

Programare probabilistă folosind PyMC

Cursul 4

Programare și modelare probabilistă - anul III

Facultatea de Informatică, UAIC

e-mail: `adrian.zalinescu@uaic.ro`

web: `https://sites.google.com/view/fiicoursepmp/home`

30 Octombrie 2023

- 1 Introducere
- 2 Aruncarea monedelor cu PyMC
 - Rezumatul distribuției a posteriori
 - Decizii bazate pe distribuția a posteriori
 - Funcția de pierdere
- 3 Sintaxa și variabilele PyMC
 - Cadrul obiectului Model
 - Variabilele PyMC
 - Includerea observațiilor în Model
- 4 Abordări în modelare

Statistica Bayesiană este conceptual foarte simplă:

- avem *cunoscute* și *necunoscute*;
- folosim *formula lui Bayes* pentru a condiționa pe cele din urmă în raport cu cele dintâi, sperând să reducem incertitudinea asupra necunoscutelor.

În ciuda acestei simplități, modelele pur probabiliste duc adesea la expresii insolubile din punct de vedere analitic.

Dezvoltarea metodelor numerice și începutul erei computaționale a transformat dramatic practica Bayesiană de analiză a datelor.

Putem gândi metodele numerice folosite pentru rezolvarea problemelor de inferență ca *motoare universale de inferență*, sau așa cum spunea Thomas Wiecki, dezvoltator principal al PyMC, *butonul de inferență*.

În cadrul unui limbaj probabilistic (PPL), utilizatorii specifică un model probabilistic prin scrierea câtorva linii de cod, după care procesul de inferență decurge în mod automat. *PyMC* este o bibliotecă pentru programare probabilistă, care asigură o sintaxă apropiată de cea folosită în literatura statistică pentru a descrie modele probabiliste:

- codul de bază este scris folosind *Python*;
- părțile care sunt computațional solicitante folosesc *NumPy* și *PyTensor*, care la rândul ei este bazată pe *Theano*;
- *Theano* este o bibliotecă Python folosită pentru că unele metode de generare, precum *NUTS*, au nevoie de calcul al gradientilor, iar Theano știe să calculeze gradientii, folosind *diferențierea automată*.

Aruncarea monedelor cu PyMC

Mai întâi: generarea datelor (în viața reală, trebuie să le colectăm)

```
import numpy as np
import scipy.stats as stats

trials = 4
theta_real = 0.35 # unknown value in a real experiment
data = stats.bernoulli.rvs(p=theta_real, size=trials)

▷ array([1, 0, 0, 0])
```

Modelul:

$$\begin{aligned}\theta &\sim \text{Beta}(\alpha, \beta) \\ y &\sim \text{Bernoulli}(p = \theta)\end{aligned}$$

```
import pymc as pm

with pm.Model() as our_first_model:
    # a priori
    theta = pm.Beta('theta', alpha=1., beta=1.)
    # likelihood
    y = pm.Bernoulli('y', p=theta, observed=data)
    idata = pm.sample(1000, random_seed=123, return_inferencedata=True)
```

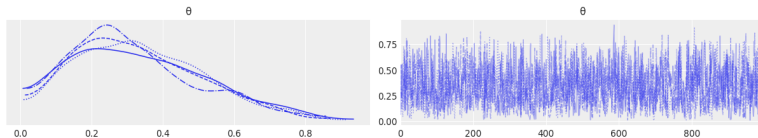
Rezumatul distribuției a posteriori

Pentru vizualizarea rezultatelor:

```
import matplotlib.pyplot as plt
import arviz as az
```

Verificăm cum arată rezultatele cu:

- `az.plot.trace(...)`



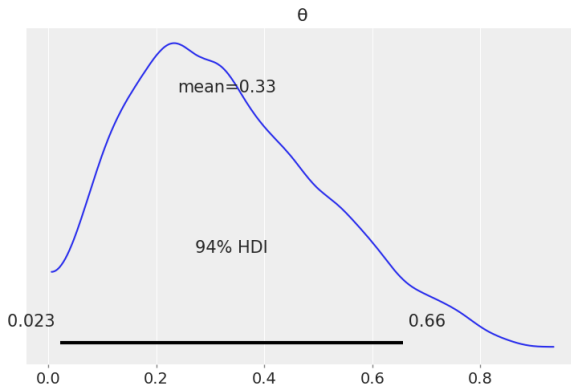
- ▶ La dreapta, avem valorile individuale ale valorilor eșantionate la fiecare pas;
- ▶ La stânga, o versiune netezită a histogramei obținută pe baza valorilor eșantionate, numită “Kernel Density Estimation (KDE)”.

- `az.summary(...)`

```
▶      mean    sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
θ  0.331  0.18  0.023   0.656    0.005    0.003   1354.0   1668.0    1.0
```

Rezumatul distribuției a posteriori

O altă posibilitate de a vizualiza distribuția a posteriori este dată de `az.plot_posterior(...)`, care arată o histogramă pentru variabilele discrete și un grafic KDE pentru cele continue:

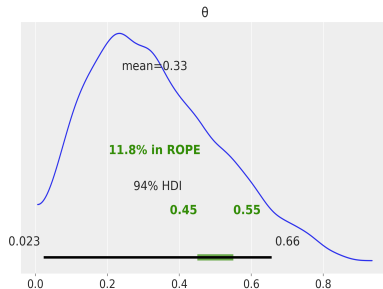


HDI înseamnă *High Density Interval*, altă denumire pentru HPD

Decizii bazate pe distribuția a posteriori

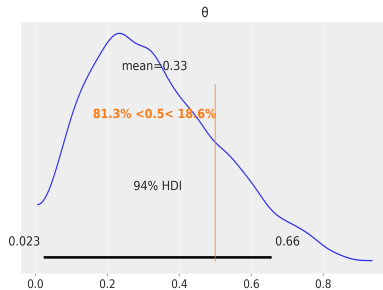
- Câteodată, a descrie distribuția a posteriori nu este de ajuns și trebuie să luăm decizii bazate pe inferență.
- Așadar, trebuie să reducem estimarea continuă la una *dihotomistă*, de tip da/nu, sănătos/bolnav, înalt/scund, etc.
- În exemplul anetrior, am putea decide doar dacă moneda este nemăsluită sau măsluită.
- Din punct de vedere practic, în loc să cerem ca $\theta = 0.5$ (probabilitatea $P(\theta = 0.5|y) = 0$), am putea afirma că moneda este nemăsluită dacă θ este într-un anume interval, de ex. $[0.45, 0.55]$.
- Numim acest interval *ROPE* (*Region Of Practical Equivalence*):
 - ▶ dacă $\text{ROPE} \cap \text{HDI} = \emptyset$, putem spune că moneda este măsluită;
 - ▶ dacă $\text{HDI} \subseteq \text{ROPE}$, putem spune că moneda este nemăsluită;
 - ▶ nu putem spune nimic în celelalte cazuri.

- `az.plot_posterior(idata, rope=[0.45, .55])`



- O altă posibilitate:

`az.plot_posterior(idata, ref_val=0.5)`



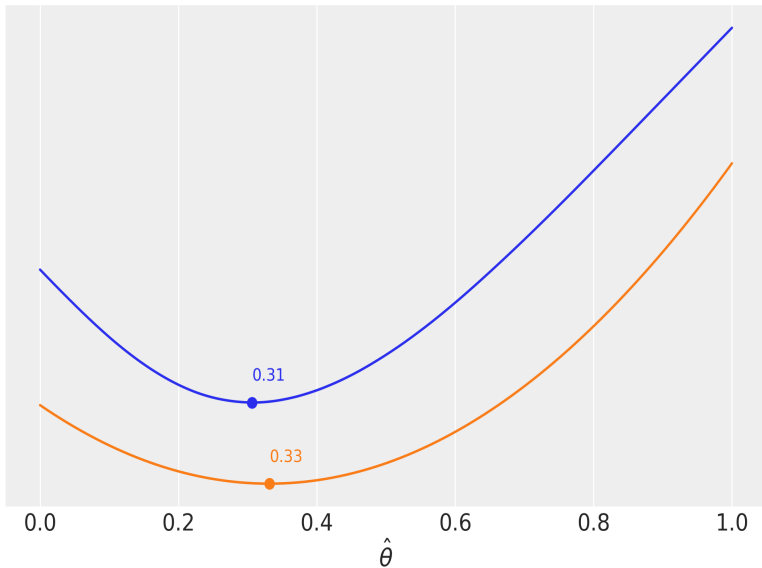
Funcția de pierdere

- Pentru a lua o decizie bună, este important să avem cel mai mare nivel de precizie pentru parametrii estimați.
- Analiza cost/beneficii poate fi făcută folosind *funcția de pierdere* (loss function) $L(\theta)$.
Exemple (θ este adevărata valoare, $\hat{\theta}$ este parametrul estimat):
 - ▶ pierdere pătratică: $L(\hat{\theta}) := (\theta - \hat{\theta})^2$;
 - ▶ pierdere absolută: $L(\hat{\theta}) := |\theta - \hat{\theta}|$;
 - ▶ pierdere 0 – 1: $L(\hat{\theta}) := I(\theta \neq \hat{\theta})$, unde I este funcția indicator.
- O estimare bună presupune o valoare mică a lui L .
- În cazul nostru, nu dispunem de valoarea θ , ci de distribuția a posteriori $P(\cdot|y)$.
- Căutăm $\hat{\theta} := \operatorname{argmin}_{\hat{\theta}} \mathbb{E}L(\hat{\theta})$, unde media \mathbb{E} se calculează în raport cu probabilitatea a posteriori.

Analizăm 2 funcții de pierdere, pătratică și absolută, explorând $\hat{\theta}$ pe un grid de 200 de valori din $[0, 1]$:

```
grid = np.linspace(0, 1, 200)
theta_pos = idata.posterior[' $\theta$ ']
lossf_a = [np.mean(abs(i - theta_pos)) for i in grid]
lossf_b = [np.mean((i - theta_pos)**2) for i in grid]

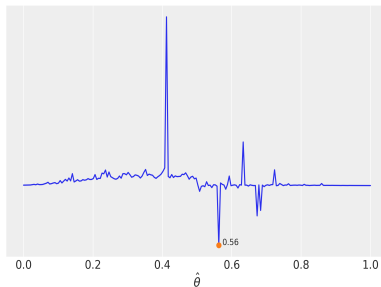
for lossf, c in zip([lossf_a, lossf_b], ['C0', 'C1']):
    mini = np.argmin(lossf)
    plt.plot(grid, lossf, c)
    plt.plot(grid[mini], lossf[mini], 'o', color=c)
    plt.annotate('{:.2f}'.format(grid[mini]),
                 (grid[mini], lossf[mini] + 0.03), color=c)
    plt.yticks([])
    plt.xlabel(r'$\hat{\theta}$')
plt.show()
```



observăm că cele două puncte de minim corespund mediei aritmetice, respectiv mediane valorilor eșantionate din distribuția a posteriori

- ▶ Dacă problema o cere, putem construi funcții de pierdere asimetrice, care ne permit să penalizăm/recompensăm anumite valori ale parametrului;
- ▶ De asemenea, putem calcula funcții de pierdere complicate, după cum arată acest exemplu arbitrar:

```
lossf = []
for i in grid:
    if i < 0.5:
        f = np.mean(np.pi *  $\theta_{pos}$  / np.abs(i -  $\theta_{pos}$ ))
    else:
        f = np.mean(1 / (i -  $\theta_{pos}$ ))
    lossf.append(f)
    ...
```



Cadrul obiectului Model

```
import pymc3 as pm
```

```
with pm.Model() as model:  
    parameter = pm.Exponential("poisson_param", 1.0)  
    data_generator = pm.Poisson("data_generator", parameter)
```

- Toate variabilele create cu un anumit Model vor fi automat atribuite acelu model;
- O variabilă definită în afara cadrului unui model va rezulta într-o eroare;
- Putem continua să lucrăm în cadrul acelui model folosind `with` cu numele obiectului model pe care l-am creat deja:

```
with model:  
    data_plus_one = data_generator + 1
```

- Odată ce au fost definite, variabilele pot fi folosite și în afara cadrului aceluși model.
- Toate variabilele atribuite unui model vor fi definite cu propriul nume, un parametru (primul) de tip string. Exemplu:

```
with pm.Model() as ab_testing:
    p_A = pm.Uniform("P(A)", 0, 1)
    p_B = pm.Uniform("P(B)", 0, 1)

# sau (redefinirea modelului cu același nume)
```

```
ab_testing = pm.Model()
```

```
with ab_testing:
    p_A = pm.Uniform("prob_A", 0, 1)
    p_B = pm.Uniform("prob_B", 0, 1)
```

PyMC are de a face cu două tipuri de variabile: *stochastice* și *deterministe*:

- *Variabilele stochastice* sunt cele care, chiar dacă știm valorile tuturor parametrilor și componentelor acestora, valoarea lor va fi tot aleatoare: de exemplu, Poisson, DiscreteUniform, Exponential.
- *Variabilele deterministe* sunt variabilele care nu sunt aleatoare atunci când cunoaștem valorile tuturor parametrilor și componentelor acestora: acestea pot fi create apelând clasa `Deterministic` în PyMC.

Inițializarea unei variabile stochastice presupune un argument de tip name, la care se adaugă parametri adiționali ce sunt specifici tipului de variabilă. De exemplu:

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

- Documentația PyMC (<https://www.pymc.io/welcome.html>) conține parametrii specifici pentru variabilele stochastice.
- Atributul name este folosit pentru apelarea ulterioară a distribuției a posteriori; cel mai bine este, deci, de a folosi un nume descriptiv (ce poate fi chiar numele variabilei).

- În problemele în care sunt folosite mai multe variabile de același tip, în loc de crearea unui vector (array) de variabile, se poate folosi parametrul shape. Se creează astfel un *array* ce se comportă ca un *NumPy* array. Exemplu: în loc de

```
beta_1 = pm.Uniform("beta_1", 0, 1)
beta_2 = pm.Uniform("beta_2", 0, 1)
...
```

se poate folosi

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

Variabile deterministe

Putem crea variabile deterministe într-un mod similar celor stochastice prin apelarea clasei `Deterministic`:

```
deterministic_variable = pm.Deterministic("variabila_determinista",  
                                          o_functie_de_variabile)
```

Apelarea `PyMC.Deterministic` este alegerea evidentă, dar nu singura pentru definirea variabilelor deterministe. Pot fi folosite operații elementare, ca adunarea, exponențierea, etc. pentru a crea variabile deterministe. Exemplu:

```
with pm.Model() as model:  
    lambda_1 = pm.Exponential("lambda_1", 1.0)  
    lambda_2 = pm.Exponential("lambda_2", 1.0)  
    tau = pm.DiscreteUniform("tau", lower=0, upper=10)  
  
new_deterministic_variable = lambda_1 + lambda_2
```

Dar dacă dorim o variabilă deterministă care să poată fi urmărită de eșantionarea folosită în inferență, trebuie să o introducem explicit cu clasa `Deterministic`.

În interiorul unei variabile deterministe, variabilele stochastice se comportă precum *scalarii* sau *NumPy* arrays. De exemplu, următorul cod este valid în PyMC:

```
def subtract(x, y):  
    return x - y  
  
stochastic_1 = pm.Uniform("U_1", 0, 1)  
stochastic_2 = pm.Uniform("U_2", 0, 1)  
  
det_1 = pm.Deterministic("Delta", subtract(stochastic_1, stochastic_2))
```

Includerea observațiilor în Model

Variabilele stochastice PyMC au argumentul `observed`. Acesta are un rol simplu: de a fixa valoarea curentă a variabilei la datele date, de regulă *NumPy* array sau *pandas* *DataFrame*:

```
import pymc as pm

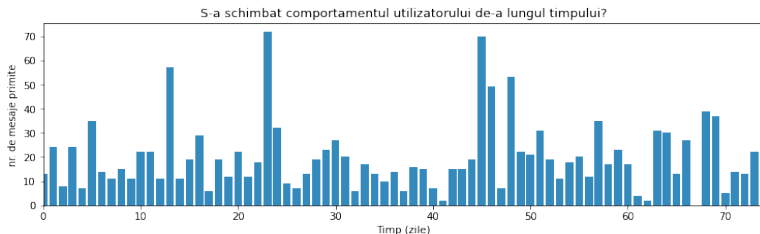
with pm.Model() as our_first_model:
    # a priori
     $\theta$  = pm.Beta('θ', alpha=1., beta=1.)
    # likelihood
    y = pm.Bernoulli('y', p= $\theta$ , observed=data)
    idata = pm.sample(1000, random_seed=123, return_inferencedata=True)
```

Abordări în modelare

Exemplu (din [BMH]):

Avem o serie de date ce numără mesajele SMS zilnice ale unui anumit utilizator:

```
plt.figure(figsize=(12.5, 3.5))
count_data = np.loadtxt("data/txtdata.csv")
n_count_data = len(count_data)
plt.bar(np.arange(n_count_data), count_data, color="#348ABD")
plt.xlabel("Timp (zile)")
plt.ylabel("nr. de mesaje primite")
plt.title("S-a schimbat comportamentul utilizatorului de-a lungul timpului?")
plt.xlim(0, n_count_data);
```



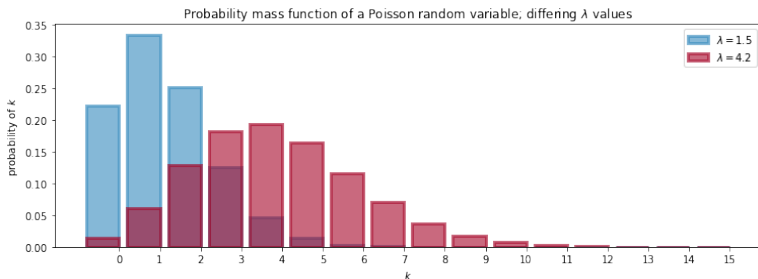
Începem prin a ne întreba cum ar putea fi aceste observații generate:

1. *Care este cea mai bună v.a. care ar descrie aceste date?*

O v.a. Poisson este un bun candidat, pentru că măsoară în general cantități ce pot fi numărate în intervale fixe de timp:

2. *Presupunând că nr. de SMS-uri este distribuit Poisson, de ce avem nevoie pentru această distribuție?*

Avem nevoie de un parametru λ :

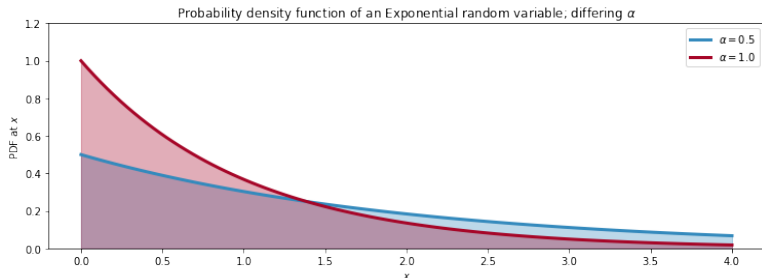


3. Îl știm pe λ ?

- Suspectăm că de fapt vor fi două valori ale lui λ , una pentru comportamentul inițial, cealaltă pentru cel final.
- Nu știm când comportamentul se schimbă, dar notăm acest moment cu τ .

4. Care este o distribuție bună pentru cei doi λ ?

- Distribuția exponențială este bună, pentru că ia valori pozitive (și $\lambda > 0$).
- Distribuția exponențială are și ea un parametru, să-l notăm α .



5. *Ce valoare are α ?*

- Am putea merge mai departe și să atribuim o distribuție lui α , dar ne oprim aici, pentru că am atins un anumit prag de ignoranță: deși pentru λ am putea anumite indicii, nu avem niciunul pentru α .
- De exemplu, am putea să-l setăm pe α astfel încât media variabilei care modelează pe λ , adică $1/\alpha$, să fie apropiată de media observată, în cazul nostru $\alpha = 1.0/\text{count_data.mean}()$.

6. *Ce valoare are τ ?*

Nu avem nici o opinie privind τ , așa că vom presupune că τ urmează o distribuție discret uniformă.

Modelul:

```
import pymc as pm

with pm.Model() as model:
    alpha = 1.0/count_data.mean() # Recall count_data is the
                                     # variable that holds our txt counts
    lambda_1 = pm.Exponential("lambda_1", alpha)
    lambda_2 = pm.Exponential("lambda_2", alpha)

    tau = pm.DiscreteUniform("tau", lower=0, upper=n_count_data - 1)
```

Următorul cod creează funcția

$$\lambda = \begin{cases} \lambda_1, & t \leq \tau; \\ \lambda_2, & t > \tau. \end{cases}$$

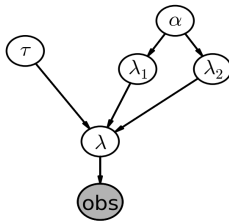
```
with model:
    idx = np.arange(n_count_data) # Index
    lambda_ = pm.math.switch(tau > idx, lambda_1, lambda_2)
```

La final, precizăm care sunt datele observate:

```
with model:
```

```
    observation = pm.Poisson("obs", lambda_, observed=count_data)
```


Sumarul modelului propus:



Inferența:

with model:

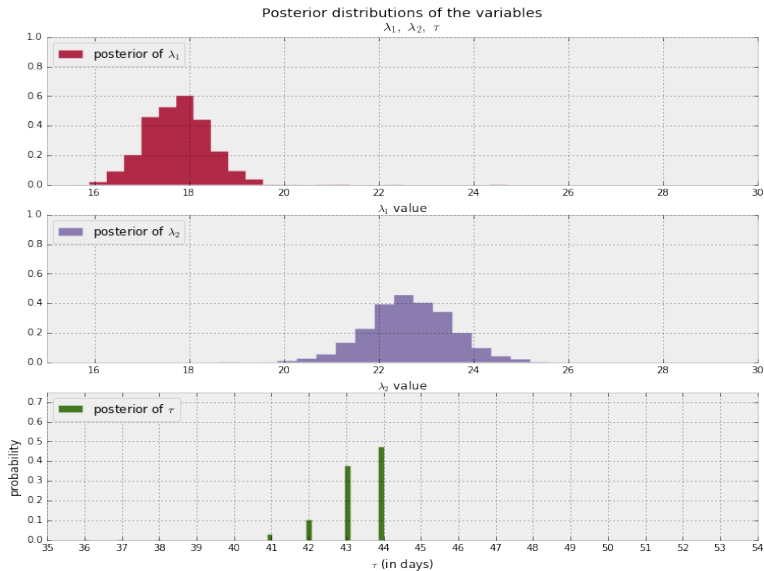
```
step = pm.Metropolis()  
trace = pm.sample(10000, tune=5000, step=step, return_inferencedata=False)
```

```
... Multiprocess sampling (4 chains in 4 jobs)  
CompoundStep  
>Metropolis: [lambda_1]  
>Metropolis: [lambda_2]  
>Metropolis: [tau]  
  
...  
...  100.00% [60000/60000 00:19<00:00 Sampling 4 chains, 0 divergences]  
  
... Sampling 4 chains for 5_000 tune and 10_000 draw iterations (20_000 + 40_000 draws total) took 33 seconds.
```

Eșantionare din λ_1 , λ_2 și τ :

```
lambda_1_samples = trace['lambda_1']  
lambda_2_samples = trace['lambda_2']  
tau_samples = trace['tau']
```

Rezumatul grafic al inferenței:

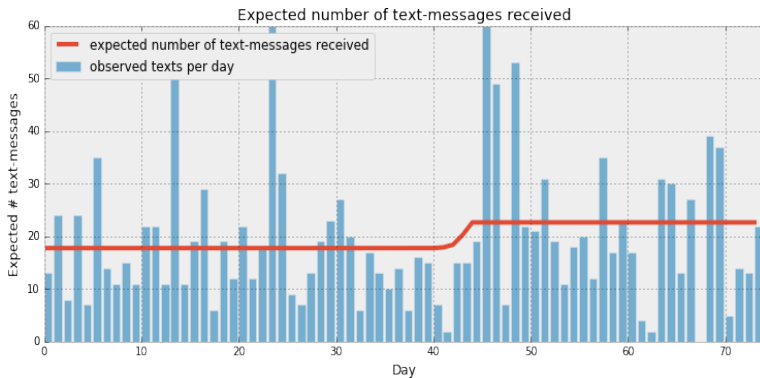


Valoarea medie a nr. de SMS-uri primite, pe zile

```
plt.figure(figsize=(12.5, 5))
# tau_samples, lambda_1_samples, lambda_2_samples contain
# N samples from the corresponding posterior distribution
N = tau_samples.shape[0]
expected_texts_per_day = np.zeros(n_count_data)
for day in range(0, n_count_data):
    ix = day < tau_samples
    expected_texts_per_day[day] = (lambda_1_samples[ix].sum()
                                   + lambda_2_samples[~ix].sum()) / N

plt.plot(range(n_count_data), expected_texts_per_day, lw=4, color="#E24A33",
         label="expected number of text-messages received")
plt.xlim(0, n_count_data)
plt.xlabel("Day")
plt.ylabel("Expected # text-messages")
plt.title("Expected number of text-messages received")
plt.ylim(0, 60)
plt.bar(np.arange(len(count_data)), count_data, color="#348ABD", alpha=0.65,
        label="observed texts per day")

plt.legend(loc="upper left");
```



Analiza arată probe în favoarea faptului că numărul (mediu) de mesaje s-a schimbat în jurul zilelor 41–44.