nuestra creencia.



Podemos sacar promedios y calcular intervalos de densidad con las muestras producidas por el algoritmo. El objeto que llamamos trace tiene la info por parámetro.

In [10]:
```
promedio_conteo = np.zeros_like(count_data, dtype='float')
#High density interval
HDI95_conteo = np.zeros_like(count_data, dtype='float')
HDI5_conteo = np.zeros_like(count_data, dtype='float')
for i, dia in enumerate(dias):
    #las muestras están ordenadas como el algoritmo las tomó
    idx = dia < trace['tau']
    antes = trace['lambda_antes'][idx]
    despues = trace['lambda_despues'][~idx]
    HDI95_conteo[i] = np.percentile(
        np.concatenate((antes,despues)),97.5
    )
    HDI5_conteo[i] = np.percentile(
        np.concatenate((antes,despues)),2.5
    )
    promedio_conteo[i] = np.concatenate((antes,despues)).mean()
```

Hay un salto en msjs x día. ¿Qué pudo pasarle a esta persona en el salto? Especular; o si hay conocimiento del área podríamos lanzar una hipótesis.

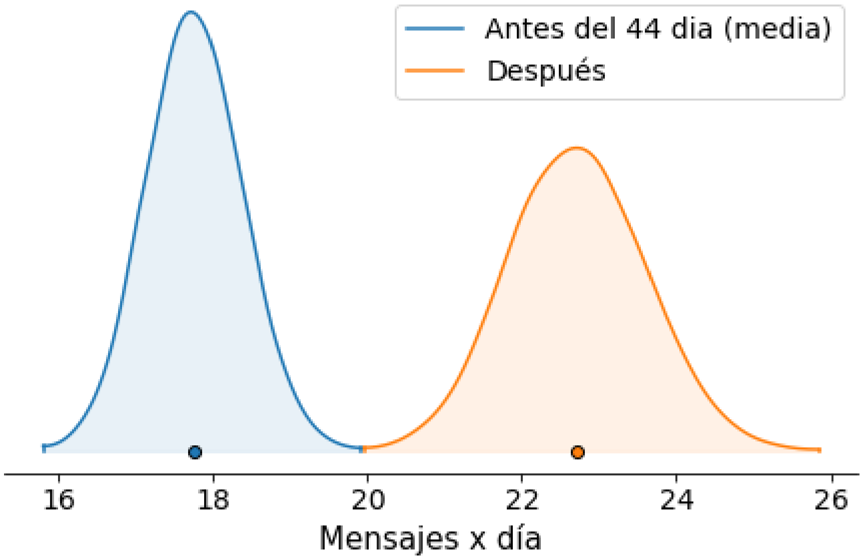


Podemos testear directamente hipótesis. Por ejemplo, ¿qué tan seguros estamos que la tasa de mensajes antes y después del día de cambio es diferente?

In [15]:

```
test = trace['lambda_antes']<trace['lambda_despues']
tA = str(int(test.mean()*100))
tB = "% seguros que msj x día (lambdas) difieren en el día del cambi
o"
print(tA + tB)
dia = int(trace['tau'].mean())
az.plot_density(
    [trace['lambda_antes'], trace['lambda_despues']],
    data_labels=["Antes del " + str(dia) + " dia (media)",
                 "Después"],
    shade=.1, hdi_prob = .999,
);
plt.xlabel('Mensajes x día', fontsize = 15)
plt.title("");
```

```
100% seguros que msj x día (lambdas) difieren en el día del cambio
```



EJERCICIOS (MENSAJES DE TEXTO POISSON)

1. Calcule el promedio de `lambda_antes` y `lambda_despues`.

In [ ]:
```
#Escriba su código aqui
```

2. Calcule el incremento porcentual promedio de mensajes por día. `pista:` Consiga la distribución de la variable *incremento porcentual*. No haga `lambda_antes.mean()/lambda_despues.mean()`. Es parecido, pero use todas las posteriores. Queremos el promedio de la distribución de *incremento porcentual*.

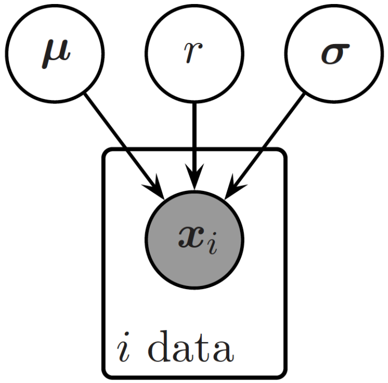
In [ ]:
```
#Escriba su código aqui
```

3. Calcule el promedio de λ_{antes} *dado* que $\tau < 45$. Recuerde: las muestras del sampleador están coordinadas. Es decir, busque los indices de τ menores a 45 y use ese indice para seleccionar muestras de λ_{antes}.

In [ ]:

```
#Escriba su código aqui
```

4. Haga el siguiente modelo gráfico con la sintaxis `dot`



In [ ]:

```
#Escriba su código aqui
```

PYMC: DETALLES

En PyMC el modelo "vive" en el objeto pm.Models

In [16]:
```
with pm.Model() as model:
    parameter = pm.Exponential("poisson_param", 1.0,
                               testval=0.5) #testval valores iniciales
    data_generator = pm.Poisson("data_generator", parameter)
```

Python trata de emular lenguaje natural. El bloque `with` trabaja con el objeto pm.Model, con el alias model, y ejecuta lo que va dentro del bloque.

Podemos trabajar con el alias

In [17]:
```
with model:
    data_plus_one = data_generator + 1
print(data_plus_one.tag.test_value) #test values: valores iniciales
print(model.basic_RVs)
rvs = model.poisson_param.random
print(np.round(rvs(),2), np.round(rvs(),2), np.round(rvs(),2))
```

```
1
[poisson_param_log__, data_generator]
4.69 1.08 1.35
```

PyMC3 tiene dos tipos de variables: estocásticas y determinísticas.

- *Estocásticas*: Variable aleatoria. Ejemplos en esta clase: `Poisson`, `DiscreteUniform`, and `Normal`.
- *Determinísticas*: El nombre lo dice todo. Ejemplos en esta clase: formula cuadratica. Una vez conozco el input x, con seguridad sé el valor.

ESTOCÁSTICAS

Ya hemos usado variables estocásticas. Por ejemplo,

```
mi_variable =
pm.DiscreteUniform("discrete_uni_var", 0, 4)
```

El nombre para Python es *mi_variable*. El nombre para el sampleador es *discrete_uni_var*. Pueden ser diferentes o iguales. Recomendación: iguales.

El 0 (límite bajo) y el 4 (alto) son los parámetros libres de la uniforme. Diferentes distribuciones y parámetros en (http://pymc-devs.github.io/pymc3/api.html (http://pymc-devs.github.io/pymc3/api.html))

Si hay N variables con distribuciones similares, se puede usar el parámetro `shape`

```
betas_regresion = pm.Uniform("betas", 0, 1,
shape = N)
```

DETERMINÍSTICAS

Hay dos formas de introducir variables determinísticas en nuestro modelo. Ya vimos una: con funciones del paquete PyMC.

In [18]:
```
n_data_points = 5
dias_o_cualquier_referencia = np.arange(n_data_points)
with pm.Model() as model:
    #Estocásticas
    lambda_1 = pm.Exponential("lambda_1", 1.0)
    lambda_2 = pm.Exponential("lambda_2", 1.0)
    tau = pm.DiscreteUniform("tau", lower=0, upper=10)

    #Determinísticas
    lambda_ = pm.math.switch(tau >= dias_o_cualquier_referencia,
                             lambda_1, lambda_2)
```

`lambda_` es determinística. Usamos la función .switch con dos de las variables estocásticas (VE). En cada iteración, VE se "materializa" y .switch se aplica

Otra forma es con `pm.Deterministic`.

In [19]:
```
def restar(x, y):
    return x - y

with pm.Model() as model_det:
    stochastic_1 = pm.Uniform("U_1", 0, 1)
    stochastic_2 = pm.Uniform("U_2", 0, 1)

    det_1 = pm.Deterministic("Delta", restar(stochastic_1, stochastic
_2))
```

La función restar debe poder trabajar con escalares o vectores.

Para una variable determinística, es recomendado primero tratar de usar funciones de PyMC o el backend (PyMC3, Theano; PyMC4, TensorFlow).

Veamos detalles del backend.

Pueden pensar en el backend como el caballo de trabajo. Hace los calculos y manipulaciones de la información.

Se basa en objetos matemáticos llamados tensores (generalización de vectores y matrices)

Acá usamos la operación `stack`. Agrupa `p1` y `p2` en una nueva variable `p`.

Se puede pensar como una lista con dos elementos (o una matriz con dos columnas). Los elementos NO son escalares.

La nueva variable `p` la podemos pasar a una variable estocástica como probabilidad de dos categorias.

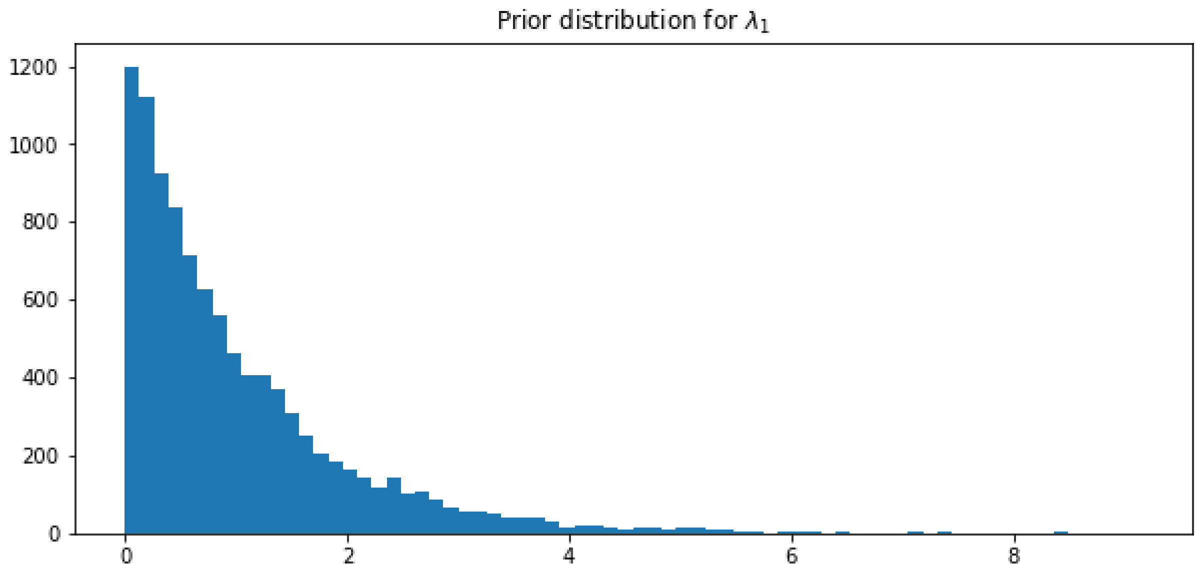
In [20]:
```
import theano.tensor as tt

with pm.Model() as theano_test:
    p1 = pm.Uniform("p", 0, 1)
    p2 = 1 - p1
    p = tt.stack([p1, p2])

    assignment = pm.Categorical("assignment", p)
```

Vamos a usar muchas operaciones del backend en el curso.

DATA/OBSERVACIONES

Hemos definido priors ...

In [21]:
```
fig = plt.figure(figsize = (10, 4.5))
samples = lambda_1.random(size=10000)
plt.hist(samples, bins=70);
plt.title("Prior distribution for $\lambda_1$");
```



... ahora veamos likelihoods y data

Es sencillo. Todas las variables estocásticas tienen un parámetro `observed`. A la que le asignemos la data es el likelihood.

In [22]:

```
# Generemos una data
data = np.array([10, 25, 15, 20, 35])
with model:
    obs = pm.Poisson("obs", lambda_, observed=data)
print(obs.tag.test_value)
print(model.free_RVs) #Latentes / no observables
print(model.observed_RVs) #Observables
```

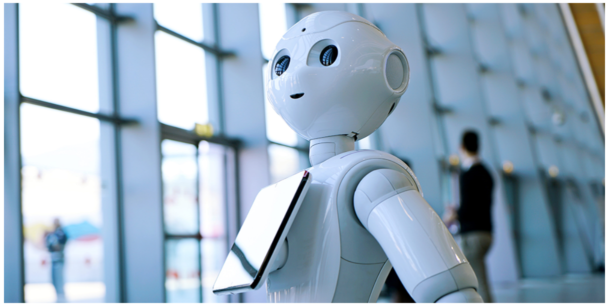
```
[10 25 15 20 35]
[lambda_1_log__, lambda_2_log__, tau]
[obs]
```

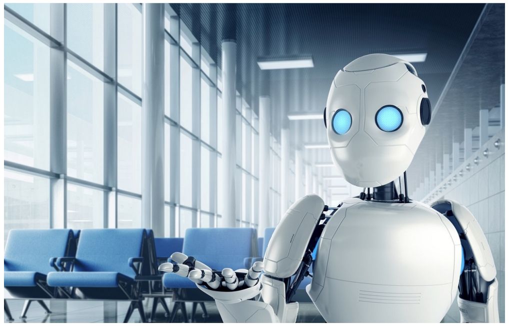
MODELOS BAYESIANOS DE ANÁLISIS DE DATOS

A/B TESTING

Tal vez el diseño experimental más simple y popular. Se busca cuál es mejor opción, A o B.

Por ejemplo, ¿qué robot debería atender en un aeropuerto?

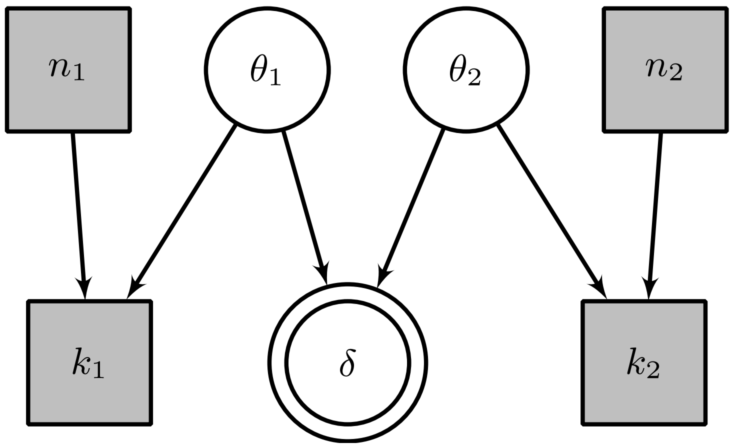




Diseño experimental: dos grupos en distintos aeropuertos. El tamaño de cada grupo es n_1 y n_2 con k_1 y k_2 conteos prefiriendo robot 1. Cada aeropuerto tiene una proporción latente θ_1 y θ_2.

Formulación bayesiana:

A/B testing



$$k_1 \sim Binomial(\theta_1, n_1)$$

$$k_2 \sim Binomial(\theta_2, n_2)$$

$$\theta_1 \sim Beta(1, 1)$$

$$\theta_2 \sim Beta(1, 1)$$

$$\delta = \theta_1 - \theta_2$$

In [23]:

```python
#No conocemos estas cantidades. Son los theta del modelo gráfico.
true_p_A = 0.05
true_p_B = 0.04

#Tenemos más datos en un sitio (A). No es problema
N_A = 1500
N_B = 750

#Generemos datos. Son los k del modelo gráfico.
datos_A = st.bernoulli.rvs(true_p_A, size=N_A)
datos_B = st.bernoulli.rvs(true_p_B, size=N_B)
print("Obs del sitio A: ", datos_A[:30], "...")
print("Obs del sitio B: ", datos_B[:30], "...")
print("Promedio A: ", np.mean(datos_A))
print("Promedio B: ", np.mean(datos_B))
```

```
Obs del sitio A:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
 0] ...
Obs del sitio B:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
 0] ...
Promedio A:  0.052
Promedio B:  0.037333333333333336
```

In [24]:

```python
# Formulemos el modelo pymc3 con priors uniformes para p_A & p_B
with pm.Model() as model:

    #Priors
    p_A = pm.Beta("p_A", 1, 1) #Beta(1,1) es uniforme en el rango 0,1
    p_B = pm.Beta("p_B", 1, 1)

    #Variable de interés: la diferencia de proporciones.
    delta = pm.Deterministic("delta", p_A - p_B)

    # Likelihood. Asumimos independencia de los dos sitios.
    obs_A = pm.Bernoulli("obs_A", p_A, observed=datos_A)
    obs_B = pm.Bernoulli("obs_B", p_B, observed=datos_B)

    # Llamemos el algoritmo para samplear.
    step = pm.Metropolis() #Tipo de algoritmo.
    nsamples = 20000
    trace = pm.sample(nsamples, step=step)
    burned_trace=trace[1000:] #burn-in; importante para Metropolis
```

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>Metropolis: [p_B]
>Metropolis: [p_A]
```

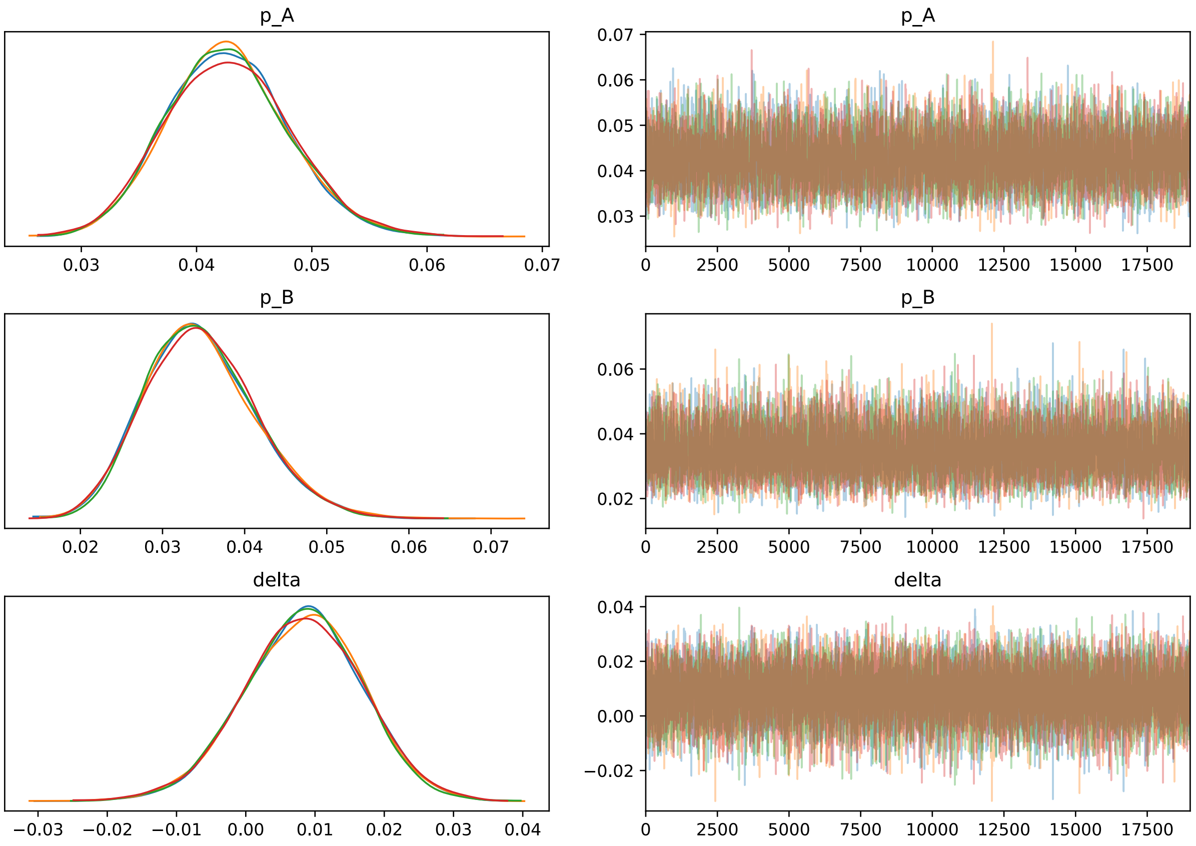
100.00% [84000/84000 00:33<00:00 Sampling 4 chains, 0 divergences]

```
Sampling 4 chains for 1_000 tune and 20_000 draw iterations (4_000 + 80_000 dr
aws total) took 34 seconds.
The number of effective samples is smaller than 25% for some parameters.
```

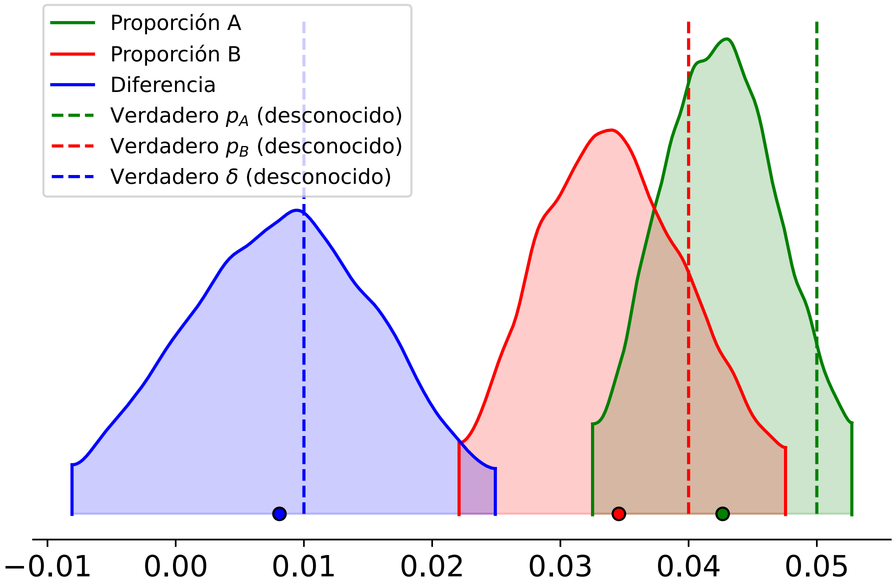
In [25]:

```
#Ahora grafiquemos
p_A_samples = burned_trace["p_A"]
p_B_samples = burned_trace["p_B"]
delta_samples = burned_trace["delta"]
data = az.from_pymc3(trace=burned_trace, model=model)

#Convergencia
az.plot_trace(data, figsize = [10,7]);
plt.savefig("img/4_CB/trace_proporcion.svg"); plt.close()

#Densidades
az.plot_density(
    [p_A_samples,p_B_samples,delta_samples],
    data_labels = ['Proporción A', 'Proporción B', 'Diferencia'],
    shade=.2, colors = ['green','red','blue'] ,hdi_prob=.95,
)
plt.vlines(true_p_A, 0, 80, linestyle="--",
           label="Verdadero $p_A$ (desconocido)", color ='green')
plt.vlines(true_p_B, 0, 80, linestyle="--",
           label="Verdadero $p_B$ (desconocido)", color ='red')
plt.vlines(true_p_A-true_p_B, 0, 80, linestyle="--",
           label="Verdadero $\delta$ (desconocido)", color ='blue')
plt.legend()
plt.title('')
plt.savefig("img/4_CB/density_proporcion.svg"); plt.close()
```

Convergencia



Densidades (todas las cadenas)



¿Por qué es más "gorda" la distribución de B? ¿Por qué tenemos menos certeza? ¿Qué tiene que ver los datos?

In [26]:

```python
# El área bajo la curva <0 es la probabilidad
# que el sitio A tenga una menor proporción que
# el sitio B.
print("Probabilidad proporcion A menor a B: %.3f" % \
    np.mean(delta_samples < 0))

print("Probabilidad proporcion A mayor a B: %.3f" % \
    np.mean(delta_samples > 0))
```

```
Probabilidad proporcion A menor a B: 0.064
Probabilidad proporcion A mayor a B: 0.936
```

¿Son esas probabilidades suficientes para decidir? Debatir

Acabamos de comparar dos proporciones con técnicas bayesianas.

El tamaño muestral se incluye automaticamente en el análisis: menor certeza para p_B.

Podemos hacer las preguntas que queramos de las muestras e.g. moda, areas bajo la curva, etc.

OTRO EJEMPLO BINARIO: MENTIRAS

Situación: un investigador le interesa la tasa de trampa en una universidad

Solución 1:

- preguntar a los estudiantes un si o un no (método directo)
    - problema: privacidad, desconfianza en promesa de no castigo.

Solución 2:

- lanzar moneda en privado.
    - Si cae cara, escriba la verdad.
    - Si cae sello vuelva lanzar. Escriba:
        - "sí hago trampa" si cae cara,
        - "no hago trampa" si cae sello.

Es privado. El experimentador no sabe la fuente de las respuestas. Pueden ser la verdad o el resultado del segundo lanzamiento.

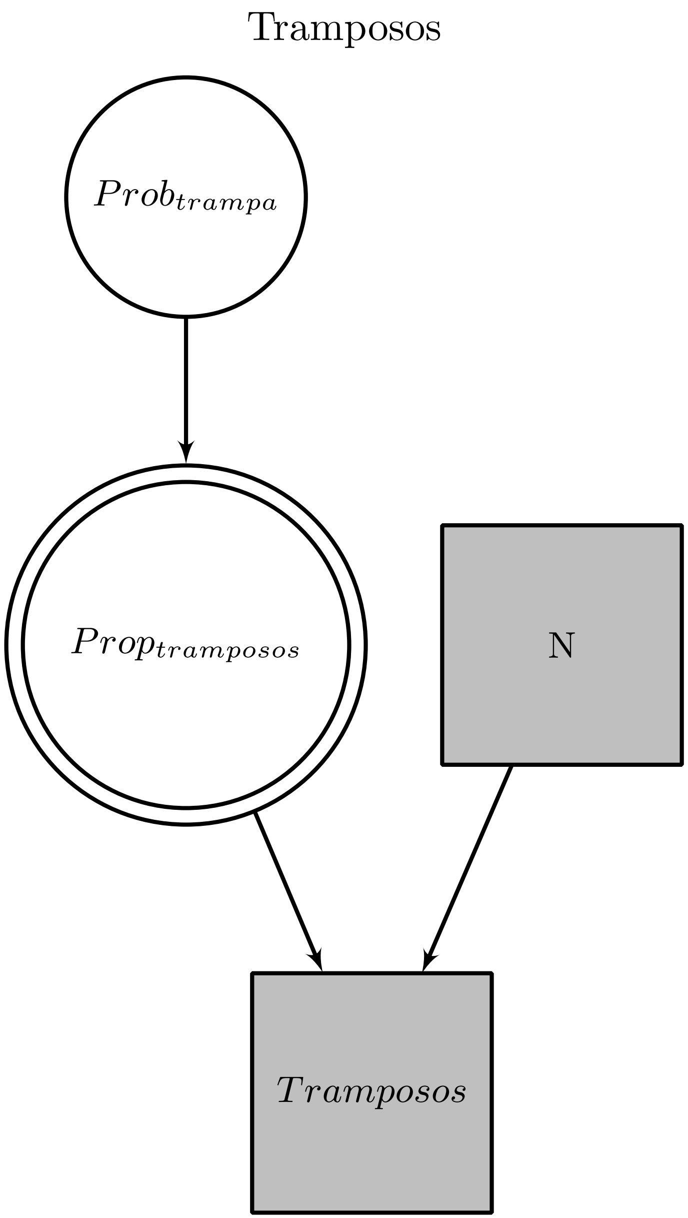
¿Qué distribución podemos usar para data binaria? Binomial

$$P(X = k|p, N) = \binom{N}{k} p^k (1-p)^{N-k}$$

k: éxitos
N: intentos
p: probabilidad éxito

En lenguaje natural:
Si conozco la probabilidad del evento (cara) y el número de intentos (veces que se lanza la moneda), sé la probabilidad de número de éxitos.

¡A modelar!



$$Prob_{trampa} \sim Uniforme(0,1)$$

$$Prop_{tramposos} = 0.5 * Prob_{trampa} + 0.5^2$$

$$Tramposos \sim Binomial(N, Prop_{tramposos})$$

In [4]:

```
prob_cara = 0.5
prob_sello = 0.5
with pm.Model() as model:
    prob_trampa = pm.Uniform("prob_trampa", 0, 1) #Prior uniforme
    prop_trampo = pm.Deterministic(
        "prop_tramposos",
        prob_cara*prob_trampa + prob_sello*prob_cara
    )
```

Ahora, definamos un likelihood binomial para la data.

In [5]:

```
# Data
N = 100 #Número de estudiantes
trampo = 35 #Número de reportes que dicen hacer trampa
with model:
    tramposos = pm.Binomial("obs", N, prop_trampo,
                            observed=trampo)

# Ya podemos samplear
with model:
    step = pm.Metropolis(vars=[prob_trampa])
    trace = pm.sample(40000, step=step)
    burned_trace = trace[15000:]
```

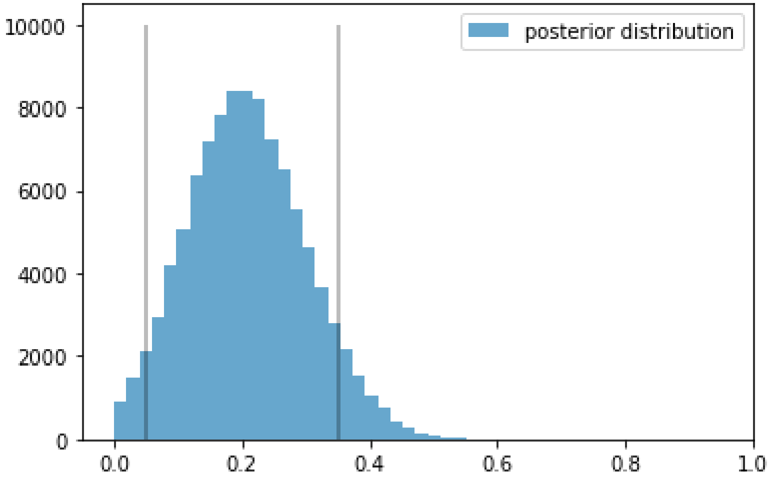
```
Multiprocess sampling (4 chains in 4 jobs)
Metropolis: [prob_trampa]
```

100.00% [164000/164000 00:51<00:00 Sampling 4 chains, 0 divergences]

```
Sampling 4 chains for 1_000 tune and 40_000 draw iterations (4_000 + 160_000 d
raws total) took 53 seconds.
The number of effective samples is smaller than 25% for some parameters.
```

In [6]:

```
p_trace = burned_trace["prob_trampa"]
plt.hist(p_trace, histtype="stepfilled", alpha=0.75, bins=30,
         label="posterior distribution", color="#348ABD")
plt.vlines([.05, .35], [0, 0], [10000, 10000], alpha=0.35)
plt.xlim(-0.05, 1)
plt.legend();
```

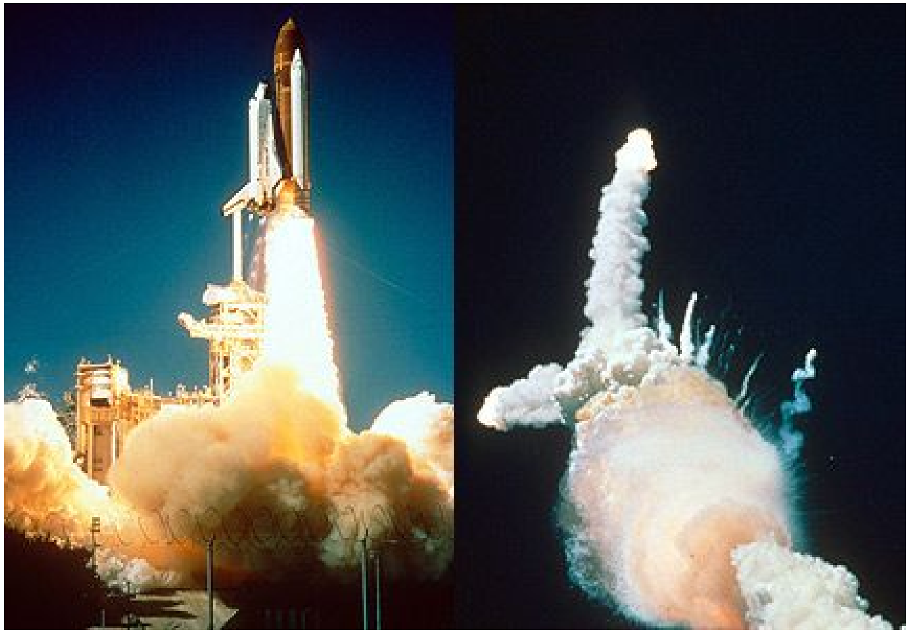


PRO TIP

Podemos crear un modelo con muchas variables con un for loop.
Hay que inicializar un array con tipo de dato `objeto`

In [7]:
```
N = 10
x = np.ones(N, dtype=object)
with pm.Model() as model:
    for i in range(0, N):
        x[i] = pm.Exponential('x_%i' % i, (i+1.0)**2)
```

OTRO EJEMPLO: DESASTRE DEL CHALLENGER



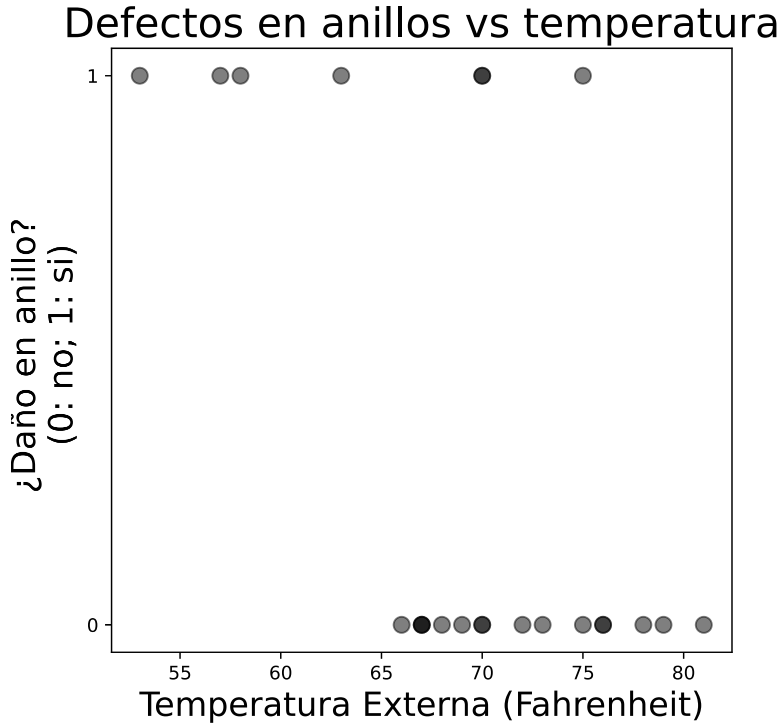
Potencial razón:

Daño de un anillo en una conexión de uno de los impulsores de un cohete (i.e. se daño una pieza)

In [8]:

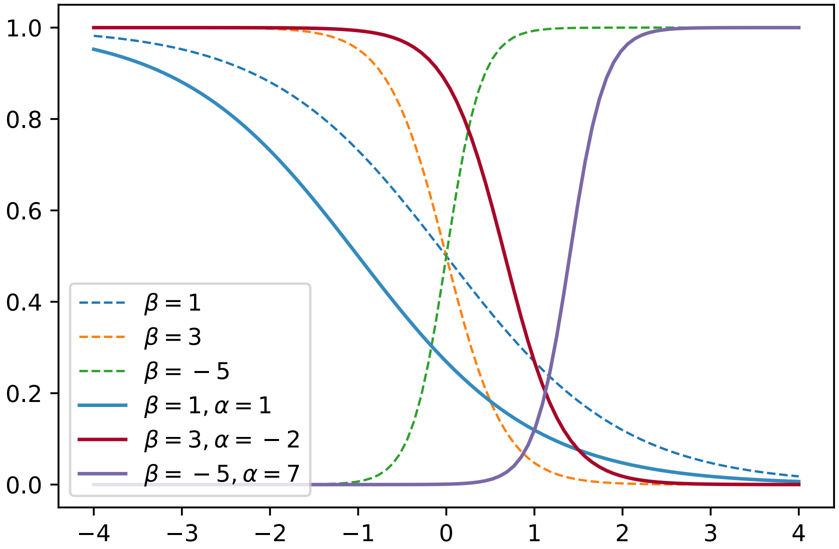
```
#Cargar data
#Filas: 23 vuelos previos.
#Col 1: temperatura externa (fahrenheit);
#Col 2: falla de anillo (0 no, 1 si)
np.set_printoptions(precision=3, suppress=True)
challenger_data = np.genfromtxt("data/4_CB/challenger_data.csv",
                                skip_header=1,
                                usecols=[1, 2], missing_values="NA",
                                delimiter=",")
#drop the NA values
challenger_data = challenger_data[~np.isnan(challenger_data[:, 1])]
```

¿Qué relación vemos entre temperatura ambiental y daño del anillo?

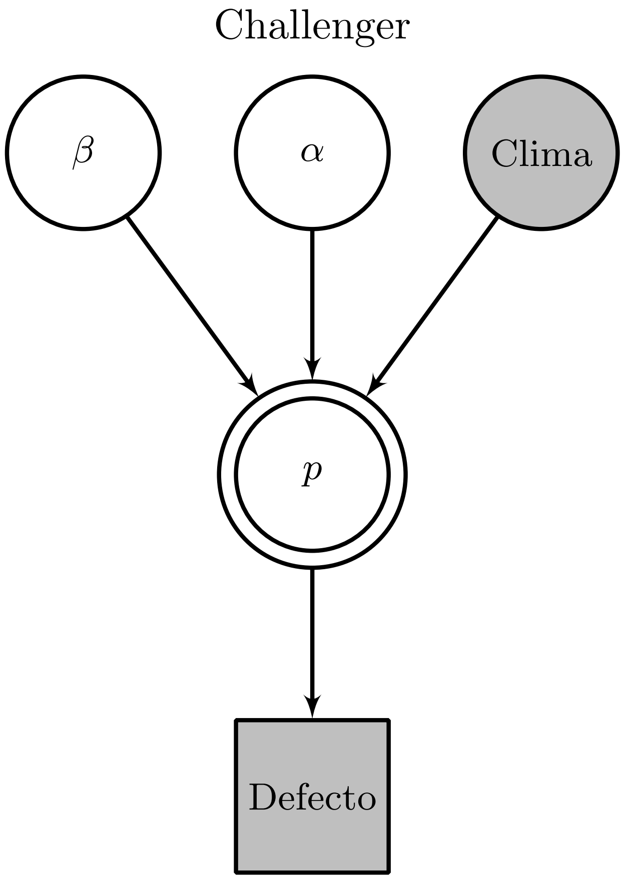


Podemos modelar variables binarias con la función logística

$$p(t) = \frac{1}{1 + e^{\beta t + \alpha}}$$



Este es el modelo bayesiano



$$\beta \sim Normal(\mu = 0, \sigma^2 = 1000)$$

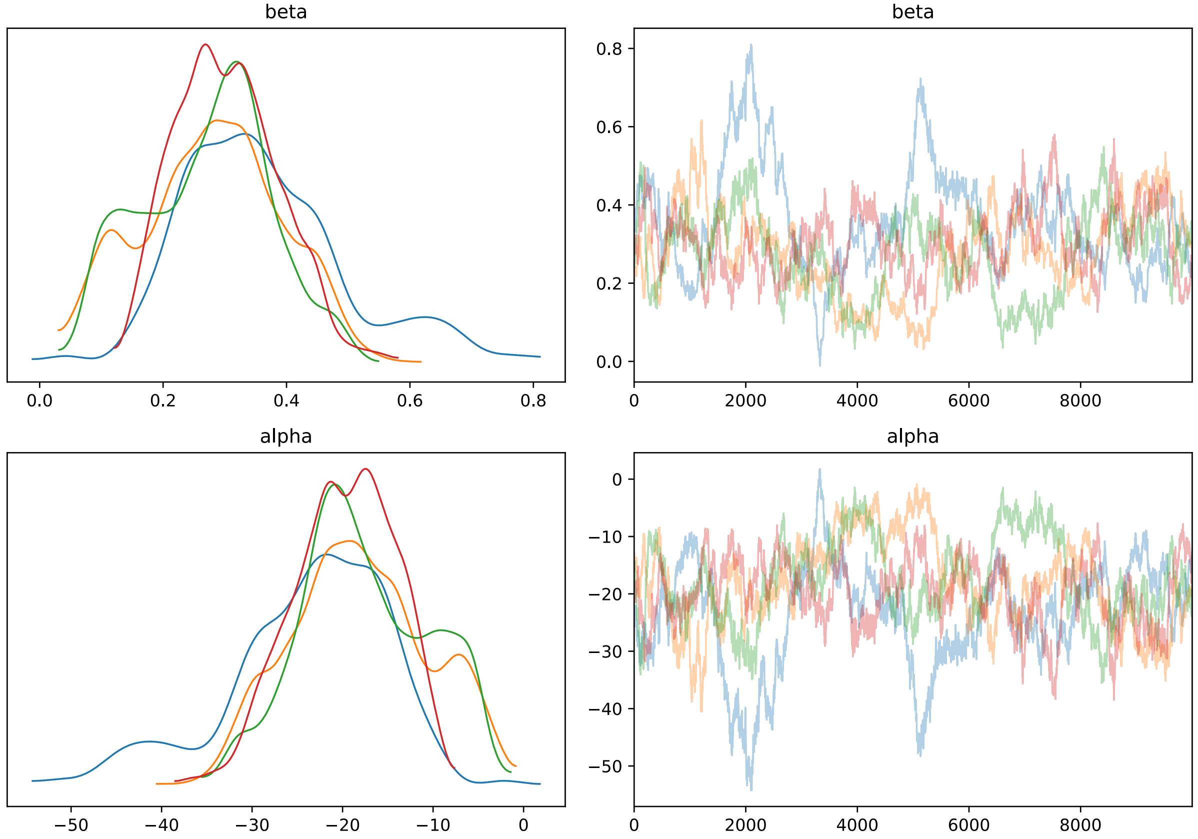
$$\alpha \sim Normal(\mu = 0, \sigma^2 = 1000)$$

$$p = \frac{1}{1+e^{\beta \times \text{Clima} + \alpha}}$$

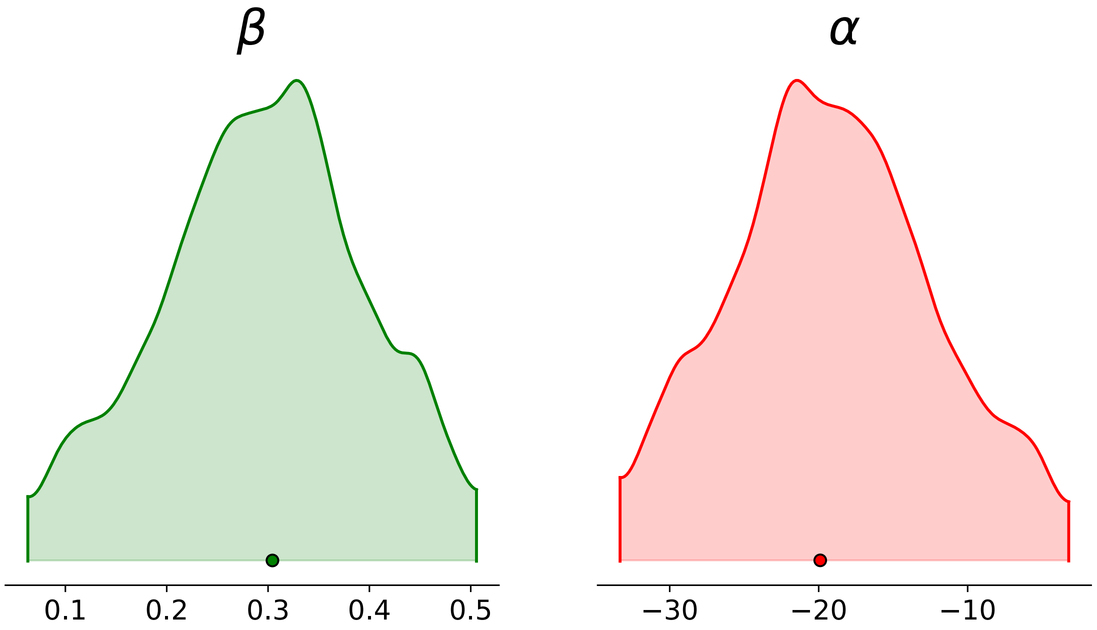
$$Defecto \sim Bernoulli(p)$$

In [ ]:
```
temperature = challenger_data[:, 0]
D = challenger_data[:, 1]  # defecto (0 no, 1 si)

with pm.Model() as model:
    #Caveat computacional: beta y alpha empiezan en cero para evitar
    # que p se vaya a las esquinas 0 o 1 desde el comienzo

    beta = pm.Normal("beta", mu=0, tau=0.001, testval=0)
    alpha = pm.Normal("alpha", mu=0, tau=0.001, testval=0)
    p = pm.Deterministic("p",
                         1.0/(1. + tt.exp(beta*temperature + alpha)))
    defecto = pm.Bernoulli("defecto", p, observed=D)

    # Sampleo
    start = pm.find_MAP() #Max. a posteriori con valores iniciales
    step = pm.Metropolis()
    trace = pm.sample(120000, step=step, start=start)
    burned_trace = trace[100000::2]
```

El modelo combina relativamente bien



Notable que β, la importancia de la temperatura, no incluya cero. La temperatura sí jugó un rol.



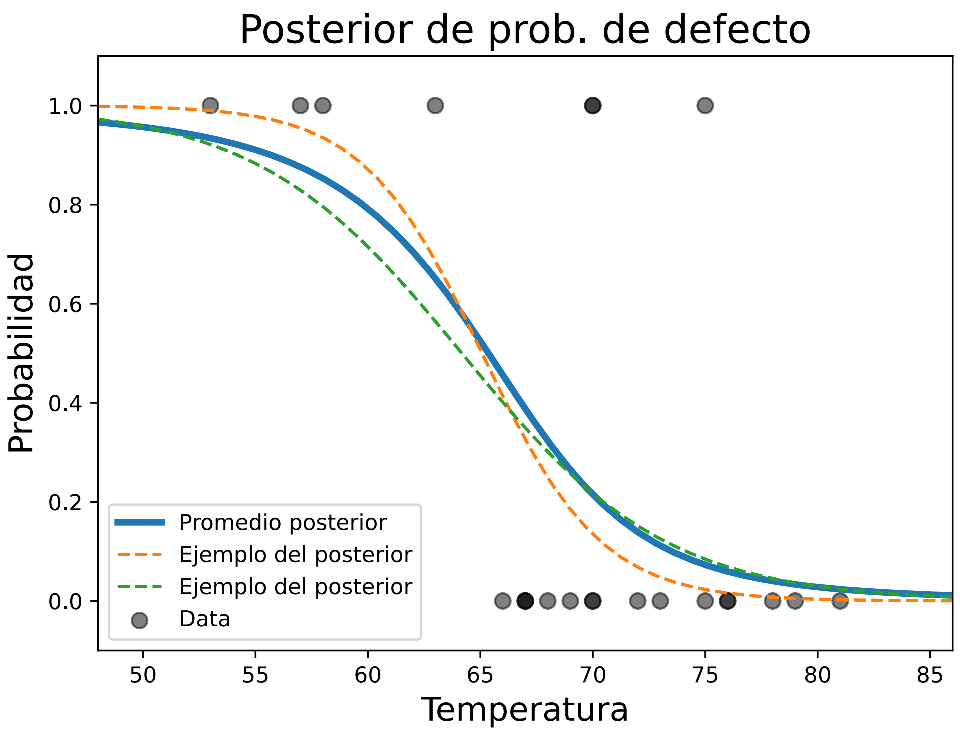
In [ ]:

```
Con las posterior podemos obtener valores de p
```

In [ ]:

```
def logistic(x, beta, alpha=0):
    return 1.0 / (1.0 + np.exp(np.dot(beta, x) + alpha))

t = np.linspace(temperature.min() - 5,
                temperature.max()+5, 50)[:, None] #[:, None] lo vuelve
 1D

p_t = logistic(t.T, beta_samples[:, None],
               alpha_samples[:, None])
mean_prob_t = p_t.mean(axis=0)
```

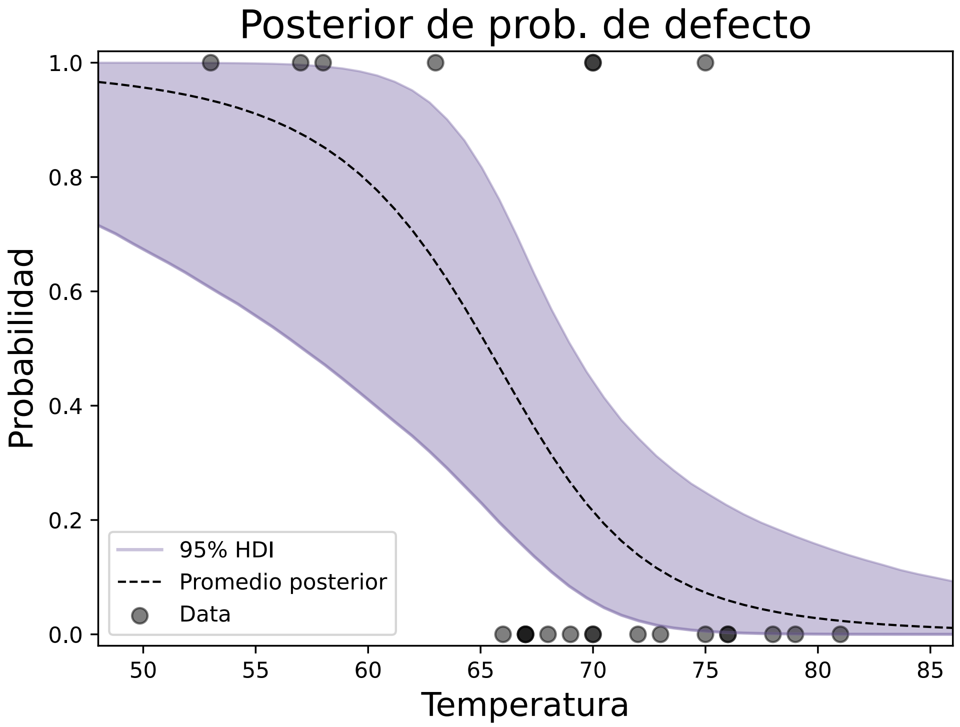
Posterior: promedio y algunos ejemplos. Generamos muchas versiones de la hipótesis logística.



Podemos incluso generar intervalos de certidumbre

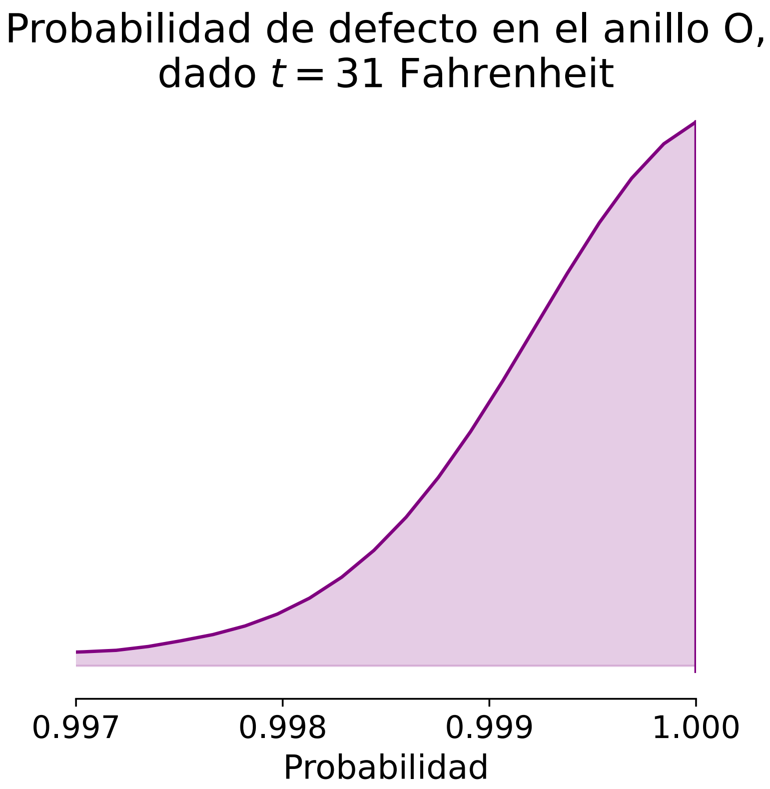
In [ ]:
```
#95 HDI
qs = mquantiles(p_t, [0.025, 0.975], axis=0)
```

Podemos saber donde tenemos más incertidumbre: temperaturas entre 50 y 70. Enfocarnos en recoger datos en esa zonas.



Podemos poner probabilidades para la temperatura que hizo el día del desastre: 31 Fahrenheit

In [ ]:
```
prob_31 = logistic(31, beta_samples, alpha_samples)
```



EN RESUMEN ...

Usamos PyMC para tomar muestras.

La sintaxis básica es

```
with pm.Model() as ELNOMBREQUEQUIERA:
    distribuciones y sampleador
```