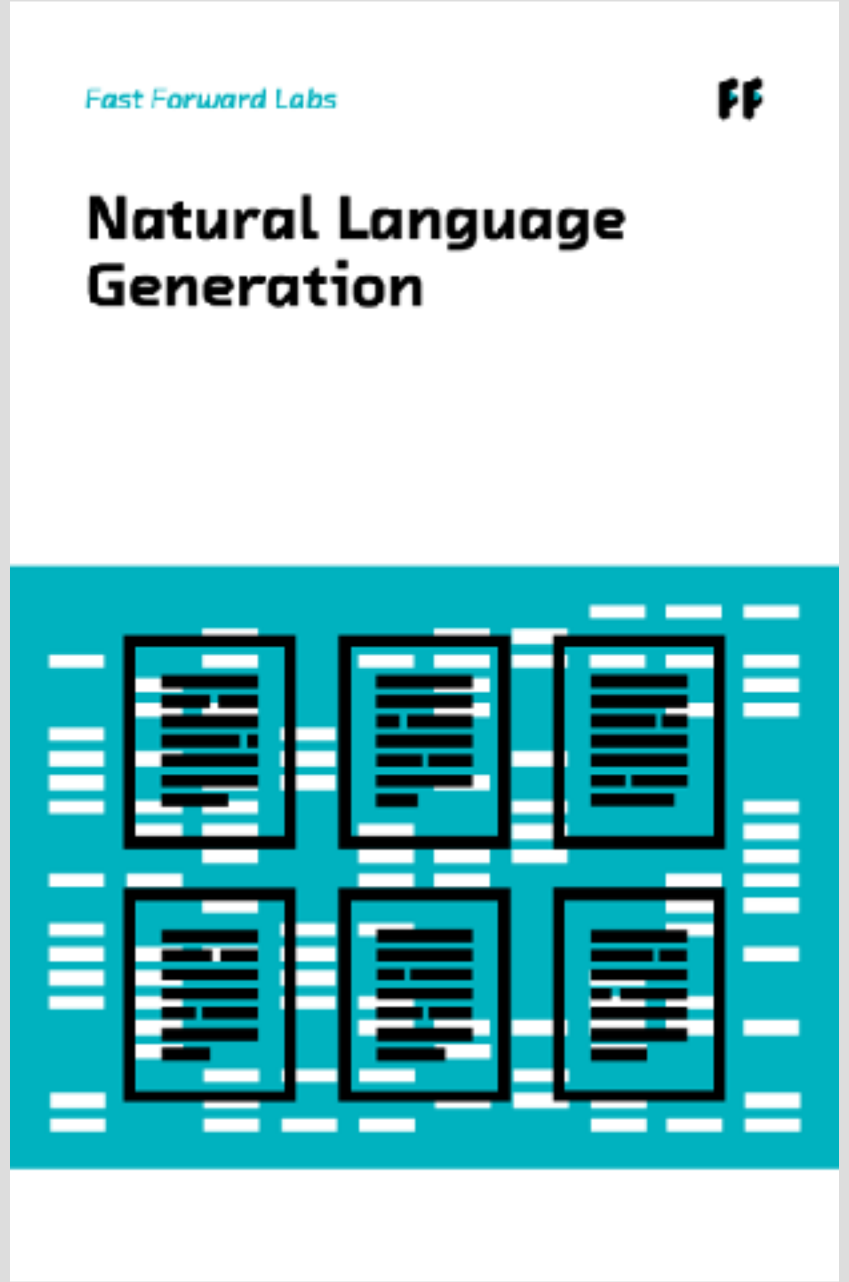
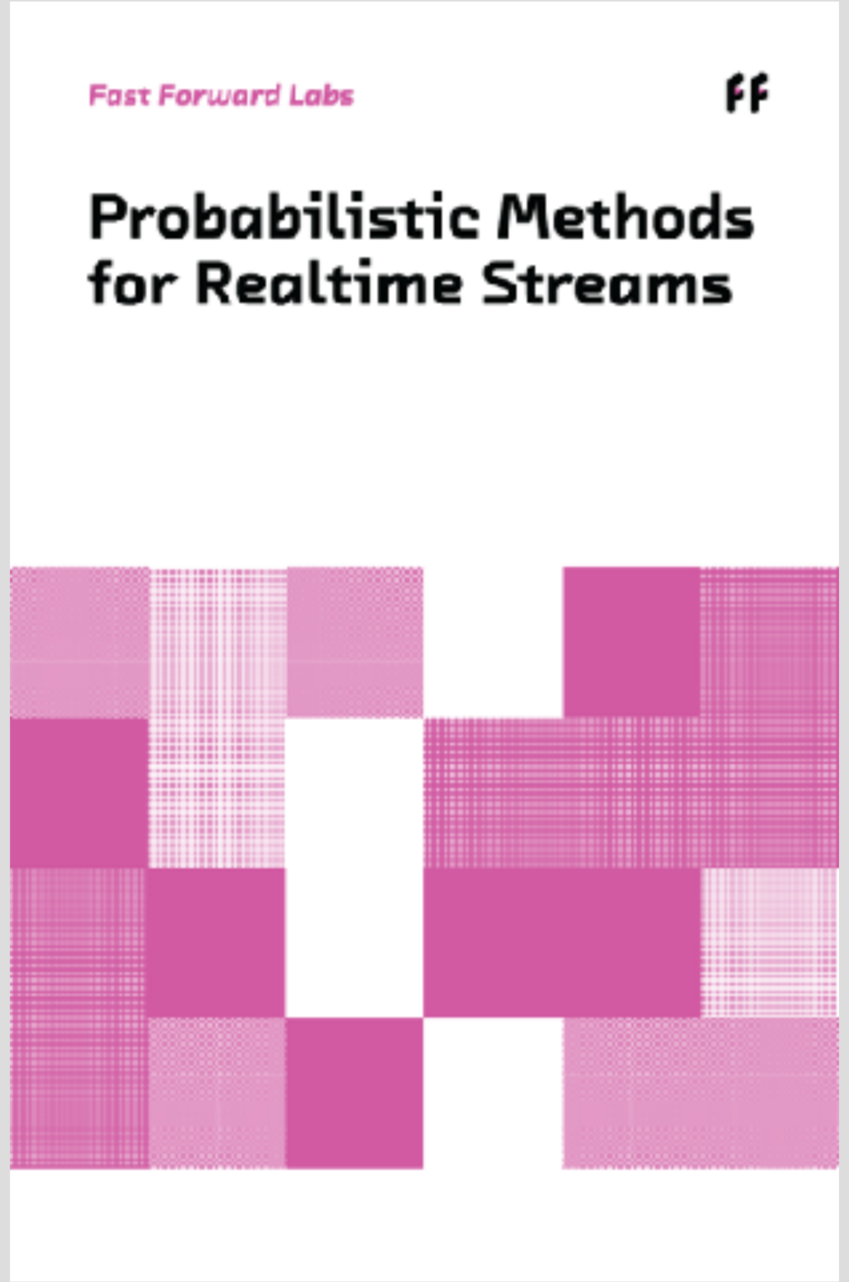
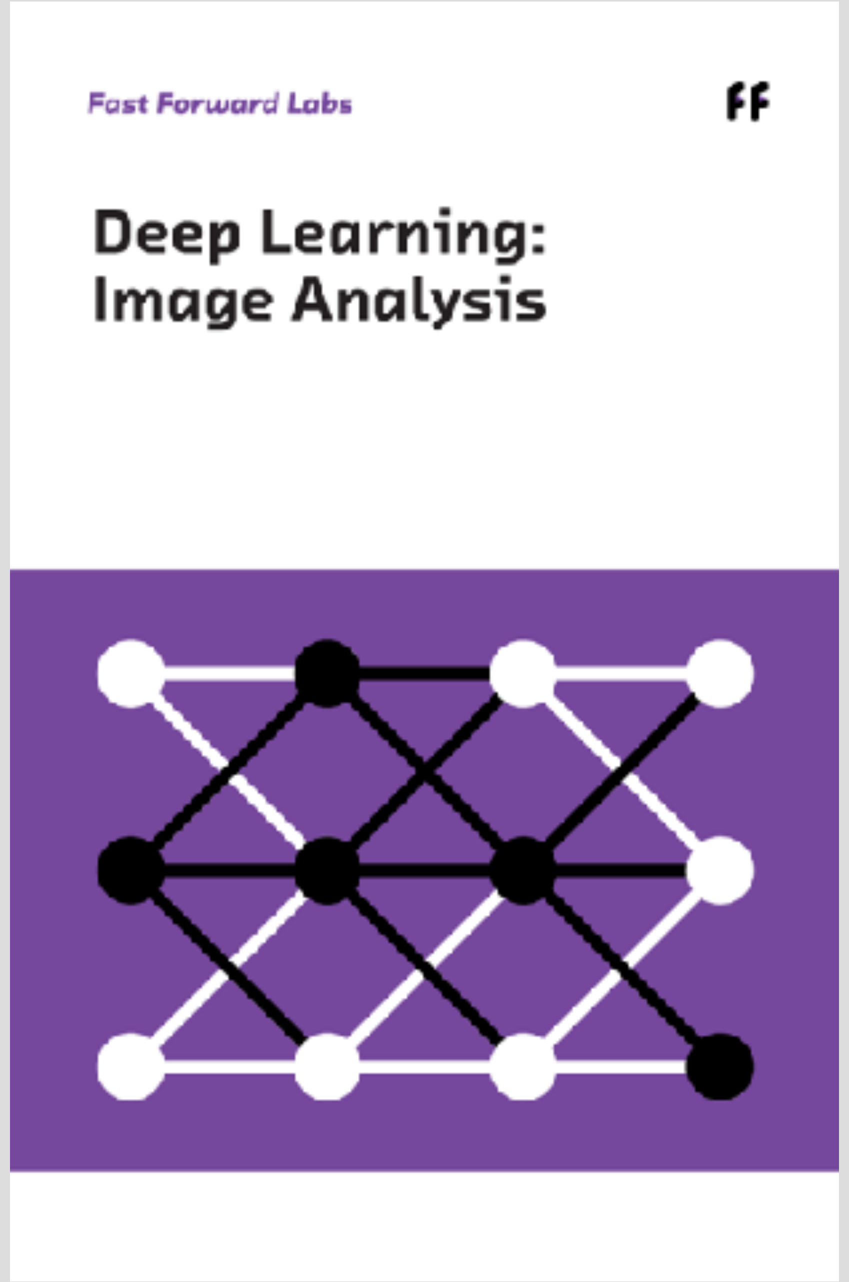
cloudera

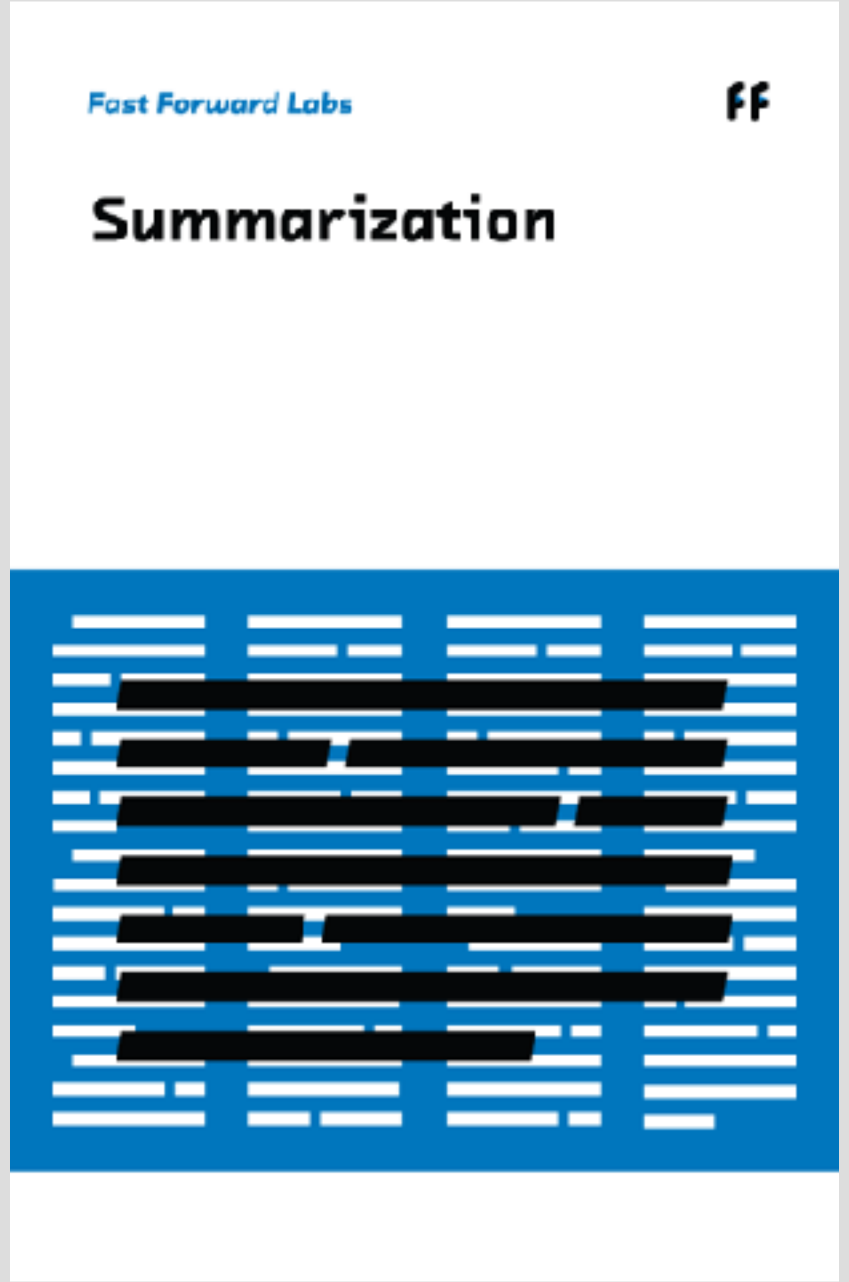
Probabilistic programming from scratch

Mike Lee Williams • Fast Forward Labs
@mikepqr • mlw@cloudera.com

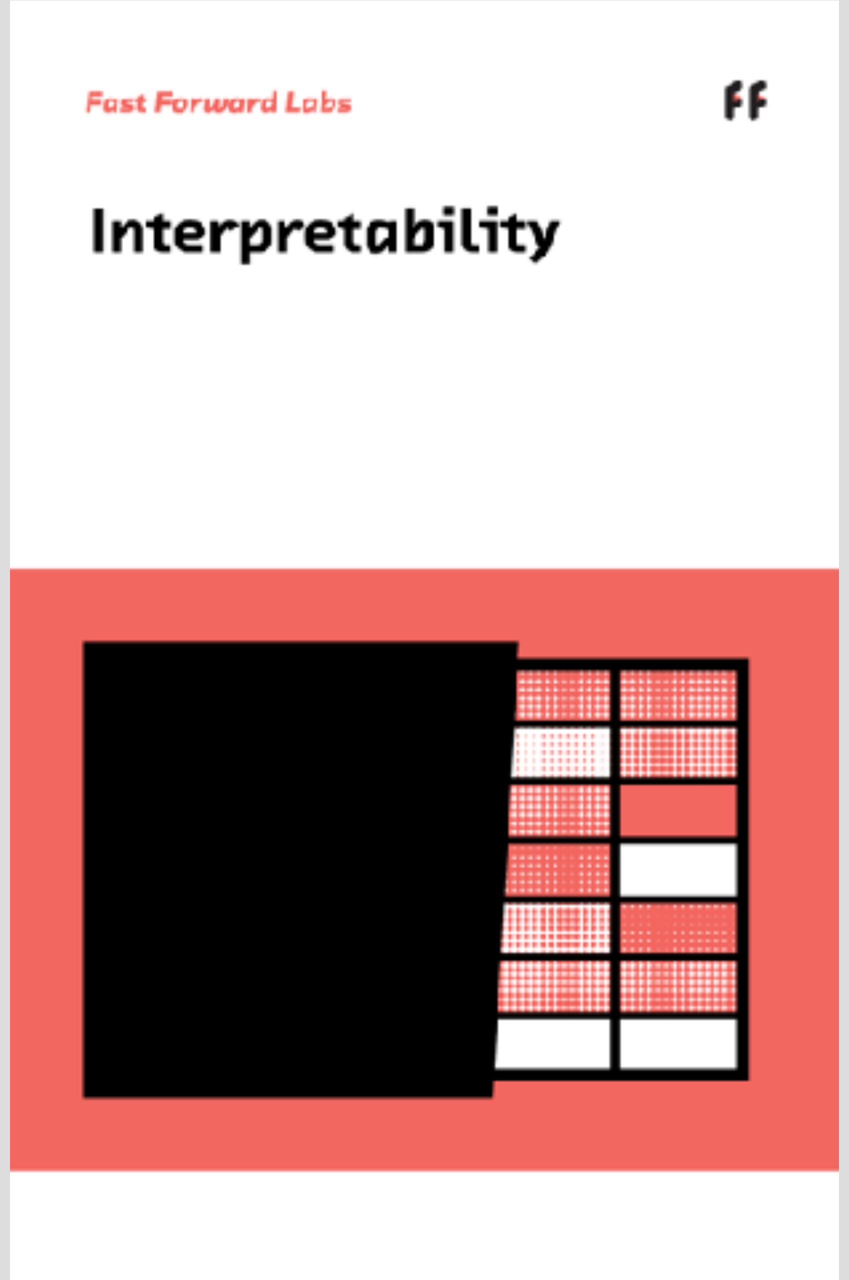

Fast Forward Labs
FF
Natural Language Generation

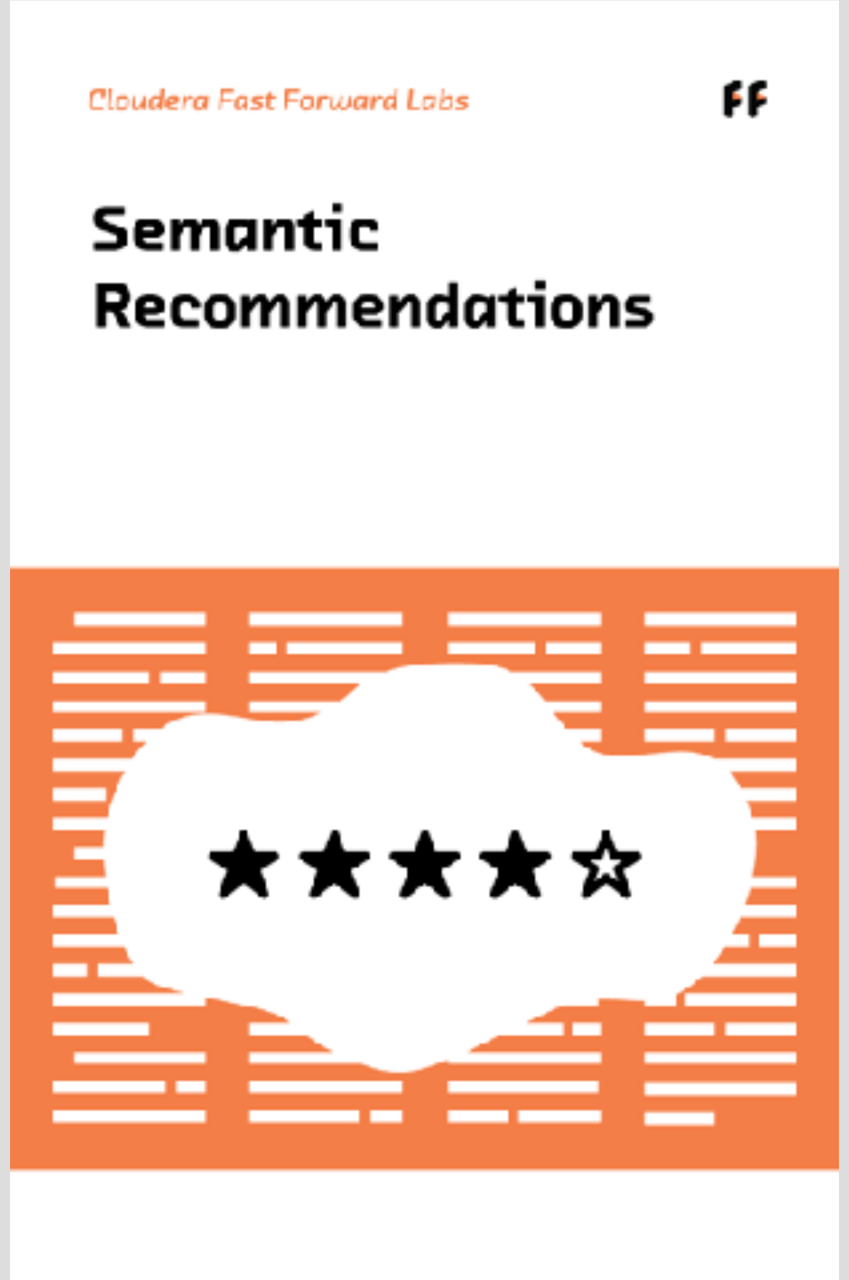

Fast Forward Labs
FF
Probabilistic Methods for Realtime Streams


Fast Forward Labs
FF
Deep Learning: Image Analysis


Fast Forward Labs
FF
Summarization


Fast Forward Labs
FF
Probabilistic Programming


Fast Forward Labs
FF
Interpretability


Cloudera Fast Forward Labs
FF
Semantic Recommendations

Bayesian inference is great in theory...

- Quantify risk
- Insert institutional knowledge
- Online learning

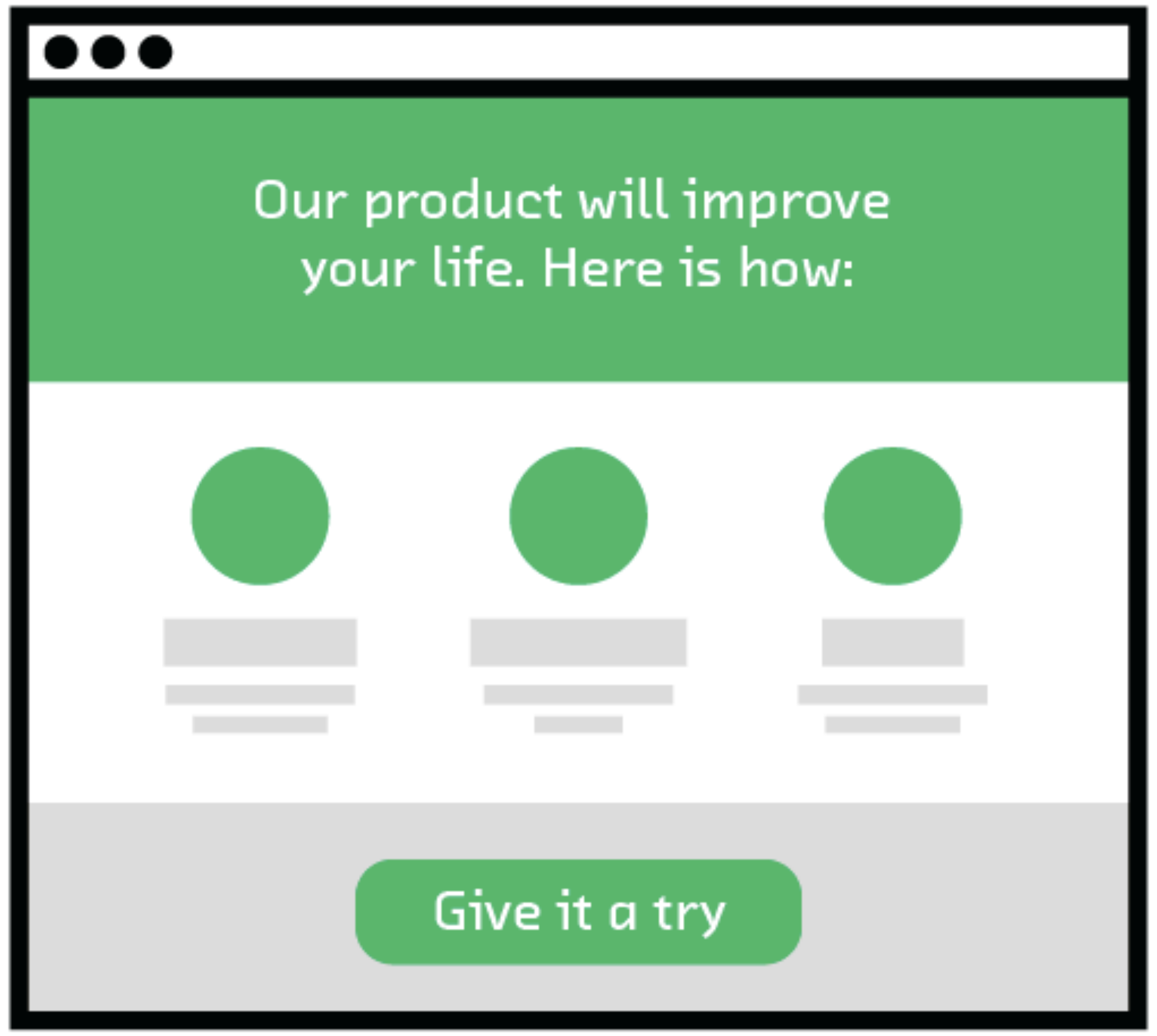
And it's pretty easy to implement from scratch

But fast implementations require cleverness…

- Metropolis Hastings
- Hamiltonian Monte Carlo with automatic differentiation and NUTS

the cleverness is now ready to abstracted away 😎

Bayesian inference is great in theory…

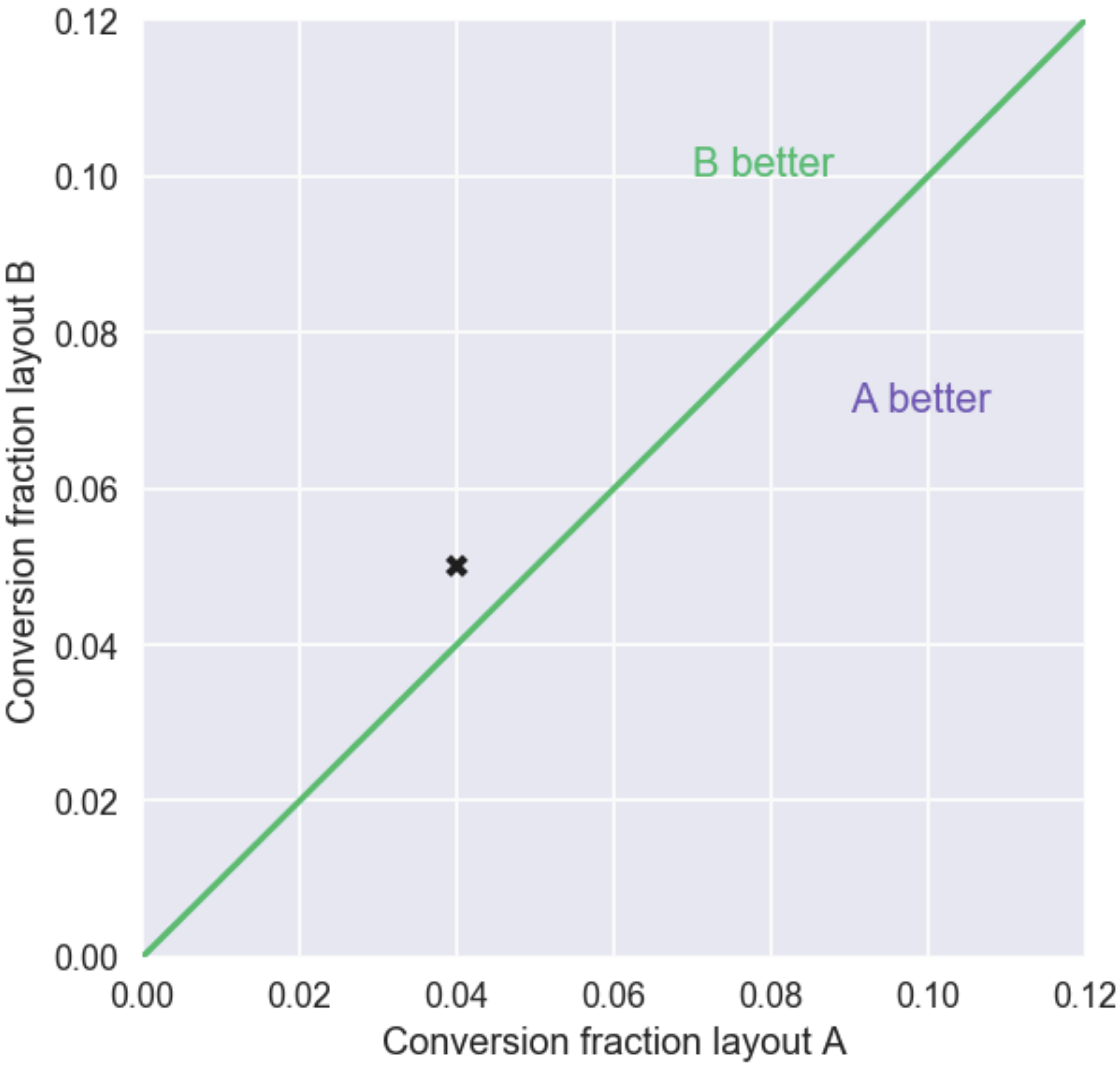


layout A

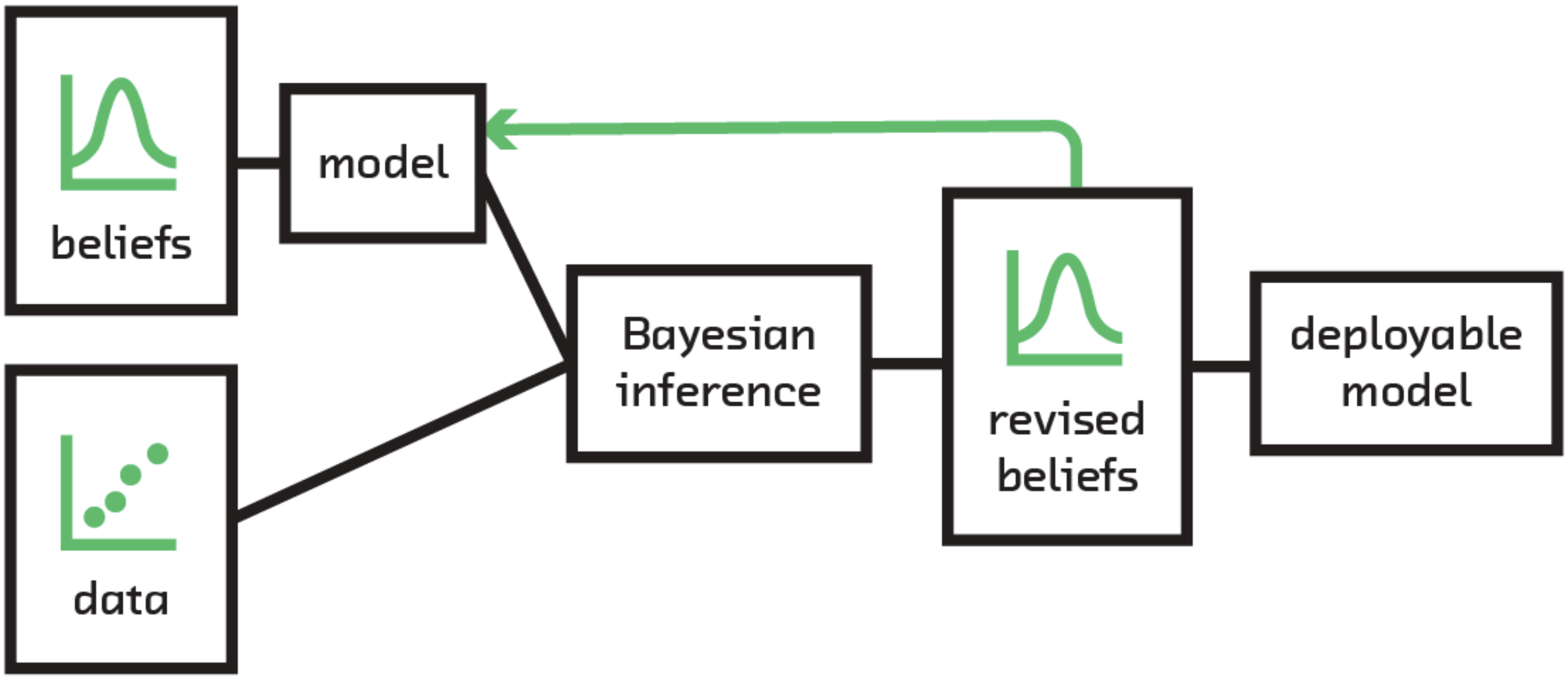
4% conversion rate

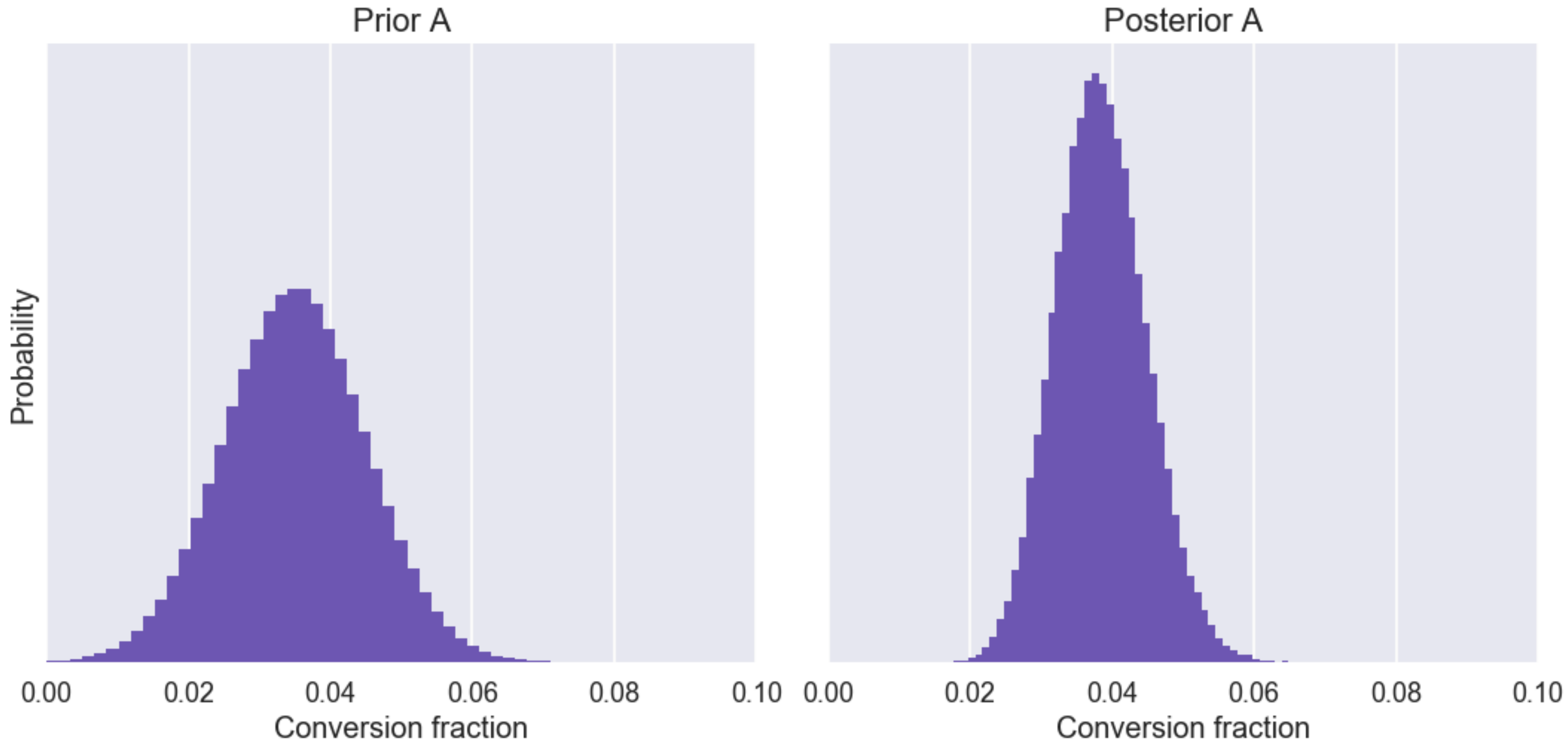
layout B

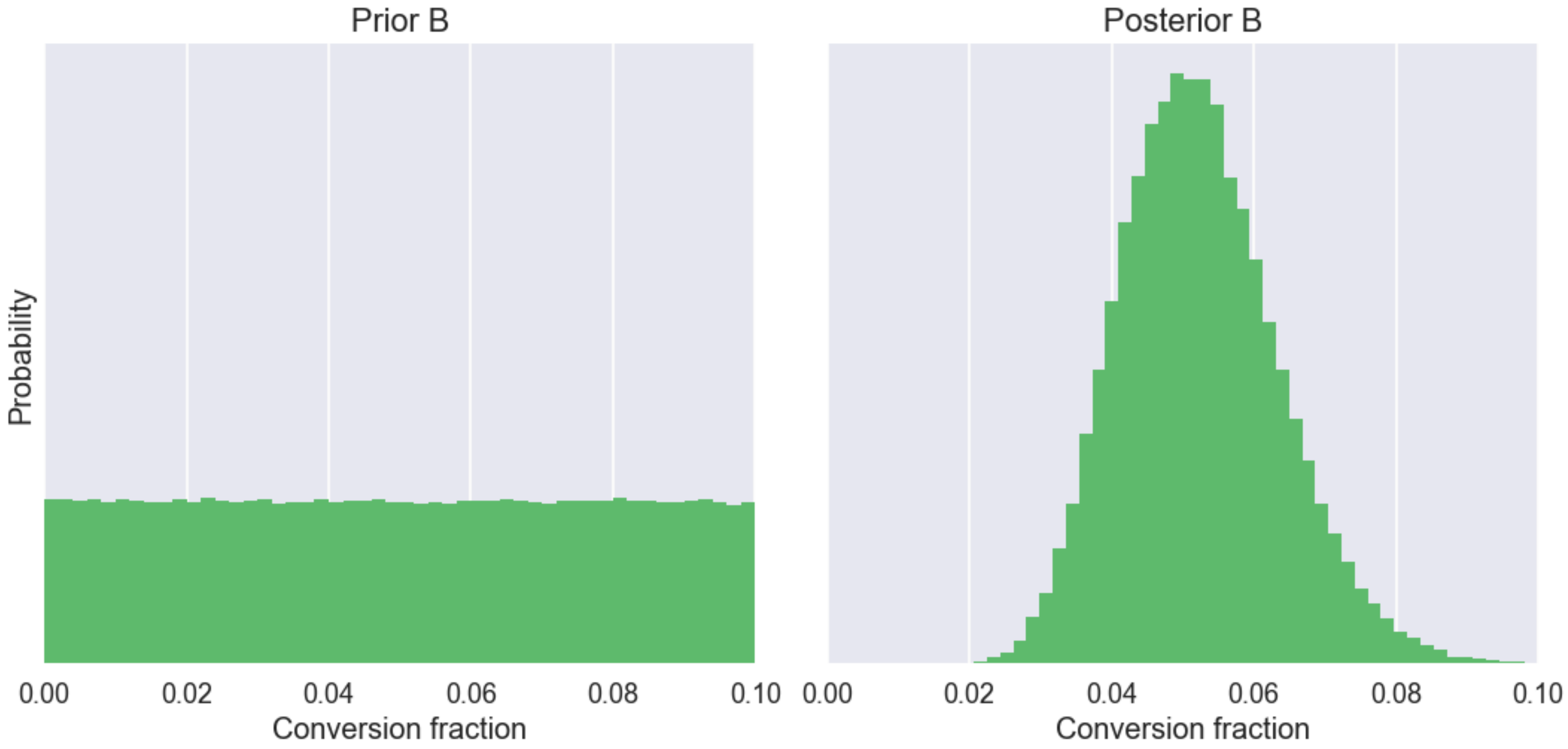
5% conversion rate

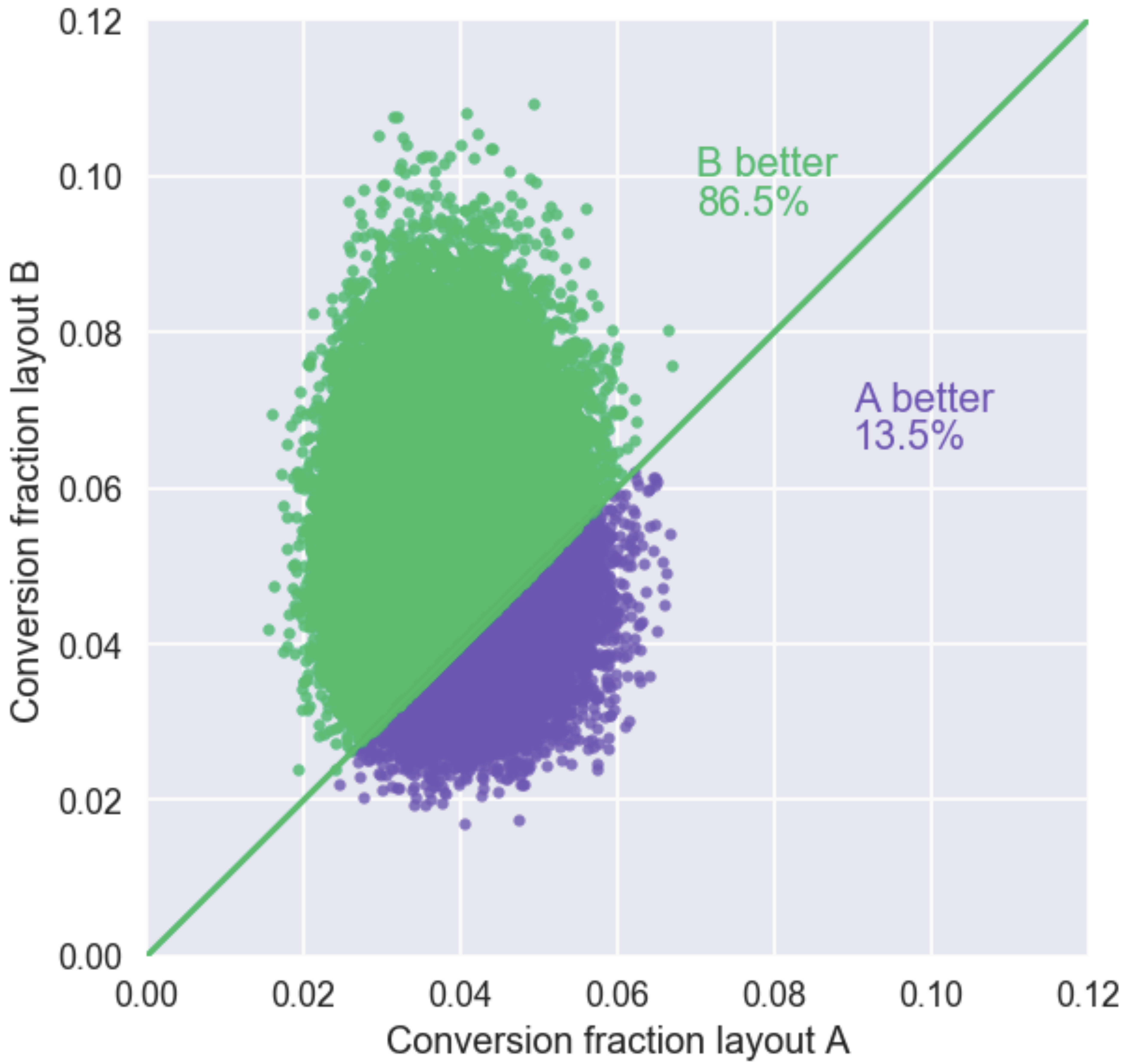

B better
A better
Conversion fraction layout B
Conversion fraction layout A
0.00
0.02
0.04
0.06
0.08
0.10
0.12


NO DATA'S
PERFECT

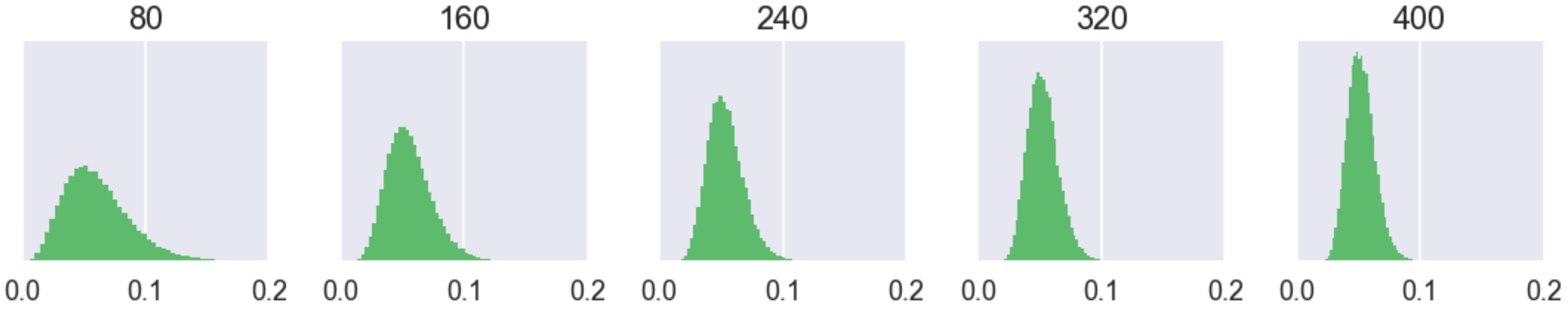

beliefs
model
data
Bayesian inference
revised beliefs
deployable model


Prior A
Posterior A
Probability
0.00
0.02
0.04
0.06
0.08
0.10
Conversion fraction
Conversion fraction


Prior B
Posterior B
Probability
0.00
0.02
0.04
0.06
0.08
0.10
Conversion fraction
0.00
0.02
0.04
0.06
0.08
0.10
Conversion fraction



B better
86.5%
A better
13.5%
0.00
0.02
0.04
0.06
0.08
0.10
0.12
Conversion fraction layout A
Conversion fraction layout B


80
160
240
320
400
0.0
0.1
0.2

Probabilistic programming from scratch

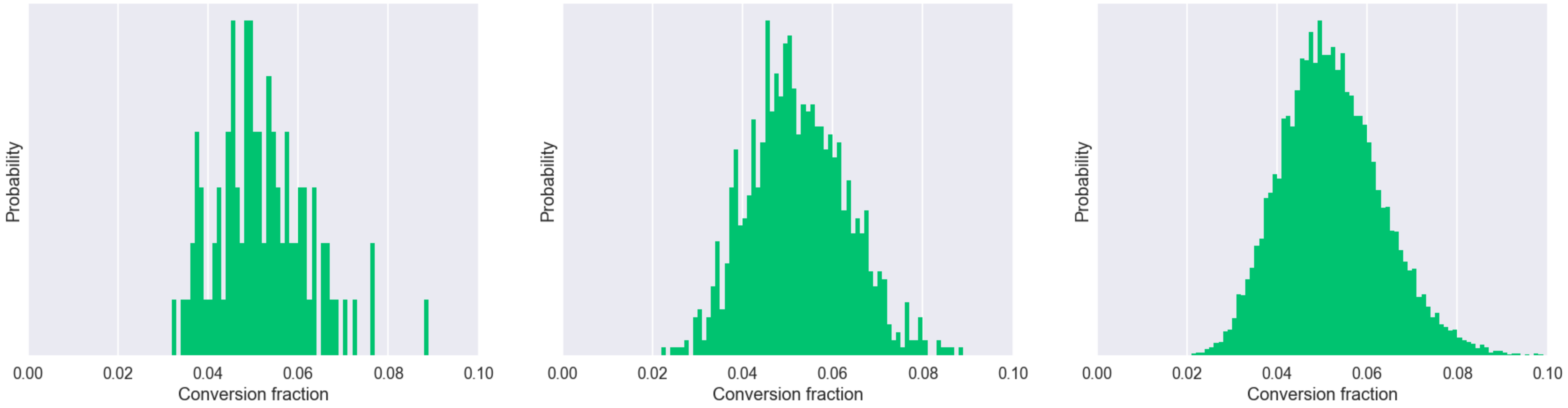
```python
def abayes(data, prior_sampler, simulate):
    """Yield samples from the posterior by Approximate Bayesian Computation."""

    # For each guess based on our prior beliefs
    for p in prior_sampler:

        # Simulate the experiment and see if it matches the real data
        if simulate(p) == data:

            # If it does, it was a good guess!
            yield p
```

```
n_converted = 20
N = 400
```

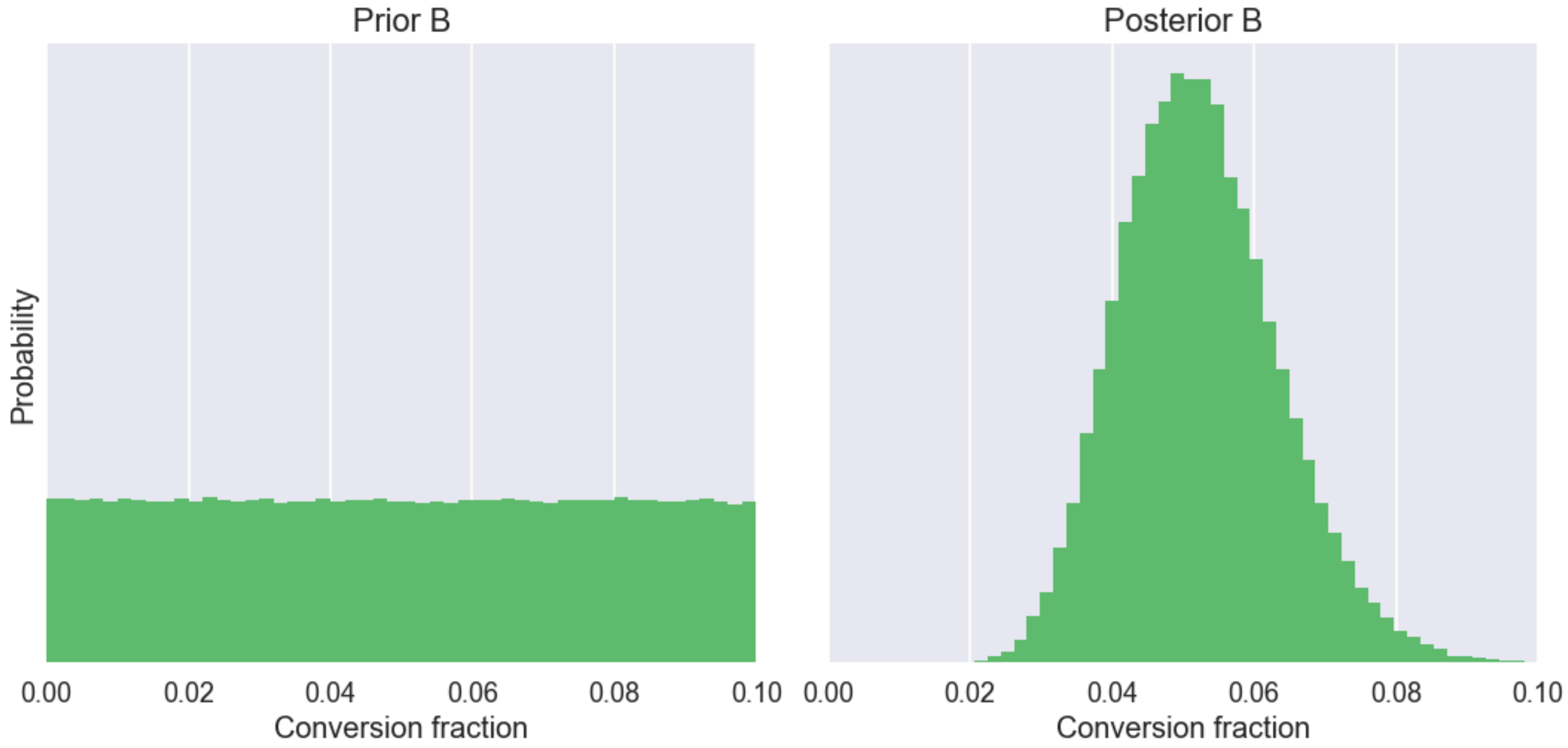
```python
from random import random

def uniform_prior_sampler():
    """Yield stream of random numbers in interval (0, 1)."""
    while True:
        yield random()


>>> s = uniform_prior_sampler()


>>> next(s)
0.259885230571928


>>> next(s)
0.794228474665430 8
```

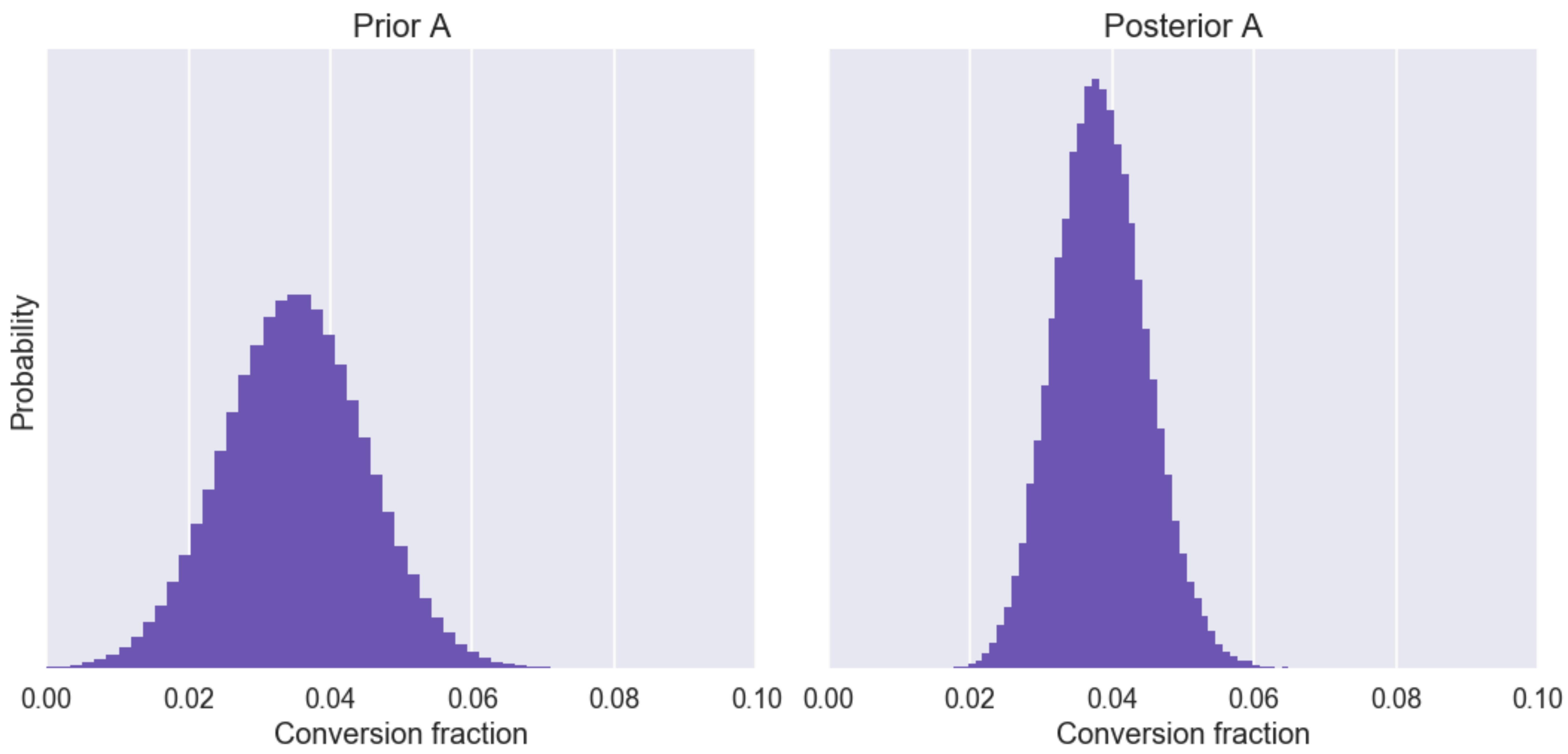
```
def simulate_conversion(p):
    """Returns number of visitors who convert given conversion fraction p."""
    conversions = 0
    for i in range(N):
        if random() < p:
            conversions += 1
    return conversions

>>> simulate_conversion(0.1)
44


>>> simulate_conversion(0.1)
52
```

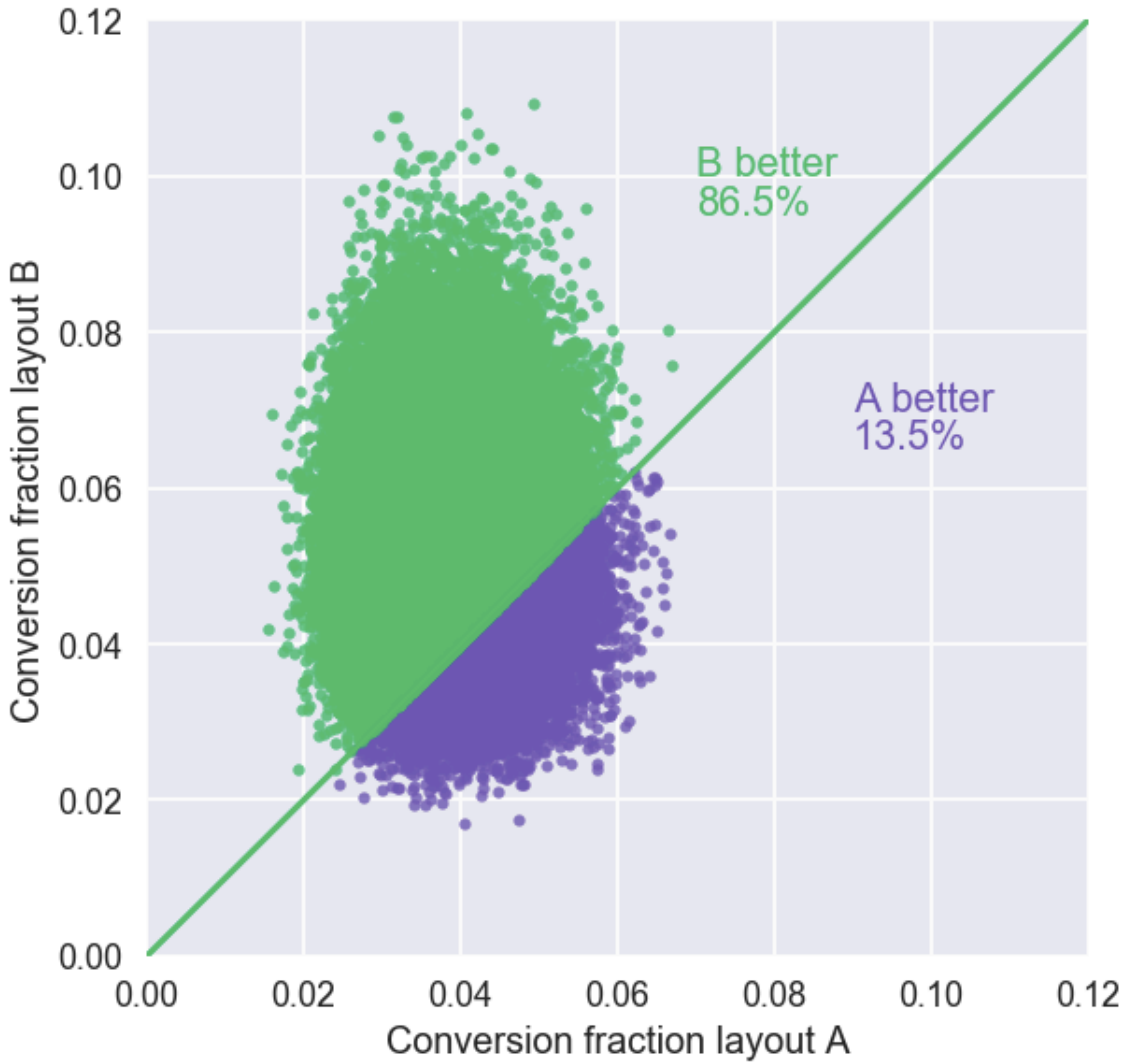
```
>>> posterior_sampler = abayes(n_converted,
                               uniform_prior_sampler(),
                               simulate_conversion)

>>> next(posterior_sampler)
0.04223951410146609

>>> next(posterior_sampler)
0.06332386076583127
```



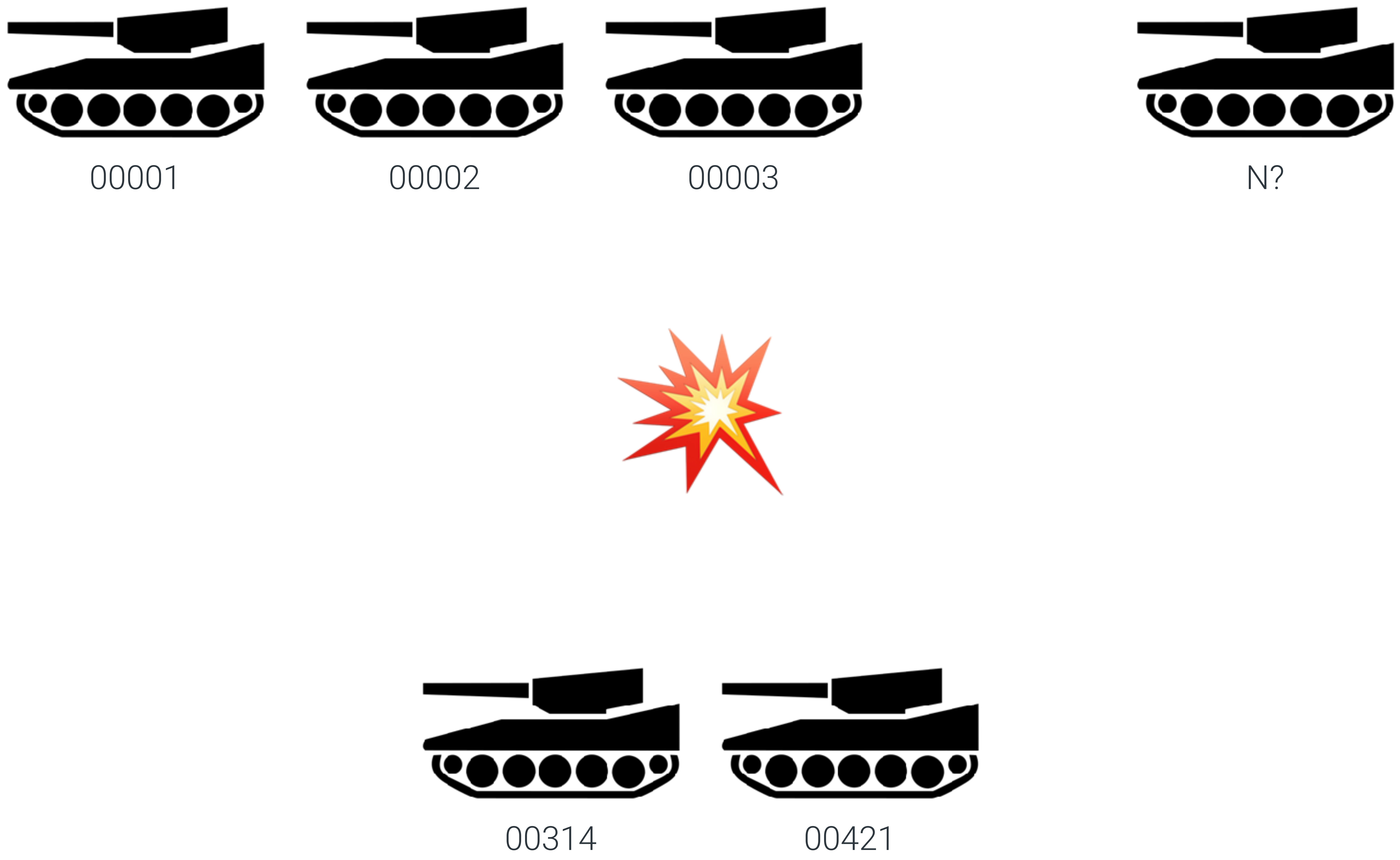
Probability
0.00
0.02
0.04
0.06
0.08
0.10
Conversion fraction
Probability
0.00
0.02
0.04
0.06
0.08
0.10
Conversion fraction
Probability
0.00
0.02
0.04
0.06
0.08
0.10
Conversion fraction


Prior B
Posterior B
Probability
0.00
0.02
0.04
0.06
0.08
0.10
Conversion fraction
0.00
0.02
0.04
0.06
0.08
0.10
Conversion fraction


Prior A
Posterior A
Probability
0.00
0.02
0.04
0.06
0.08
0.10
Conversion fraction
Conversion fraction

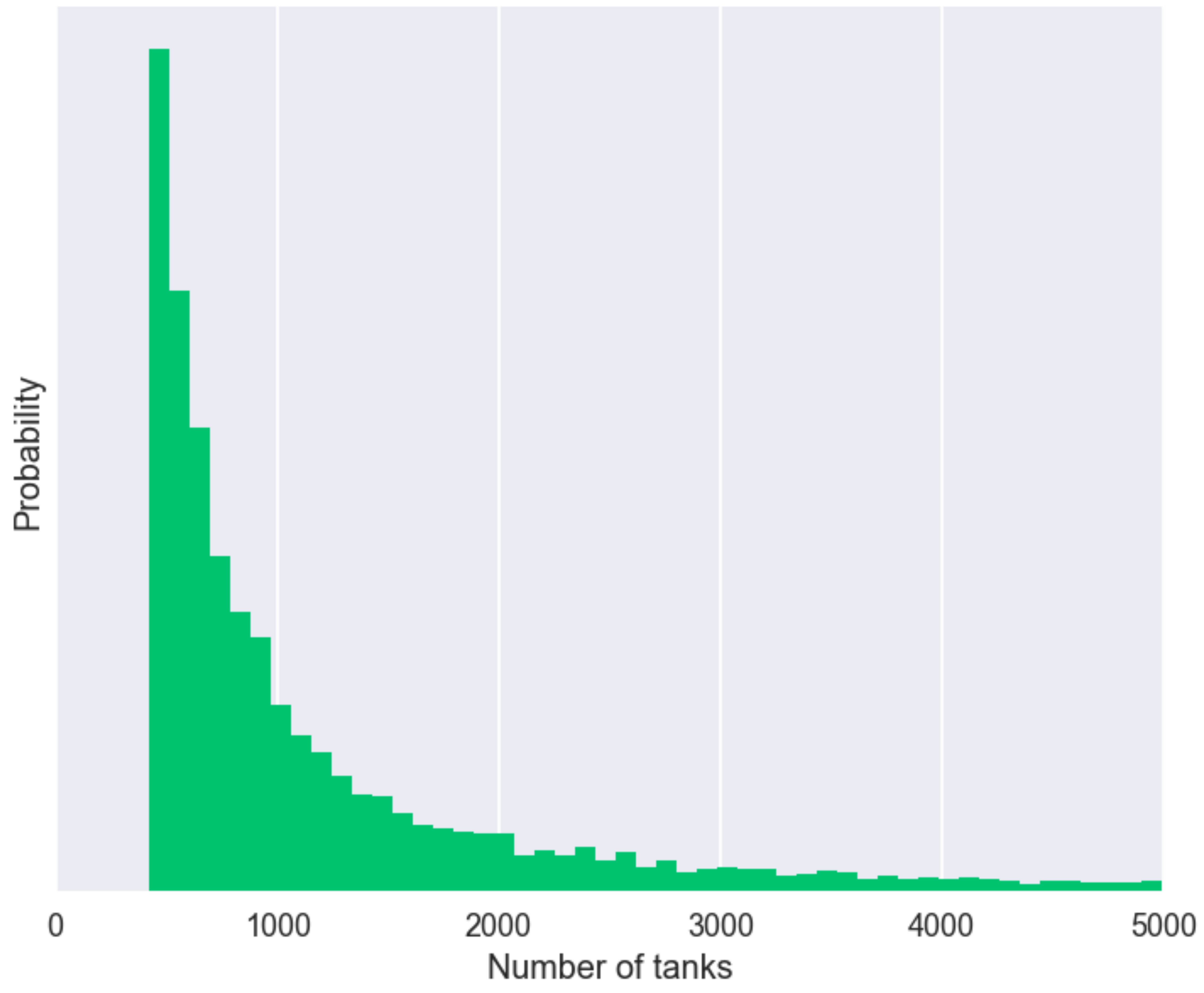


B better
86.5%
A better
13.5%
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Conversion fraction layout B
0.00
0.02
0.04
0.06
0.08
0.10
0.12
Conversion fraction layout A


00001
00002
00003
N?
00314
00421

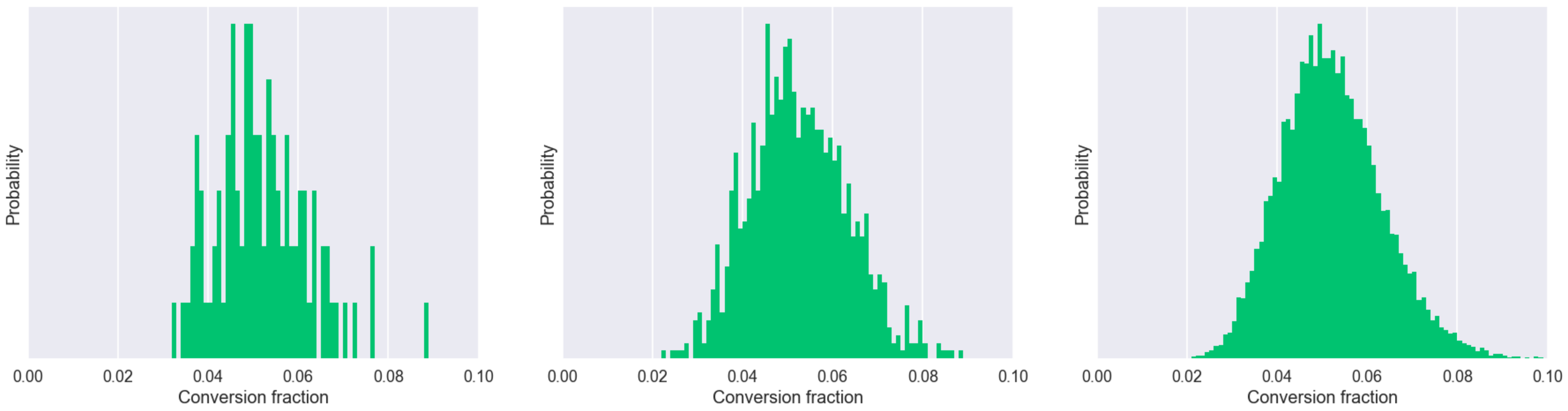
```python
def abayes(data, prior_sampler, simulate):
    """Yield samples from the posterior by Approximate Bayesian Computation."""

    # For each guess based on our prior beliefs
    for p in prior_sampler:

        # Simulate the experiment and see if it matches the real data
        if simulate(p) == data:

            # If it does, it was a good guess!
            yield p
```

```python
import random

data = [314, 421]

def tank_prior_sampler():
    while True:
        yield random.randint(421, 5000)

def simulate_tank_capture(N):
    return random.sample(range(N), 2)

tank_posterior_sampler = abayes(data,
                                tank_prior_sampler(),
                                simulate_tank_capture)
```



Probability
0
1000
2000
3000
4000
5000
Number of tanks

But Bayesian inference is slow if you’re not careful.



Probability
Conversion fraction
0.00
0.02
0.04
0.06
0.08
0.10
Probability
Conversion fraction
0.00
0.02
0.04
0.06
0.08
0.10
Probability
Conversion fraction
0.00
0.02
0.04
0.06
0.08
0.10

```
47%|                    | 47/100 [00:02<00:04, 25.84it/s]
```

```
>>> N *= 2                # 800 visitors
>>> n_converted *= 2  # 40 conversions
```

```
 47%|████████████████████▋                       | 47/100 [00:15<00:31,  6.62it/s]
```

```python
def abayes(data, prior_sampler, simulate):
    """Yield samples from the posterior by Approximate Bayesian Computation."""

    # For each guess based on our prior beliefs
    for p in prior_sampler:

        # Simulate the experiment and see if it matches the real data
        if simulate(p) == data:

            # If it does, it was a good guess!
            yield p
```

Being careful requires cleverness...

```python
import numpy as np
import itertools as it


def metropolis_hastings(dist, x0=0, burnin=1000, alpha=0.5, verbose=False):
    x = x0
    samples_accept = 0
    for i in it.count(1):
        candidate = np.random.normal(loc=x, scale=alpha)
        candidate_prob = min([1.0, dist(candidate) / dist(x)])
        accept = np.random.rand()
        if accept < candidate_prob:
            samples_accept += 1
            x = candidate
        if i > burnin:
            yield x, i, samples_accept
```

Hamiltonian Monte Carlo

- explores efficiently

with automatic differentiation

- differentiates automatically

and NUTS

- …and is idiot-proof 😎

Bayesian inference is great in theory...

- Quantify risk
- Insert institutional knowledge
- Online learning

And it's pretty easy to implement from scratch

But fast implementations require cleverness…

- Metropolis Hastings
- Hamiltonian Monte Carlo with automatic differentiation and NUTS

the cleverness is now ready to abstracted away 😎

Probabilistic programming in the real world

PyMC3

```
>>> from pymc3 import Model, DiscreteUniform, sample

>>> with Model():
        n_tanks = DiscreteUniform('n_tanks', lower=max(captured_tanks), upper=5000)
        obs = DiscreteUniform('obs', lower=0, upper=n_tanks, observed=captured_tanks)
        trace = sample(10000)

Assigned Metropolis to n_tanks
100%|██████████| 10000/10000 [00:02<00:00, 3998.03it/s]
```

PyMC3

```
data {
    int<lower=0> N;
    int<lower=0> N_features;
    matrix[N, N_features] X;
    int<lower=0,upper=1> repaid[N];
}
parameters {
    vector[N_features] p_coef;
}
model {
    vector[N] p;
    p_coef ~ cauchy(0, 2.5);
    p = logit(X * p_coef);
    repaid ~ bernoulli(p);
}
```


Loan Officer Simulator
Round: 1
Capital: $100,000
Loan Applications
Exp. Profit
% Repaid
Upside
Downside
$28,500 at 20%
#1455183
13k 77% 21k 23k
$22,500 at 20% for 3 years
#8650164
10k 79% 16k 18k
$22,500 at 20%
#6581413
10k 75% 16k 17k
$24,000 at 20%
#450871
9k 75% 17k 21k
$25,500 at 20%
#457500
8k 69% 19k 22k
$22,500 at 15% for 3 years
#1524226
6k 75% +12k -19k
$16,500 at 20%
#3384062
6k 69% 12k 13k
Loan Actions
Adjust the interest rate below to see how it effects loan repayment probabilities
Adjust Rate:
22.5%
Set New Rate
Current Rate:
15.0%
Approve Loan
Probability Metrics
Int. Rate | Expected Profit | % Repaid | Upside | Downside
15.0% | $5,600 | 75% | +$11,700 | -$18,800
22.5% | $6,000 | 59% | +$18,900 | -$20,700
+7.5% | +$400 | -16% | +$7,100 | -$1,800
Probable Repayment Outcomes
80k 70k 60k 50k 40k 30k 20k 10k 0
Each gray bar in this graph represents a repayment amount with a 1% probability
Partial Repayment Distribution
Chance of partial repayment:
24.7%
40.8%
4% 3% 2% 1% 0%
0.0 5.0k 10k 15k 20k 25k 30k 35k 40k 45k 50k


Probabilistic Real Estate
?
Where can I buy a property for
$3,000,000
NEIGHBORHOOD
HEX
Hex 15221
DUMBO-Vinegar Hill-Downtown Brooklyn-Boerum Hill, Brooklyn
You would have a high (74%) probability of being able to afford a property for $3.0M in 2018.
PROBABILITY DISTRIBUTION FOR 2018
40%
30%
20%
10%
0%
$0
$1M
$2M
$3M
$4M
$5M
$6M
$7M
$8M
PROBABILITY BY YEAR
100%
80%
60%
40%
20%
0%
2010
2011
2012
2013
2014
2015
2016
2017
2018
MODE:
Probability
Price
+
−
Battery Park City-Lower Manhattan
Brooklyn Heights-Cobble Hill
DUMBO-Vinegar Hill-Downtown Brooklyn-Boerum Hill
Fort Greene
Clinton Hill
Williamsburg
E BROADWAY
MADISON ST
SOUTH ST
WATER ST
MAIDEN LN
WILLIAMSBURG BRIDGE
BROOKLYN BRIDGE
YORK ST
PROSPECT ST
JAY ST
TILLARY ST
FLUSHING AVE
PARK AVE
MYRTLE AVE
DEKALB AVE
FULTON ST
LAFAYETTE AVE
ATLANTIC AVE
SCHERMERHORN ST
FURMAN ST
HICKS ST
COURT ST
UNION ST
VAN BRUNT ST
COLUMBIA ST
BROOKLYN-QUEENS EXPY
HUGH L. CAREY TUNNEL
VANDERBILT AVE
WASHINGTON AVE
CLASSON AVE
KENT AVE
BEDFORD AVE
BROADWAY

Options

Stan

- great for offline analysis 👍
- but it's awkward to productize

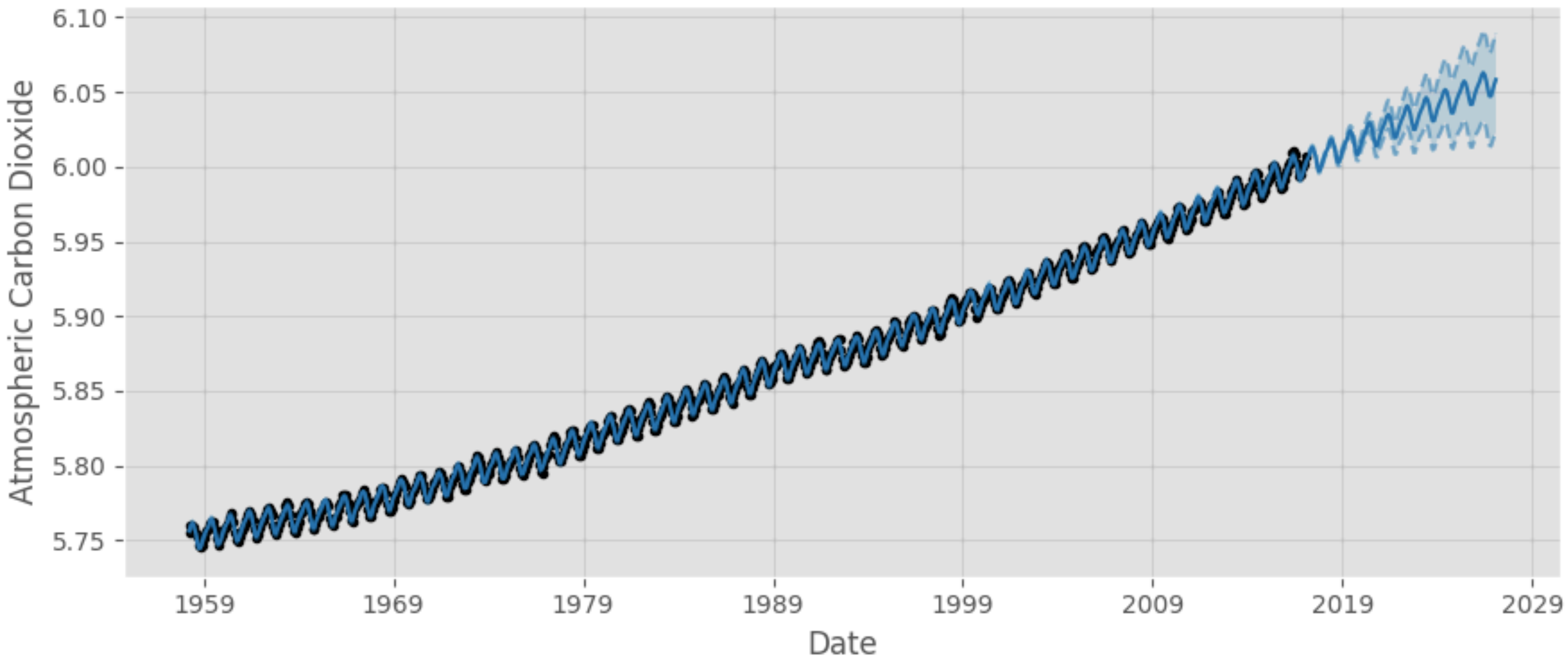
pymc3

- algorithmically half a step behind (much less true than it used to be)
- much easier to build products with 😎
- pymc4 on Tensorflow Probability coming soon

Others

- Tensorflow Probability, Edward, Anglican, Figaro, Pyro

Prophet (Facebook)



Next steps

The algorithms behind probabilistic programming
http://blog.fastforwardlabs.com/2017/01/30/the-algorithms-behind-probabilistic-programming.html

NYC Real Estate Simulator
http://fastforwardlabs.github.io/pre/

Probabilistic programming from scratch
https://www.oreilly.com/learning/probabilistic-programming-from-scratch

Or get in touch! @mikepqr or mlw@cloudera.com