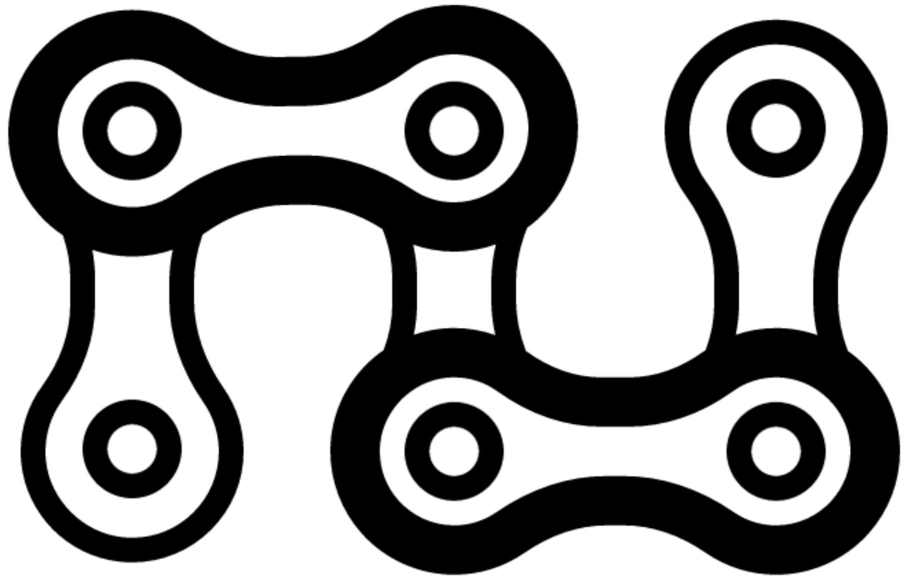
SaaaS: Sampling as an Alg Service

From for-loop to PyMC3



Vincent D. Warmerdam

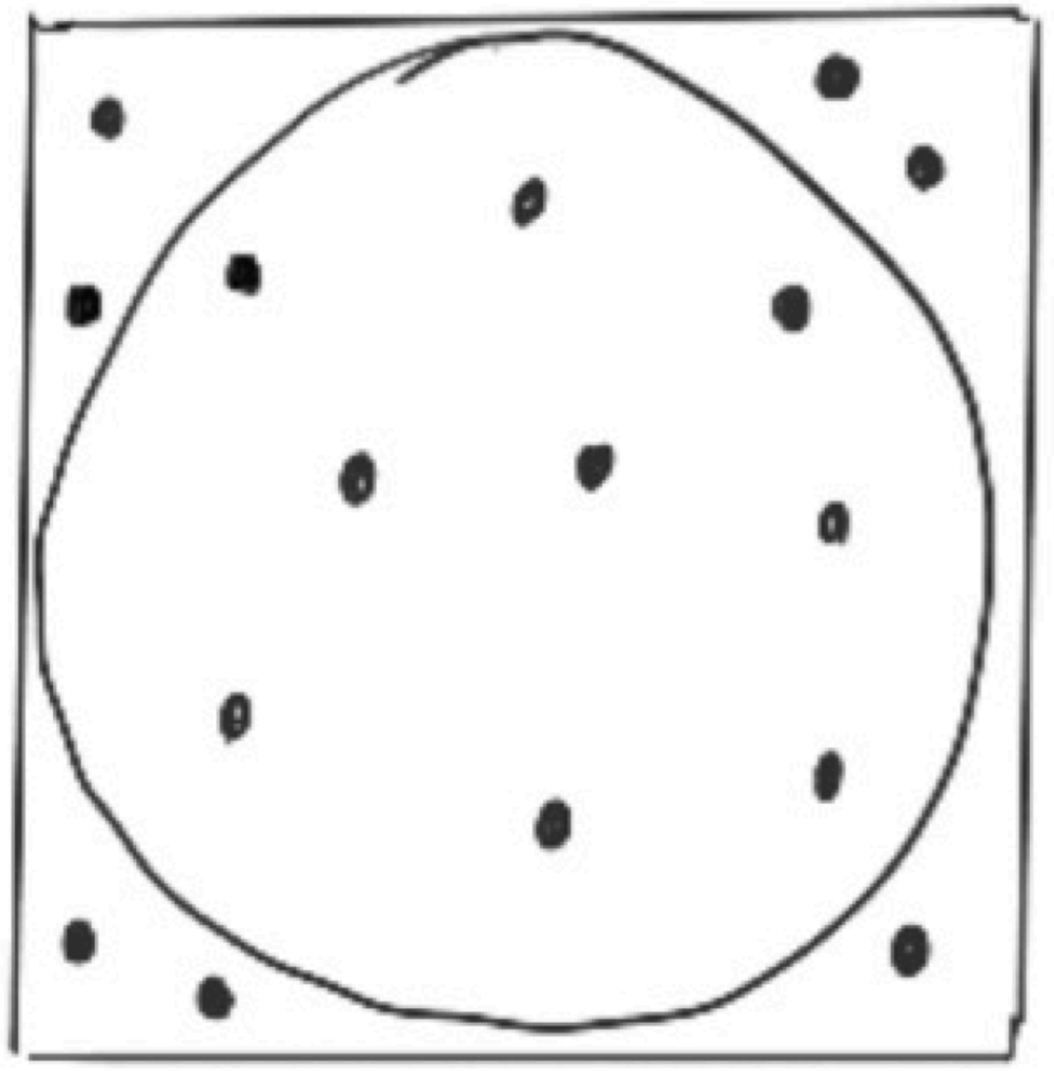
Today

- explain sampling methods as a concept
- demonstrate minimum viable forloop
- translate idea to machine learning algorithm
- apply idea to kaggle sub-problem
- apply idea to timeseries task
- build a competing algorithm in tensorflow

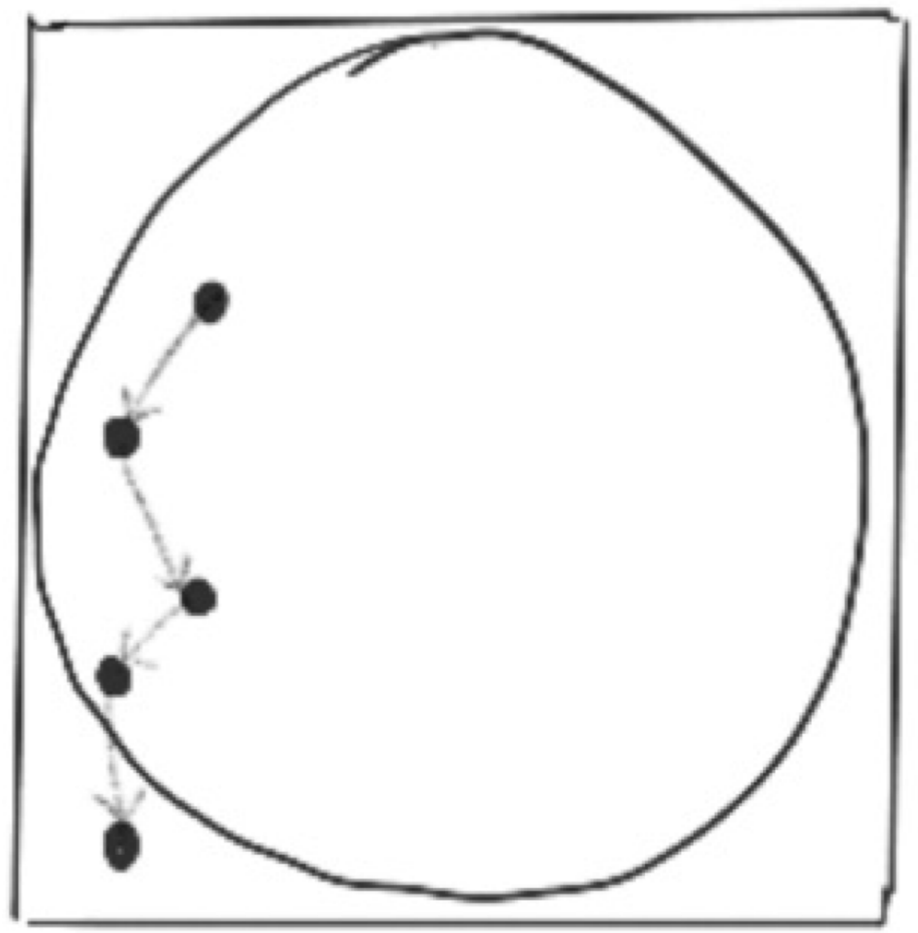
Sampling

The simplest example ...

Idea of Sampling: π



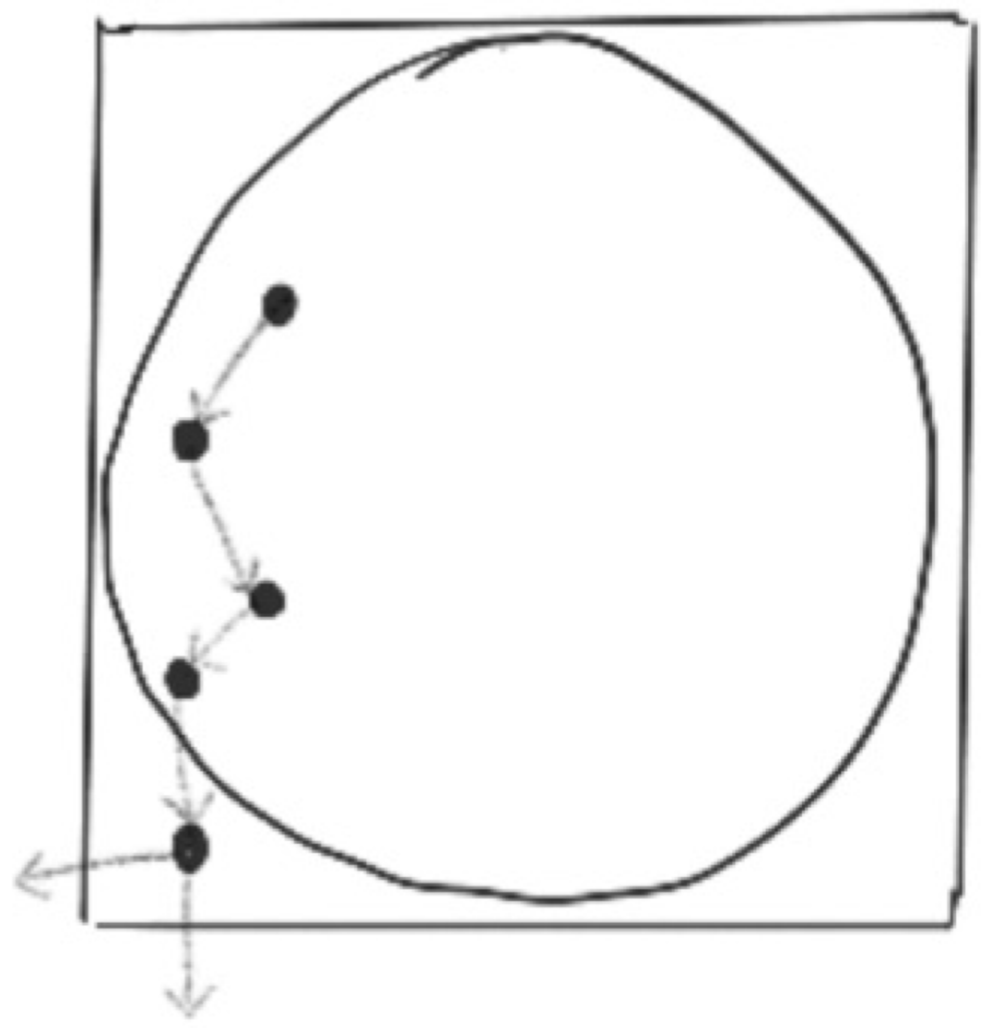
Alternative Sampling: π



Alternative Sampling: π

"Seems logical enough"
What might go wrong here?

Alternative Sampling: π



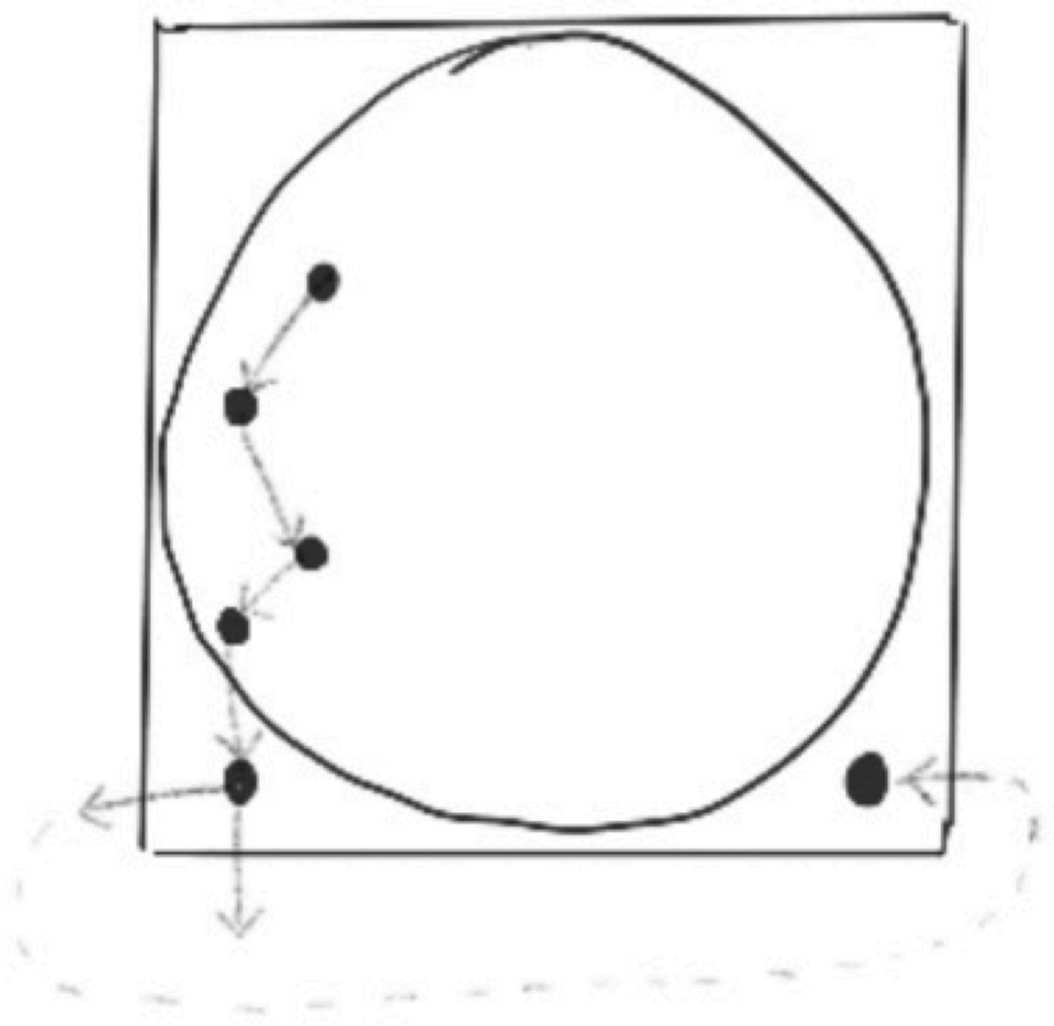
Alternative Sampling: π

But what about those boundaries?

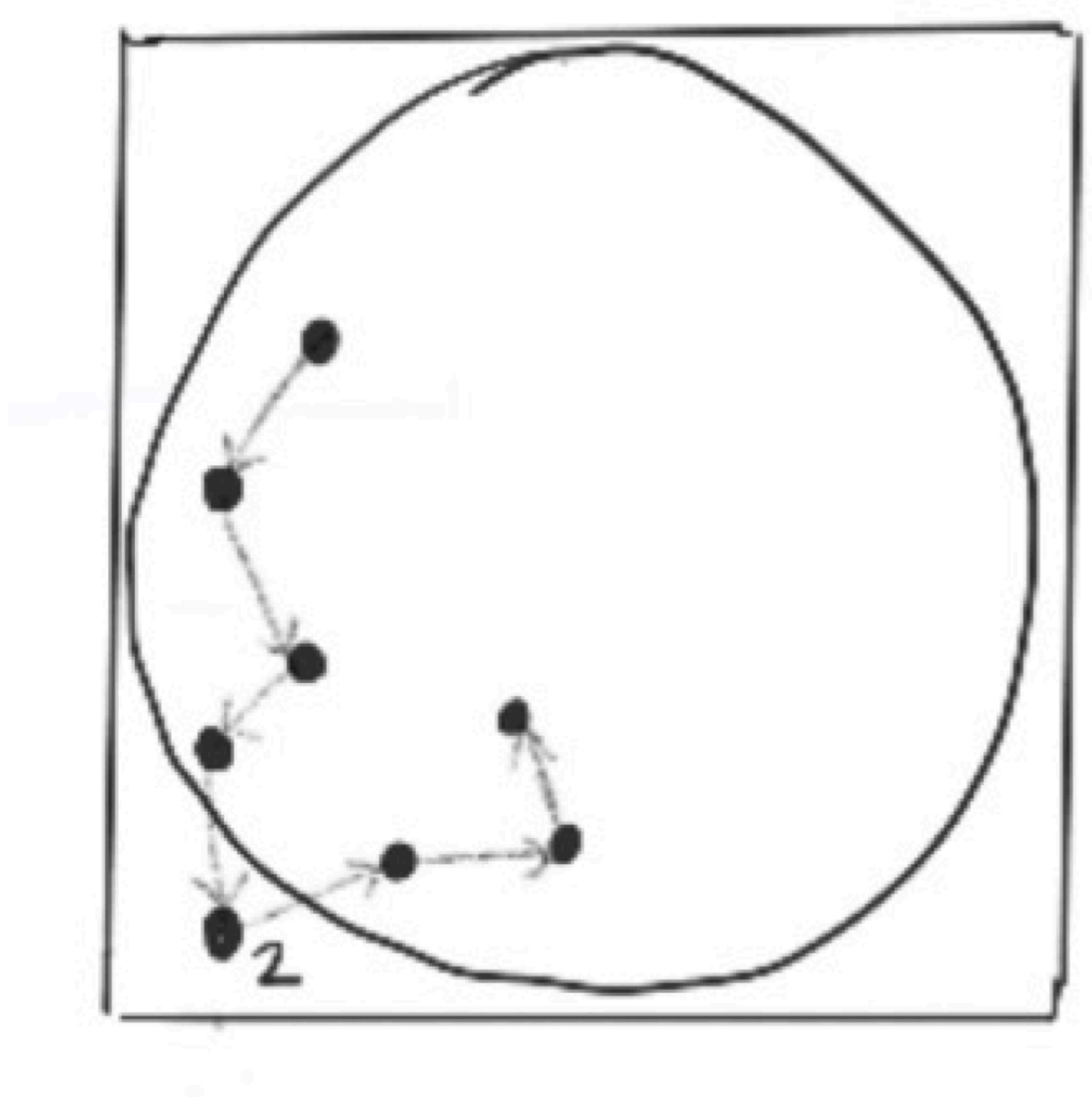
We only want to explore the region of interest, we cannot have our poor pebble go to infinity. We may need to twiddle a bit of the markov stepping rule.

We could try out two approaches.

Alternative 1: π



Alternative 2: π



Idea of Sampling: π

We can use direct sampling or indirect sampling to estimate π.

Idea of Sampling: π

We can use direct sampling or indirect sampling to estimate π.

This indirect sampling won't be as random initially, but after a (long) while we will reach a very similar estimate of π using either of the sampling methods.

Idea of Sampling: π

We can use direct sampling or indirect sampling to estimate π.

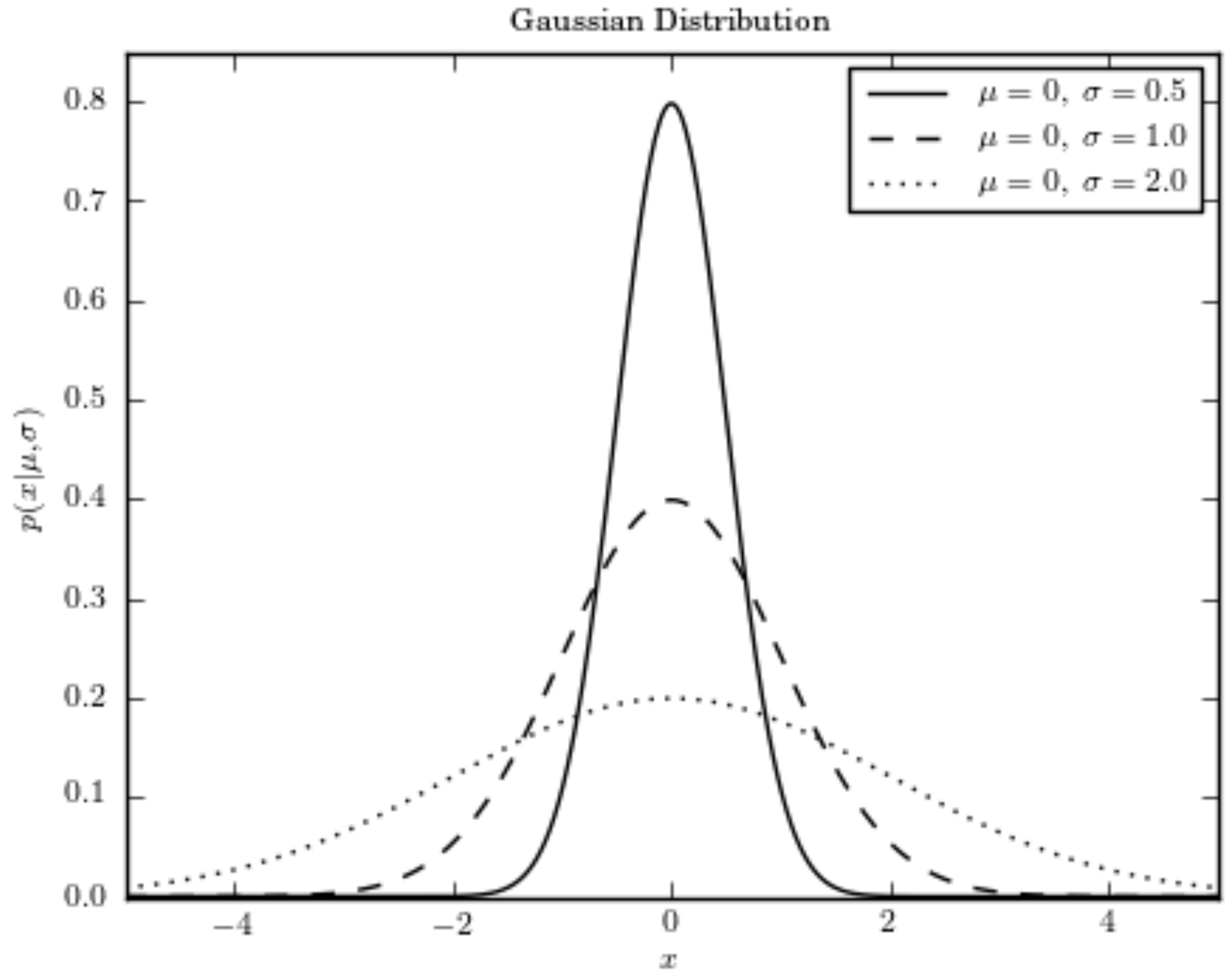
This indirect sampling won't be as random initially, but after a (long) while we will reach a very similar estimate of π using either of the sampling methods.

We have just seen a markov chain being used to infer a value of interest. Let's see if we can use this idea to perhaps also sample other values.

Idea of Sampling: μ, σ.

Now suppose that I have some data points $D = \{1, 3, 4, 5, 6\}$. I'm assuming that this data comes from a normal distribution and I'd like to discover what normal distribution the data might have came from.

Remember. In normal distributions μ and σ have different shapes.



Idea of Sampling: μ, σ.

Now suppose that I have some data points $D = \{1, 3, 4, 5, 6\}$. I'm assuming that this data comes from a normal distribution and I'd like to discover what normal distribution the data might have came from.

I accept the data to be the only truth I have and I accept that I'll be a bit uncertain about my estimate of μ and σ.

Idea of Sampling: μ, σ.

Now suppose that I have some data points $D = \{1, 3, 4, 5, 6\}$. I'm assuming that this data comes from a normal distribution and I'd like to discover what normal distribution the data might have came from.

I accept the data to be the only truth I have and I accept that I'll be a bit uncertain about my estimate of μ and σ.

I wouldn't mind being able to quantify this uncertainty by the way.

Idea of Sampling: μ, σ.

Let's pick a random value of μ and σ and let's start to do a markov chain around the parameter space. Let's do this lots and lots of times (k times let's say).

$$(\mu_1, \sigma_1) \rightarrow (\mu_2, \sigma_2) \rightarrow (\mu_3, \sigma_3) \rightarrow \ldots \rightarrow (\mu_k, \sigma_k),$$

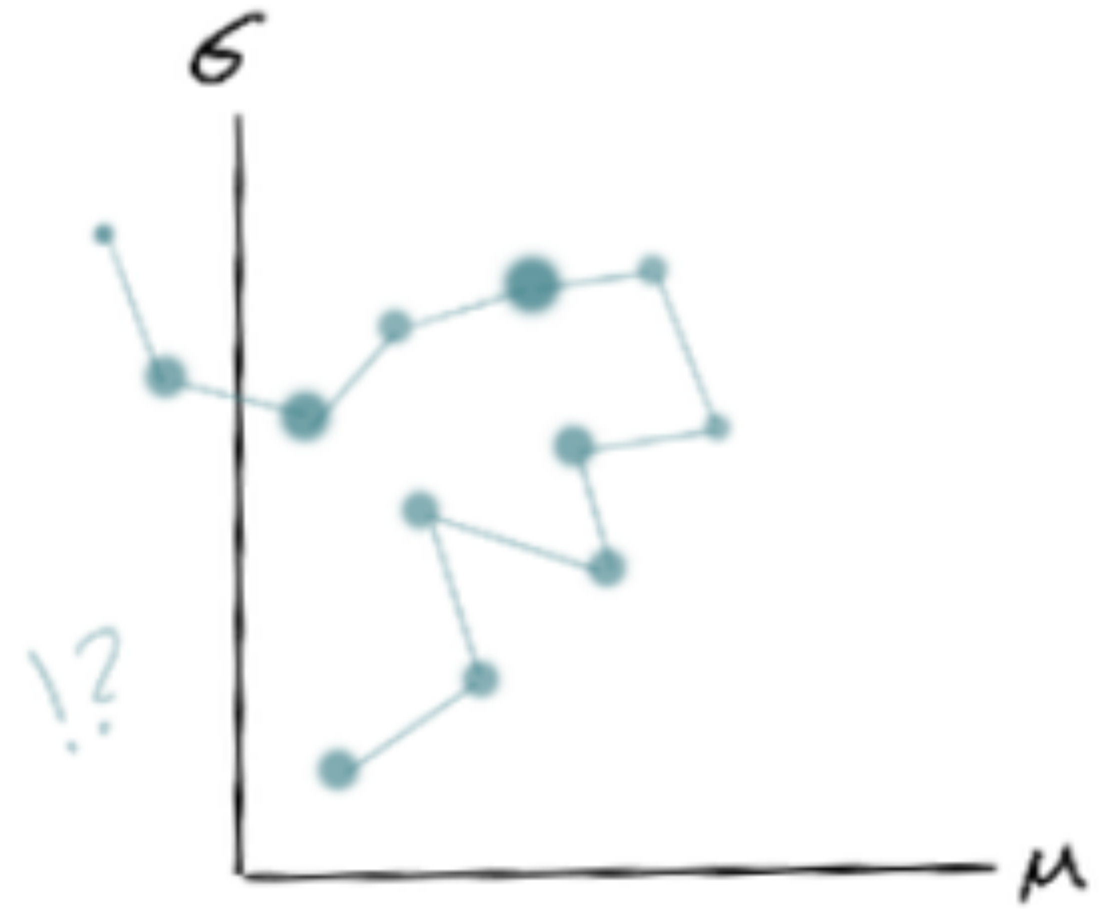
Idea of Sampling: μ, σ.

Let's pick a random value of μ and σ and let's start to do a markov chain around the parameter space. Let's do this lots and lots of times (k times let's say).

$$(\mu_1, \sigma_1) \rightarrow (\mu_2, \sigma_2) \rightarrow (\mu_3, \sigma_3) \rightarrow \ldots \rightarrow (\mu_k, \sigma_k),$$

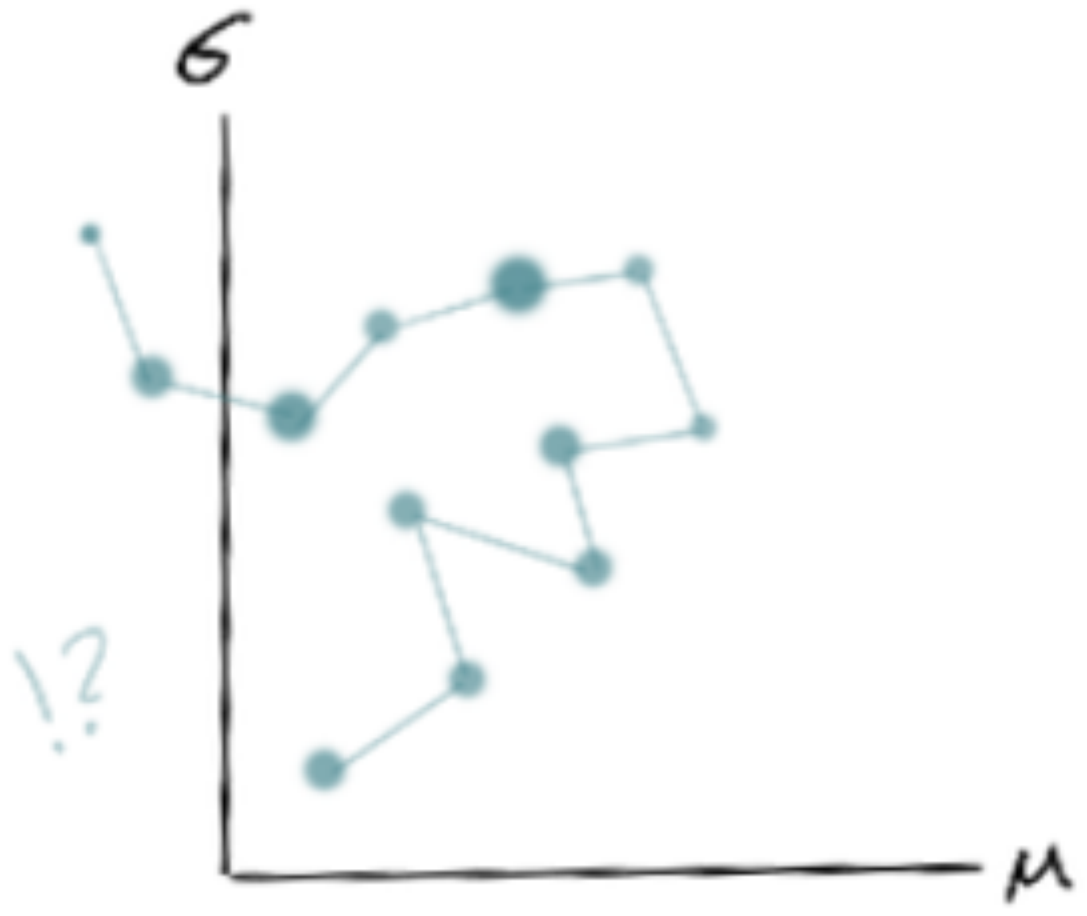
What might go wrong here?

Idea of Sampling: μ, σ.



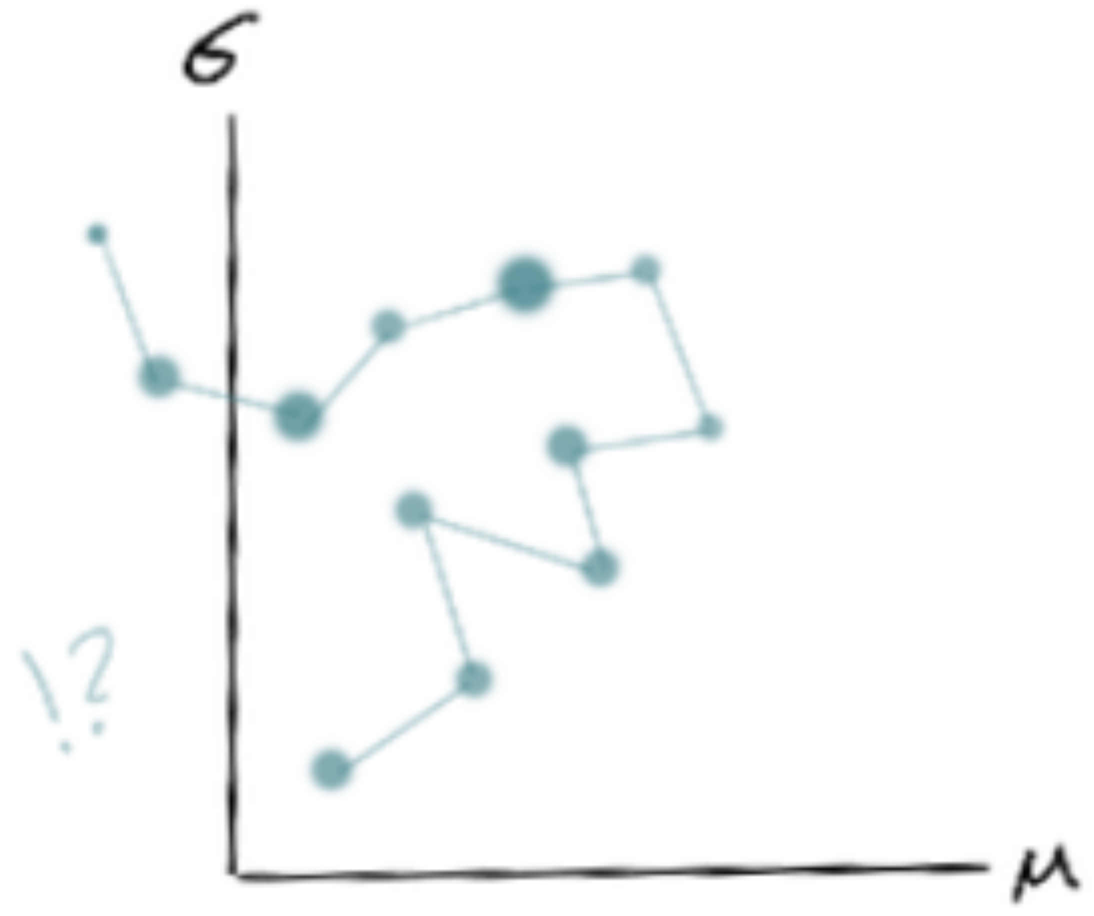
We might explore areas we do not care for.

Idea of Sampling: μ, σ.



We need to think of a way to prevent this.

Idea of Sampling: μ, σ.



How do we stay clear of areas that are less likely?

Measure Likelihood

We need a way to score our $(\mu_{k+1}, \sigma_{k+1})$ samples. How about ...

... if the next parameter $(\mu_{k+1}, \sigma_{k+1})$ we sample is likely, we should move our pebble.

Measure Likelihood

We need a way to score our $(\mu_{k+1}, \sigma_{k+1})$ samples. How about ...

... if the next parameter $(\mu_{k+1}, \sigma_{k+1})$ we sample is likely, we should move our pebble.

... if the next parameter $(\mu_{k+1}, \sigma_{k+1})$ we sample is *not* likely, we should perhaps stand still for a turn and try again.

Measure Likelihood

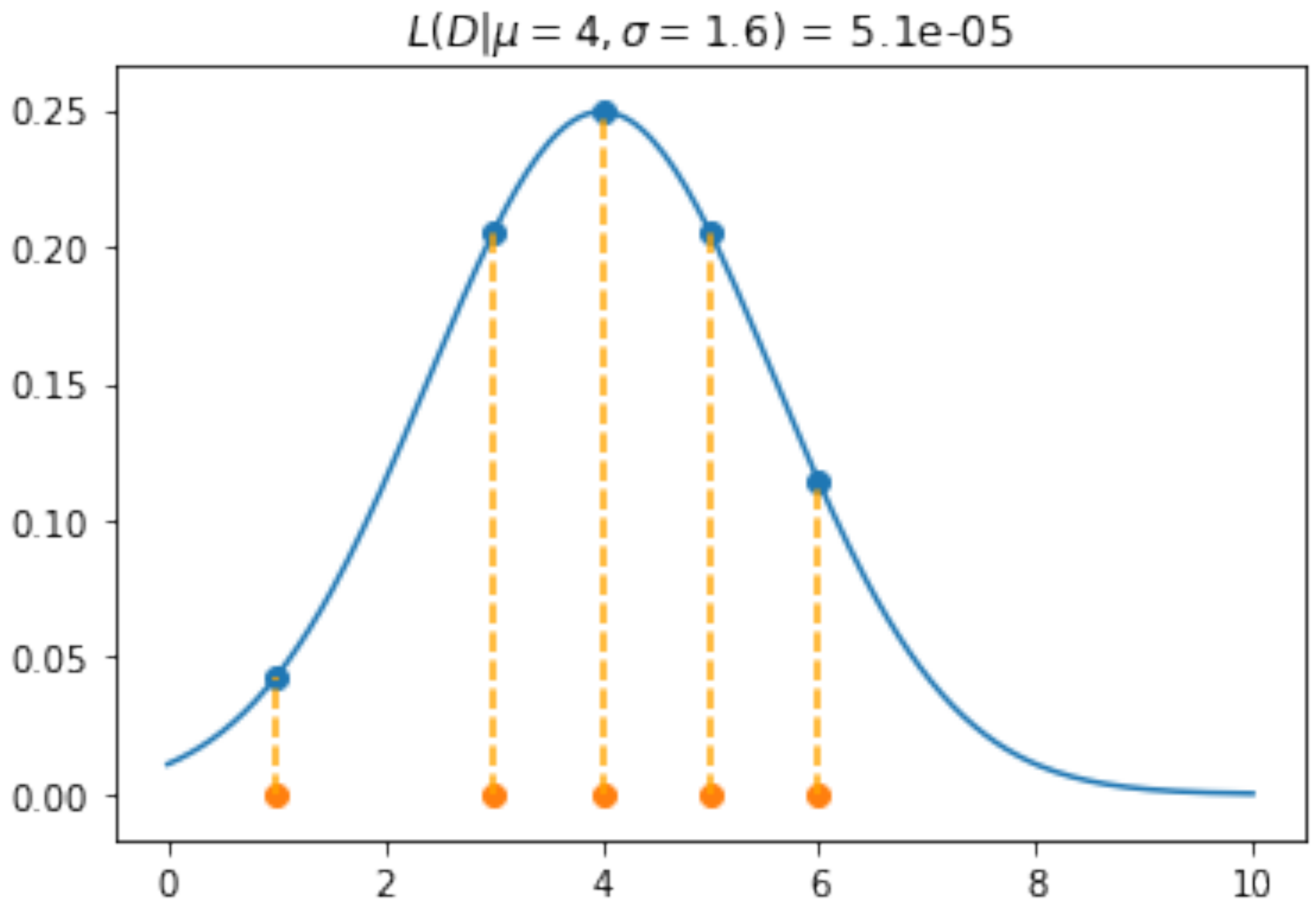
We need a way to score our $(\mu_{k+1}, \sigma_{k+1})$ samples. How about ...

... if the next parameter $(\mu_{k+1}, \sigma_{k+1})$ we sample is likely, we should move our pebble.

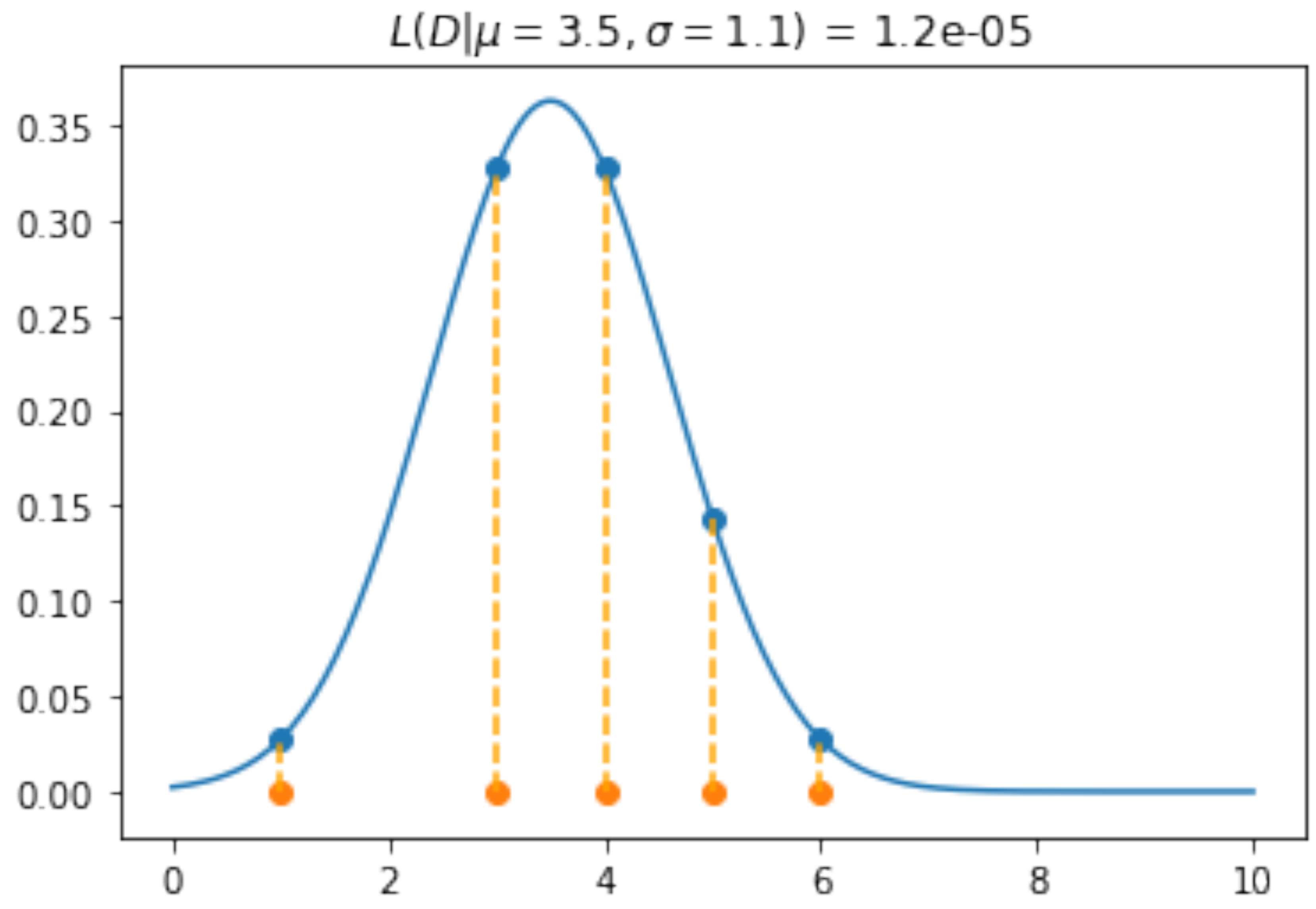
... if the next parameter $(\mu_{k+1}, \sigma_{k+1})$ we sample is *not* likely, we should perhaps stand still for a turn and try again.

How should we measure likelihood?

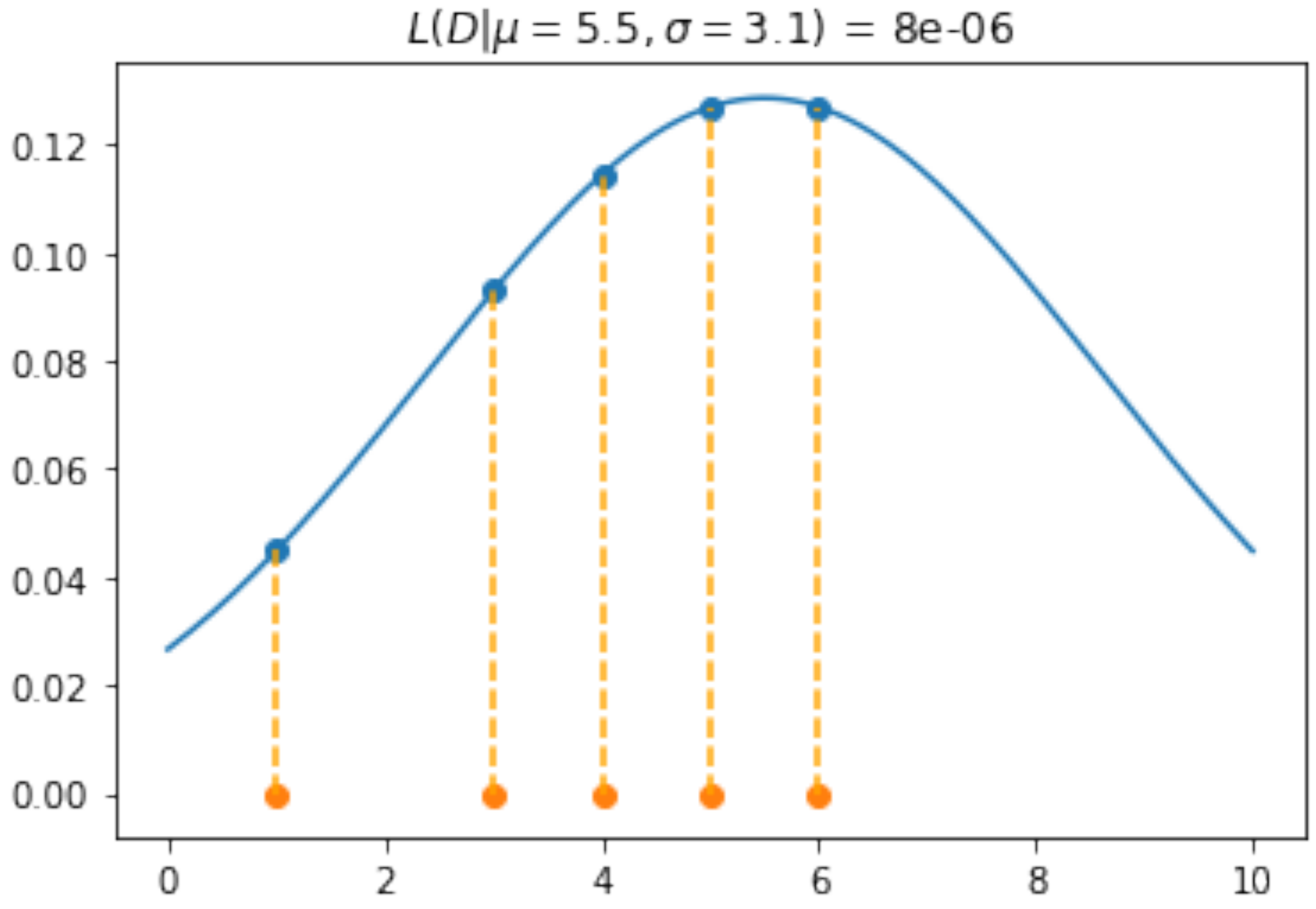
Use a Normal Distribution?



Different Parameters have Different Scores



Different Parameters have Different Scores



Likelihood

We can markov sample in search space, we can calculate a score but we still need a method of dealing with the score.

Likelihood

Let's consider an example for intuition.

1. $(\mu_A, \mu_A) \rightarrow$ score = 0.6
2. $(\mu_B, \mu_B) \rightarrow$ score = 0.2

Likelihood

Let's consider an example for intuition.

1. $(\mu_A, \mu_A) \rightarrow$ score = 0.6
2. $(\mu_B, \mu_B) \rightarrow$ score = 0.2

We need to find a way to make the step decision proportional to the likelihood score.

Likelihood

Let's consider an example for intuition.

1. $(\mu_A, \mu_A) \rightarrow$ score = 0.6
2. $(\mu_B, \mu_B) \rightarrow$ score = 0.2

How about;

$$p(\text{pick } B) = \frac{0.2}{0.6}$$

Recap

I just discussed a lot, let's recap what we have.

Recipe

We just discussed all the components we need.

1. We have data D and parameters μ, σ we want to estimate.

Recipe

We just discussed all the components we need.

1. We have data D and parameters μ, σ we want to estimate.

2. We are going to sample around the $[\mu, \sigma]$-space, markov style.

Recipe

We just discussed all the components we need.

1. We have data D and parameters μ, σ we want to estimate.
2. We are going to sample around the $[\mu, \sigma]$-space, markov style.
   2a. If the next μ_{k+1}, σ_{k+1} is more likely, move.
   2b. If not, apply a coin flip and decide to stay or move on.

Recipe

We just discussed all the components we need.

1. We have data D and parameters μ, σ we want to estimate.

2. We are going to sample around the $[\mu, \sigma]$-space, markov style.
   2a. If the next μ_{k+1}, σ_{k+1} is more likely, move.
   2b. If not, apply a coin flip and decide to stay or move on.

3. Repeat a whole lotta times.

Let's code this.

First we need to set stuff up.

```
import numpy as np
from scipy import stats

nums = np.random.normal(loc=1, scale=0.5, size=10)
mu = [0]
sigma = [3]
stepsize = 0.1

def calc_lik(data, mu, sigma):
    return np.prod(stats.norm.pdf(data, loc=mu, scale=sigma))
```

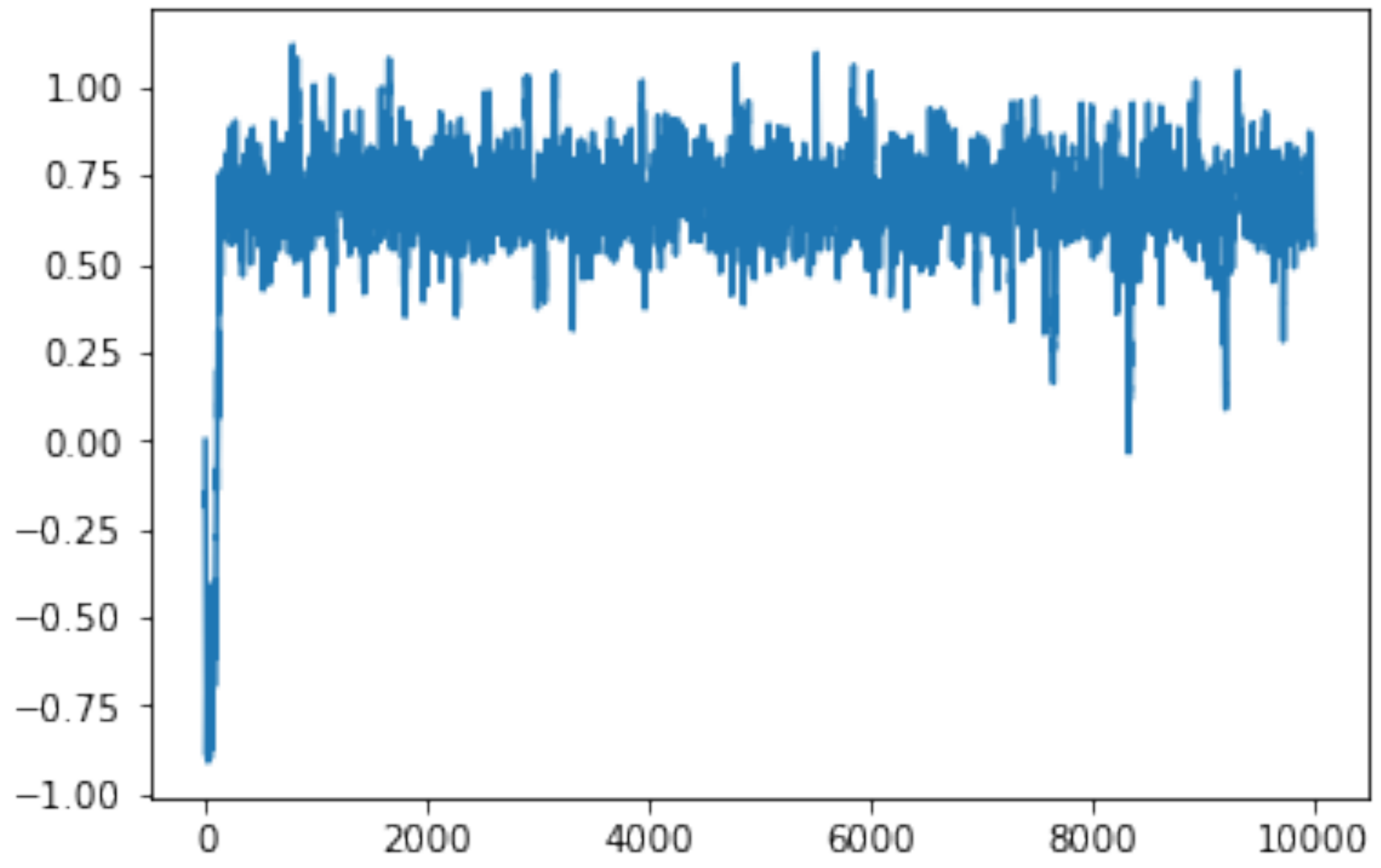
Let's code this

Next we can just sample!

```
for i in range(10000):
    new_mu = mu[-1] + np.random.normal(0, stepsize, 1)[0]
    new_sigma = sigma[-1] + np.random.normal(0, stepsize, 1)[0]
    old_loglik = calc_lik(nums, mu[-1], sigma[-1])
    new_loglik = calc_lik(nums, new_mu, new_sigma)
    if np.random.random() < new_loglik/old_loglik:
        mu.append(new_mu)
        sigma.append(new_sigma)
    else:
        mu.append(mu[-1])
        sigma.append(sigma[-1])
```

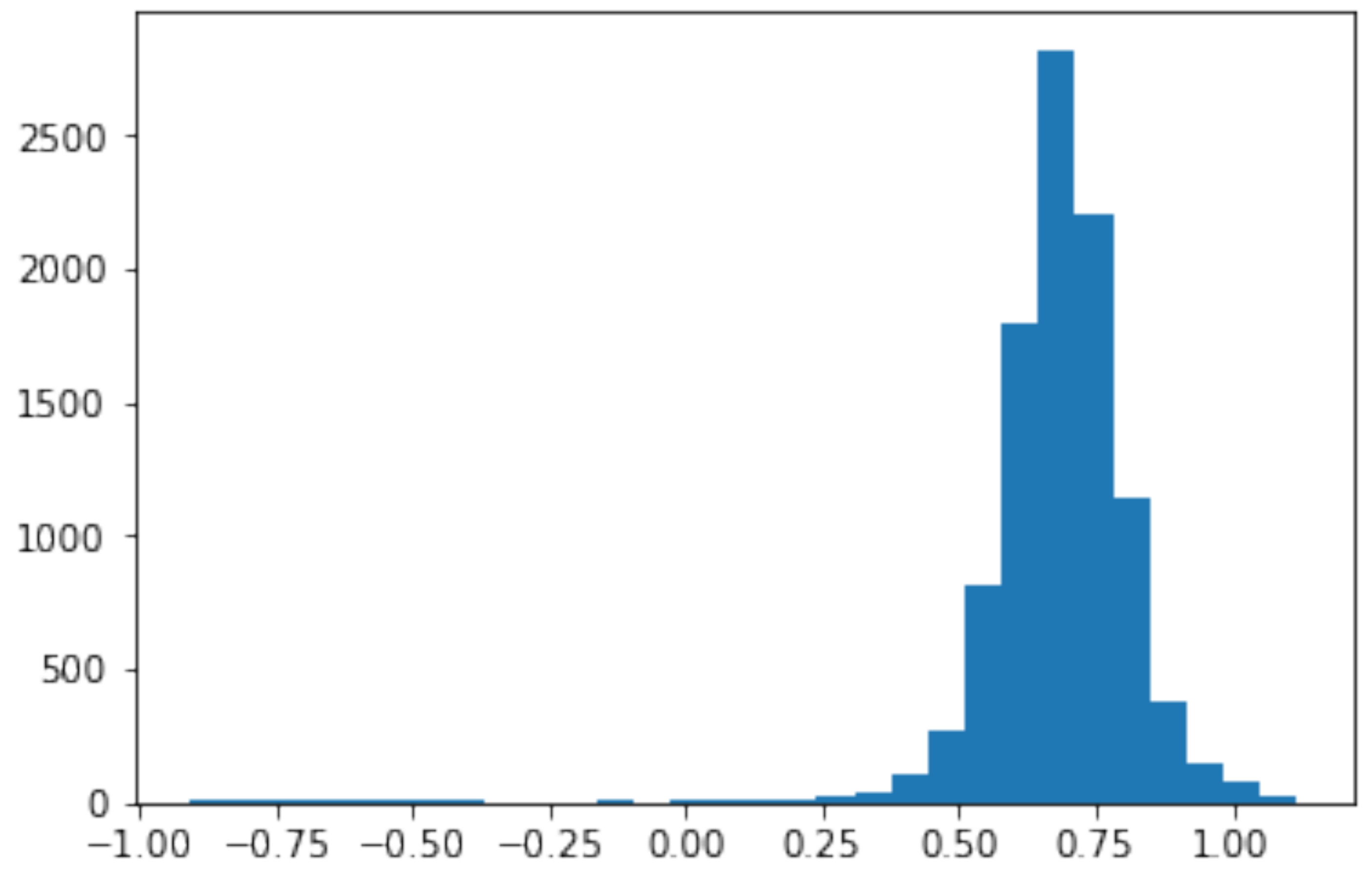
sampling results for μ

This is the value for μ over the iterations.



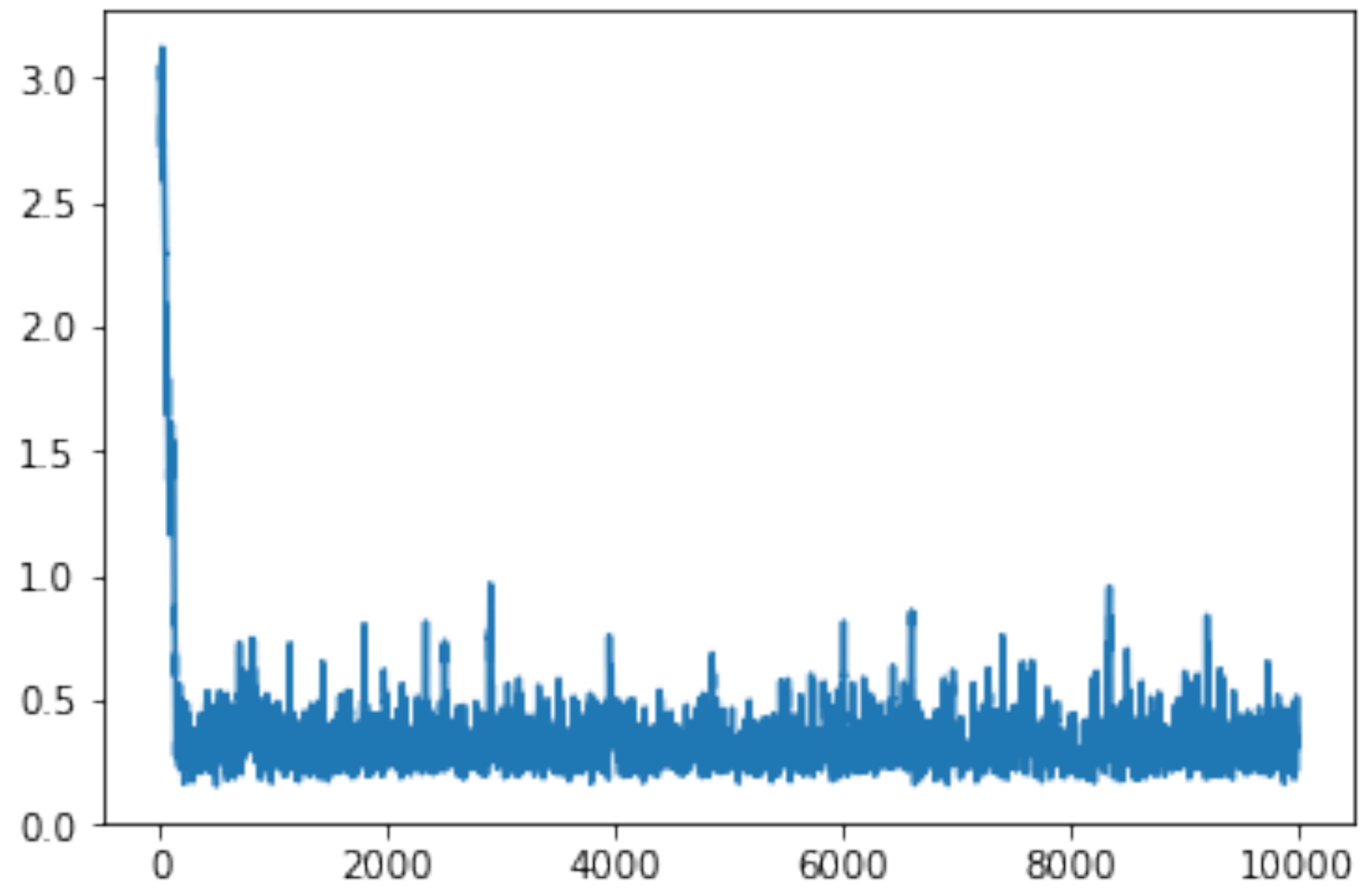
sampling results for μ

We can turn this series into a histogram.



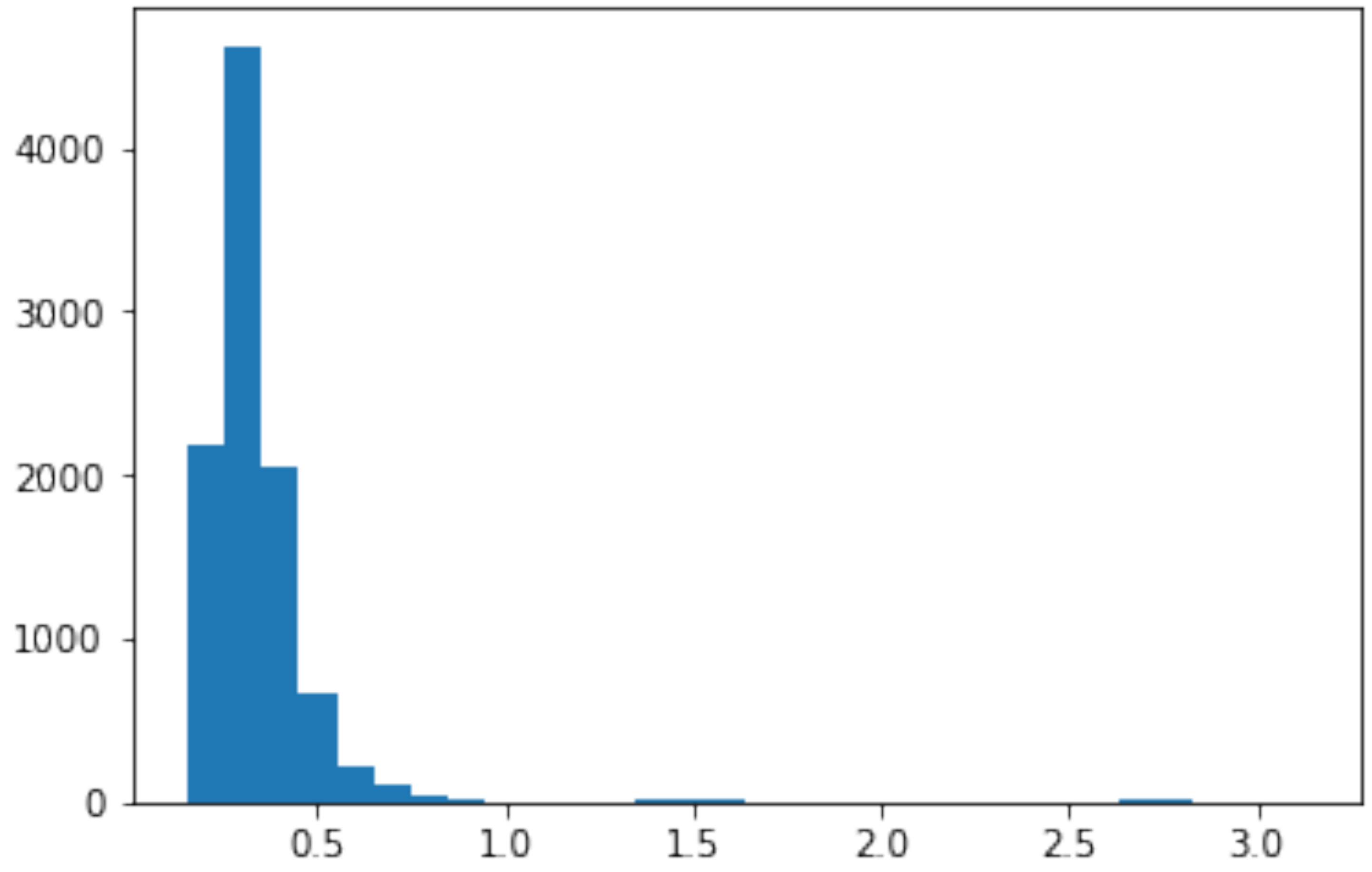
sampling results for σ

This is the value for σ over the iterations.



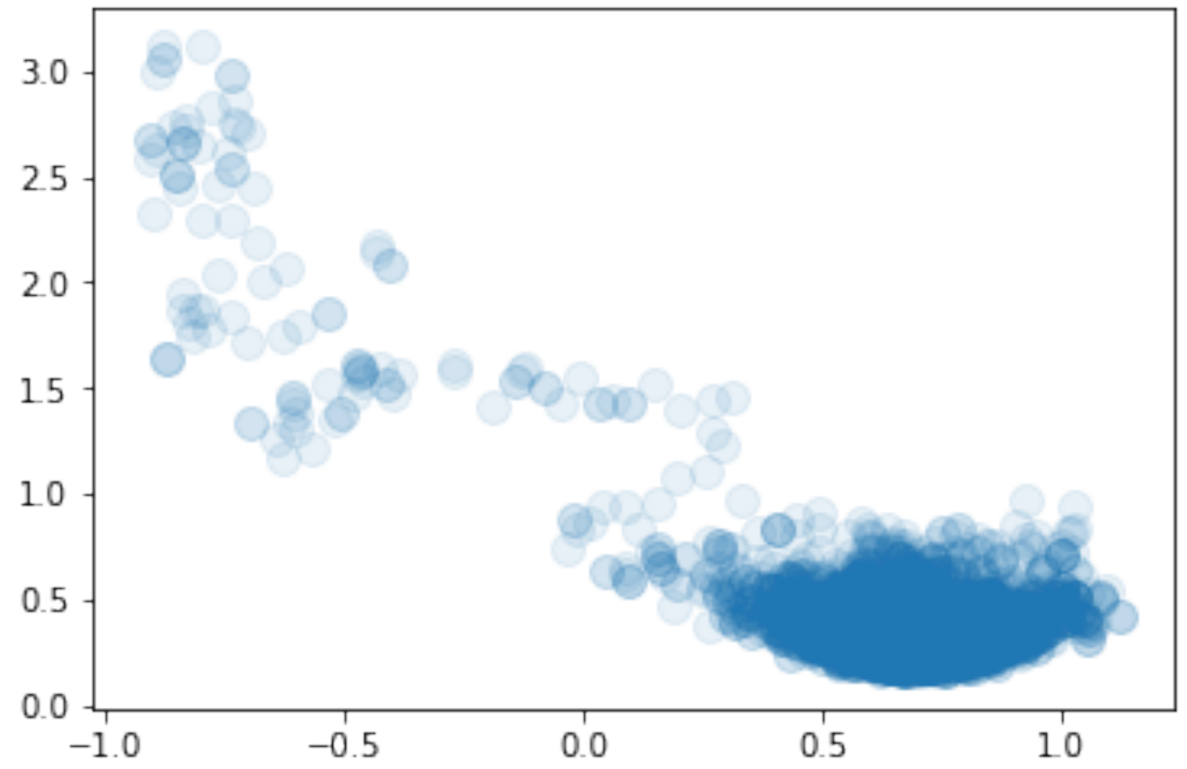
sampling results for σ

This too can be turned into a histogram.



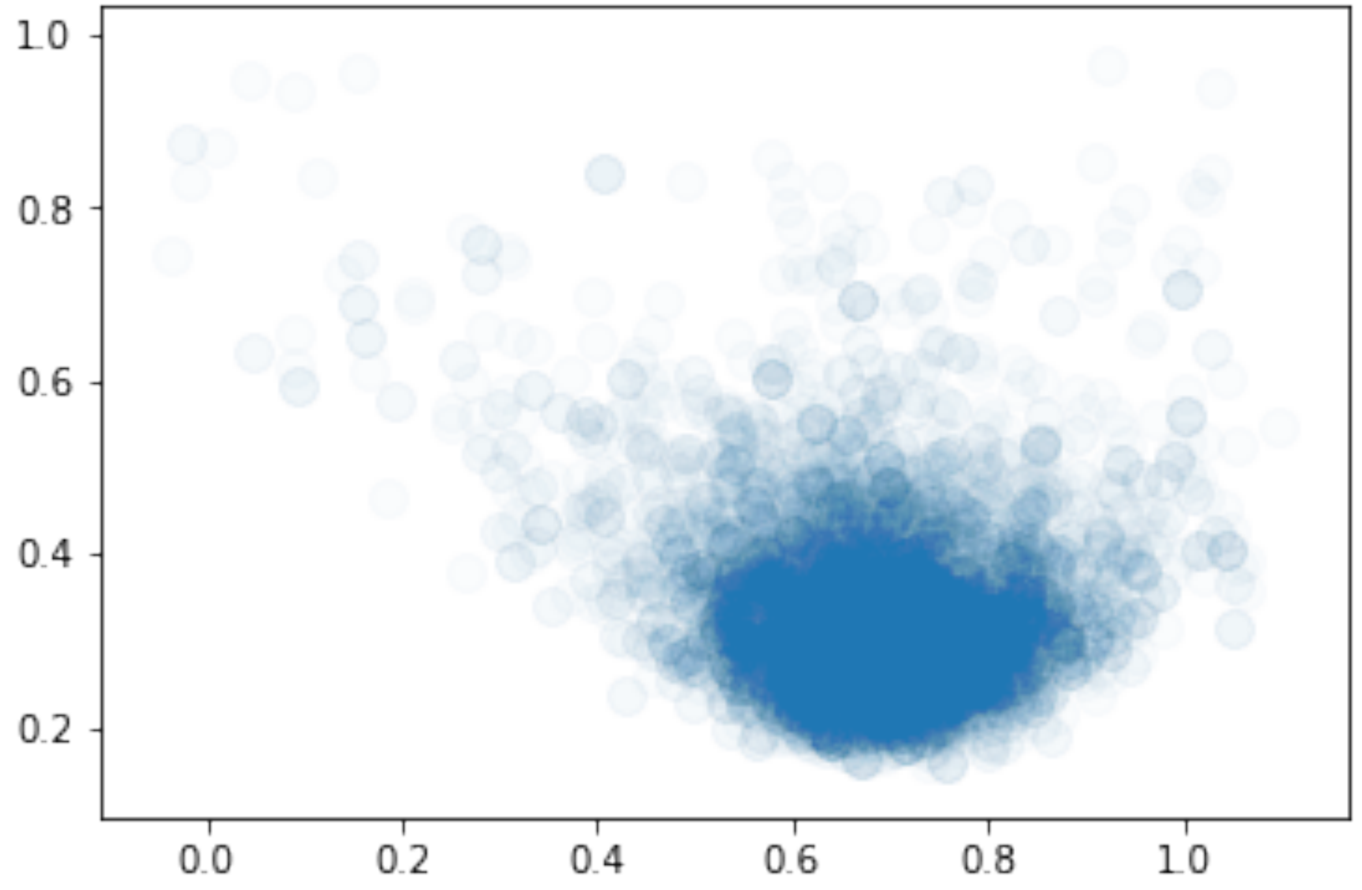
sampling results for μ, σ

The history confirms that initially we were in a region we quickly left for a more likely region.



sampling results for μ, σ

If we remove the first 1000 samples, it looks better.



a few lessons sofar

We've just demonstrated the idea behind sampling algorithms.

- our implementation is probably not very fast
- it may be better to use a specialized package
- we need to talk about more practical examples

a point in between

These are some personal tips.

- whenever you do not understand an algorithm, build it yourself [make it as simple as possible]
- try to write code that you love re-using, simple helper functions really clean up the code
- if the algorithm is new, debugging is hard. prevent debugging by testing in your notebook with a cell that contains some `assert` statements

Just a little maths.

Why does this algorithm matter to bayesians?

Just a little maths.

Bayesians care about one thing;

$$p(\theta|D) \propto p(D|\theta)p(\theta)$$

In our example, this translates to;

$$p(\mu, \sigma|D) \propto p(D|\mu, \sigma)p(\mu, \sigma)$$

Just a little maths.

Bayesians care about one thing;

$$p(\theta|D) \propto p(D|\theta)p(\theta) = \Pi_i p(d_i|\theta)p(\theta)$$

Typically we can easily model $p(d_i|\theta)$. In our example we picked a normal distribution but we can go quite a bit further.

Just a little maths.

Bayesians care about one thing;

$$p(\theta|D) \propto p(D|\theta)p(\theta) = \Pi_i p(d_i|\theta)p(\theta)$$

The cool thing about sampling though is that we are **very** flexible in our choice of models. We only need to know relative likelihoods from our model $p(d_i|\theta)$ in order to uncover $p(\theta|D)$.

Let's demonstrate this with a silly example first.

One Kaggle Example

Kaggle: Santas Uncertain Bags

Santa has 1000 bags to fill to fill with 9 types of gifts. Due to regulations at the North Pole workshop, no bag can contain more than 50 pounds of gifts. If a bag is overweight, it is confiscated by regulators from the North Pole Department of Labor without warning! Even Santa has to worry about throwing out his bad back.

We don't know the weight of the gifts but we want to send as much mass of presents as possible.

We were given priors!

The different gifts had different weights.

```
horse = max(0, np.random.normal(5,2,1)[0])
ball = max(0, 1 + np.random.normal(1,0.3,1)[0])
bike = max(0, np.random.normal(20,10,1)[0])
train = max(0, np.random.normal(10,5,1)[0])
coal = 47 * np.random.beta(0.5,0.5,1)[0]
book = np.random.chisquare(2,1)[0]
doll = np.random.gamma(5,1,1)[0]
block = np.random.triangular(5,10,20,1)[0]
```

So one could wonder, will it PyMC3?

Small example

We know the weight of certain bags but we don't know the weights of the individual gifts. Suppose that we have 4 gifts and that we've weighted 3 bags that contain them such that we have the following system of linear equations.

$$\begin{aligned} s_1 &= w_1 + w_2 \\ s_2 &= w_1 + w_2 + w_3 \\ s_3 &= w_3 + w_4 \end{aligned}$$

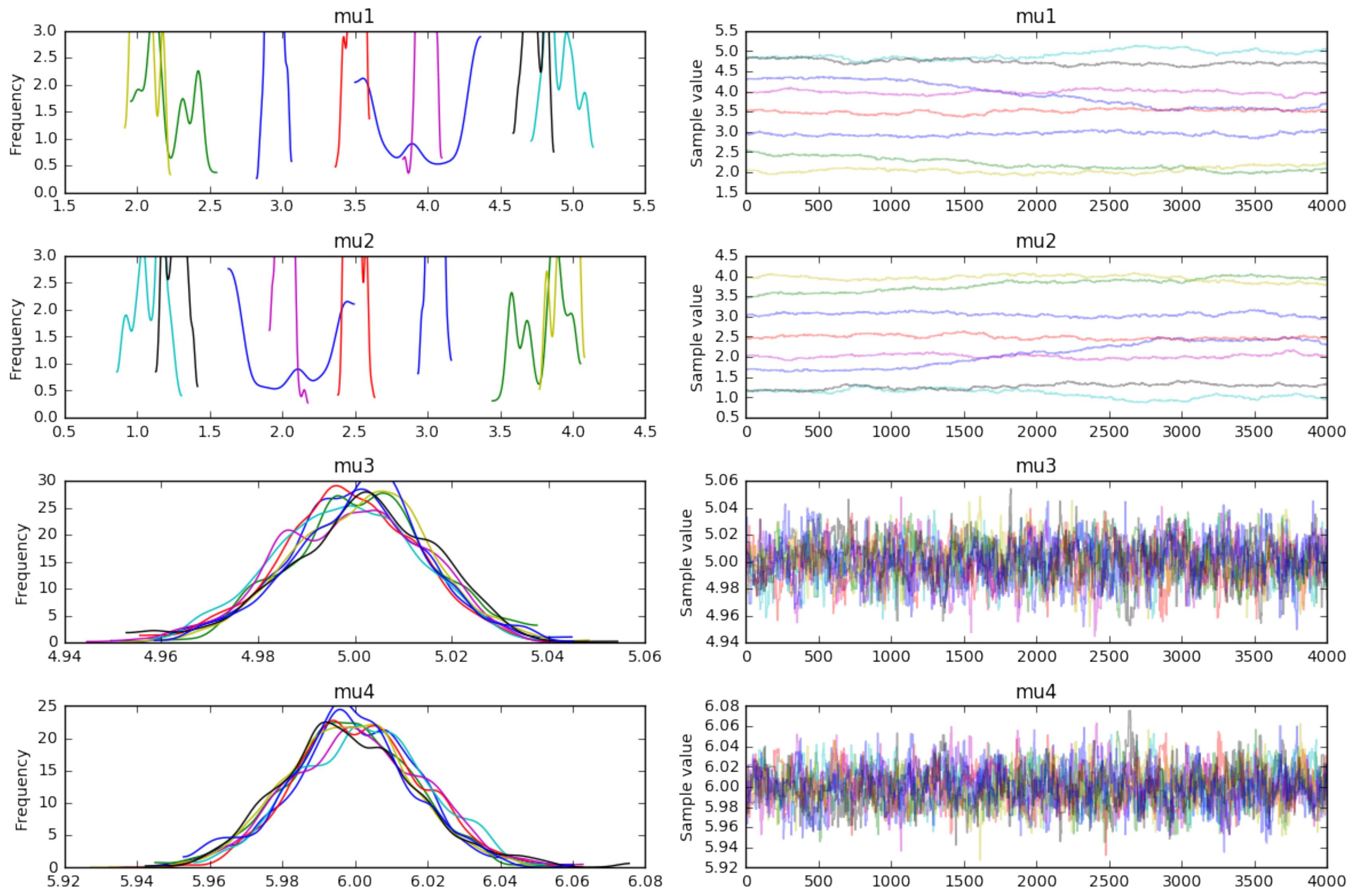
I know the values of s_* and I'd like to infer w_*.

Chug it in PyMC3

Simple code on simulated data.

```
w1, w2, w3, w4 = 2,4,5,6

with pm.Model() as basic_model:
    mu1 = pm.Normal('mu1', mu=5, sd=2)
    mu2 = pm.Normal('mu2', mu=5, sd=2)
    mu3 = pm.Normal('mu3', mu=5, sd=2)
    mu4 = pm.Normal('mu4', mu=5, sd=2)
    s1 = pm.Normal('s1', mu=mu1 + mu2, sd=0.01, observed=w1+w2)
    s2 = pm.Normal('s2', mu=mu1 + mu2 + mu3, sd=0.01, observed=w1+w2+w3)
    s3 = pm.Normal('s3', mu=mu3 + mu4, sd=0.01, observed=w3+w4)
```

The view



Given this system of equations;

$$
\begin{aligned}
s_1 &= w_1 + w_2 \\
s_2 &= w_1 + w_2 + w_3 \\
s_3 &= w_3 + w_4
\end{aligned}
$$

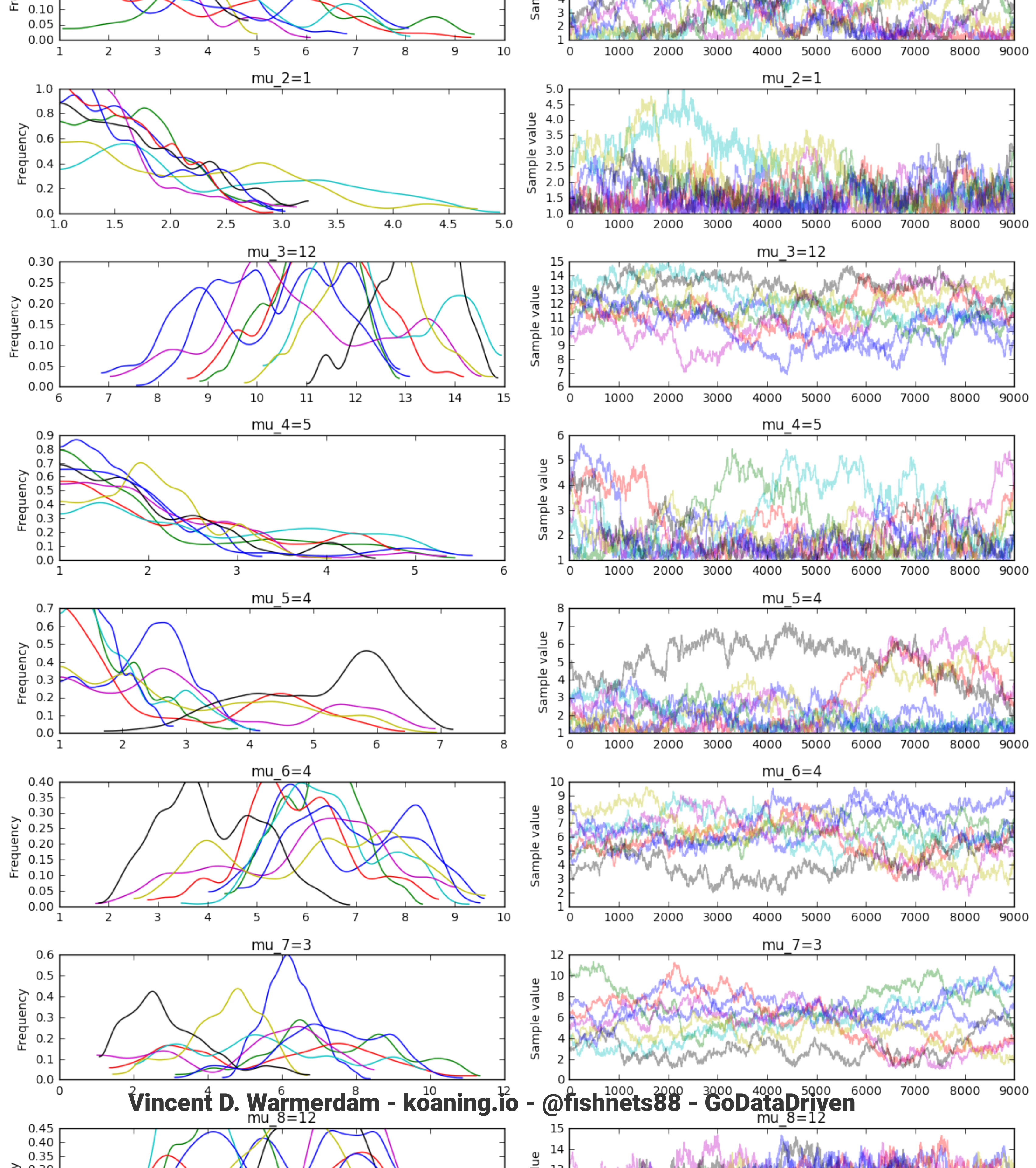
it makes sense that w_3 and w_4 converge. A nice thing about the sampler approach is that we do not focus on a maximum likelihood but on the entire likelihood space of the parameters of interest.

Moar?

I liked what I saw, but I wanted to know if it would scale.

$$
\begin{aligned}
w_3 + w_2 + w_4 + w_9 + w_5 + w_7 &= 27 \\
w_6 + w_7 + w_3 + w_4 + w_5 + w_8 &= 40 \\
w_0 + w_9 + w_4 + w_6 + w_3 + w_1 &= 36 \\
w_2 + w_7 + w_5 + w_6 + w_0 + w_4 &= 28 \\
w_1 + w_2 + w_6 + w_5 + w_8 + w_4 &= 28 \\
w_4 + w_0 + w_8 + w_3 + w_7 + w_6 &= 47
\end{aligned}
$$

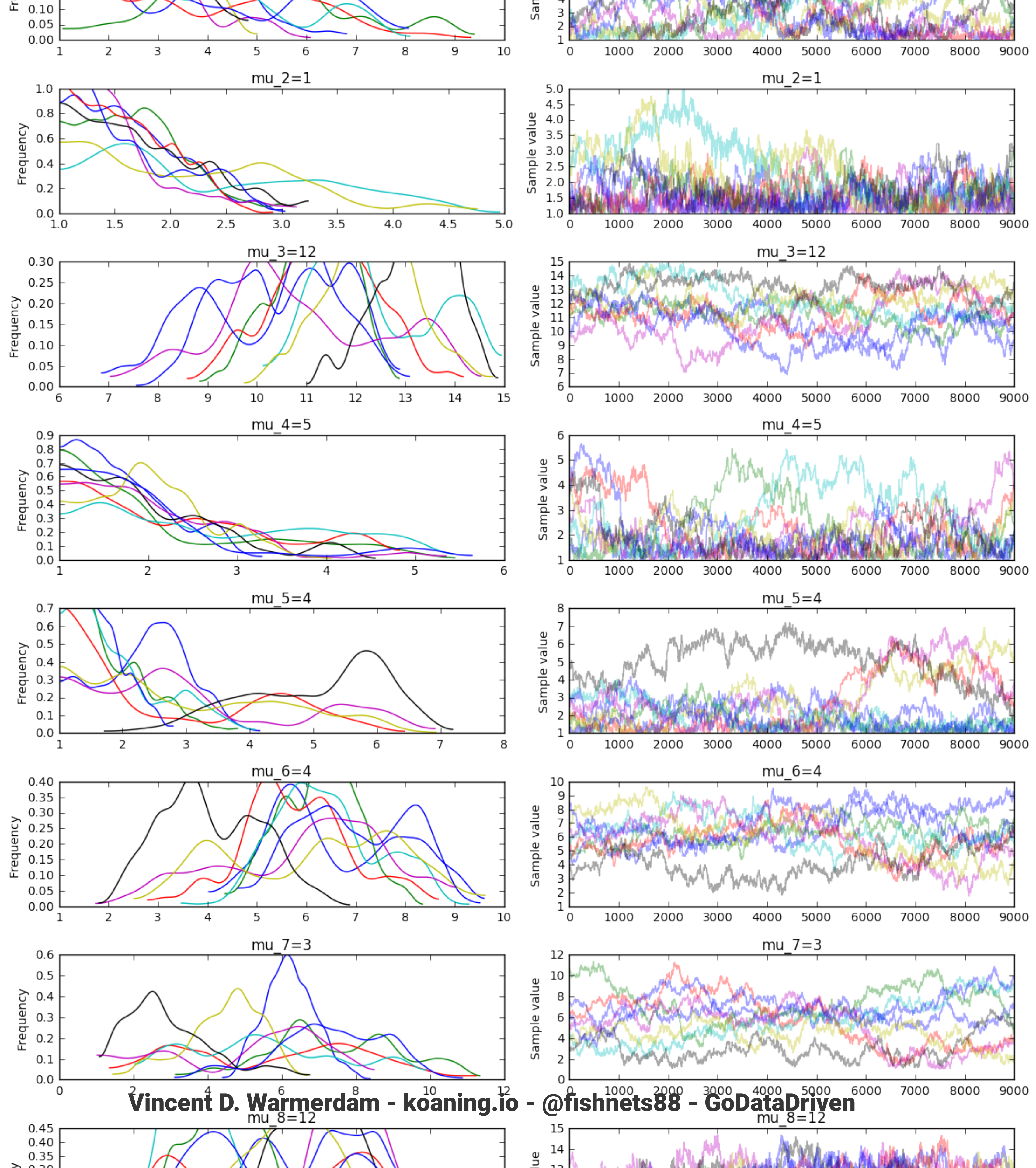
Note; I have 10 unknowns and 6 equations.



The view

As you can see, it starts getting all over the place. There are multiple ways to explain the weight of a bag and this is expressed by the variance in our found samples. Note that some weights seem to have converged but certainly not all.

This was not going to beat kaggle for me.



The view

As you can see, it starts getting all over the place. There are multiple ways to explain the weight of a bag and this is expressed by the variance in our found samples. Note that some weights seem to have converged but certainly not all.

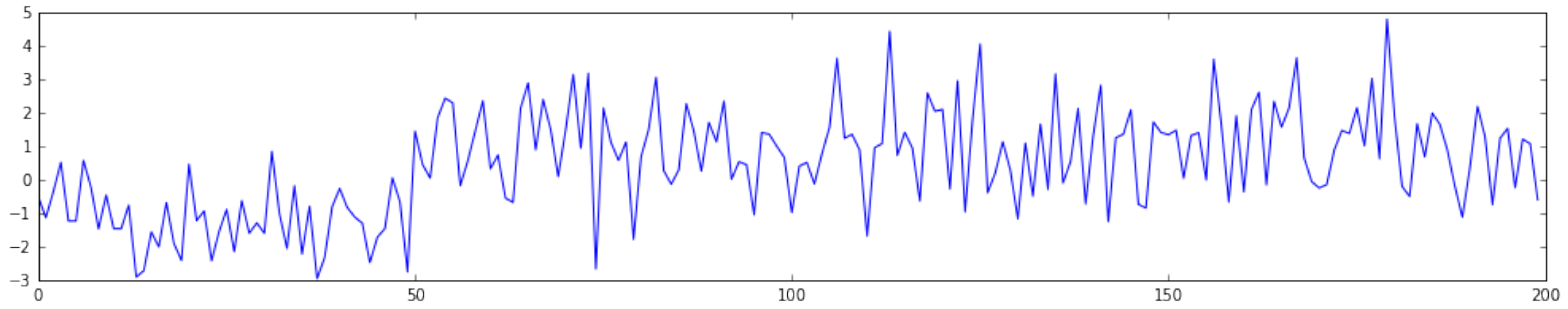
This was not going to beat kaggle for me.

Next Example

Let's now consider timeseries.

The Switchpoint Problem

Let's say we have some signal over time and suddenly a switch in the system happends. We want to have an algorithm find this switch for us in hindsight.



The Switchpoint Problem

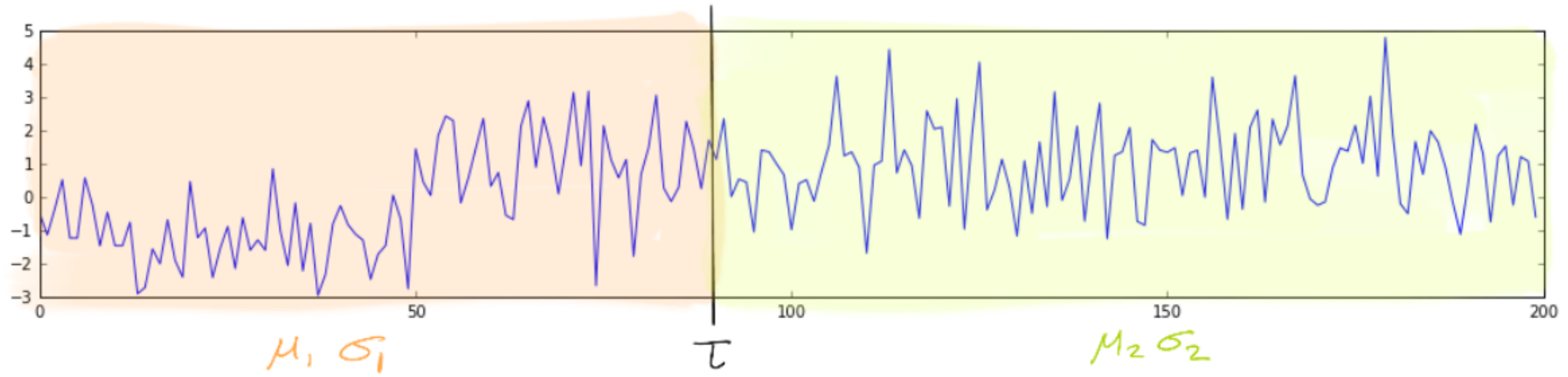
The goal for us is to find out where the changepoint is algorithmically as well as derive the mean and variance of the timeseries before and after the changepoint.

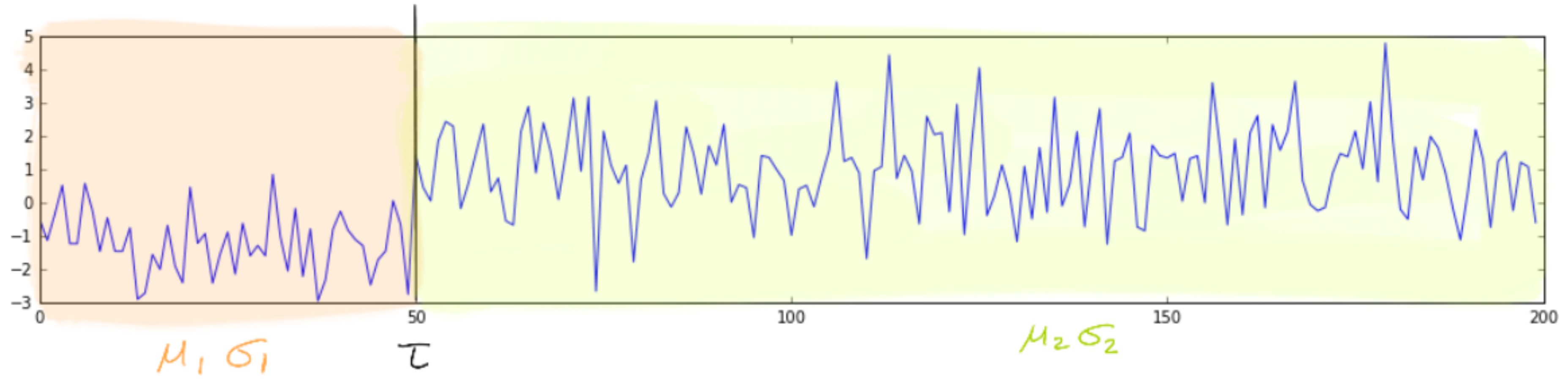
$$\mathbb{P}(\theta|D)$$

In this case $\theta = \{s, \mu_1, \mu_2, \sigma_1, \sigma_2\}$, where s is the moment in time where the switch occured, μ_1, σ_1 correspond to the mean and variance before the switchpoint and μ_2, σ_2 correspond to the mean and variance after the switchpoint.

The Switchpoint Problem

Or, in laymans terms.





The Switchpoint Problem

$$\mathbb{P}(\theta|D) = \mathbb{P}(s, \mu_1, \mu_2, \sigma_1, \sigma_2|D)$$

The nice thing about being a bayesian is that you can use maths to help you think about what this means.

$$\begin{aligned}\mathbb{P}(s, \mu_1, \mu_2, \sigma_1, \sigma_2|D) &\propto \mathbb{P}(D|s, \mu_1, \mu_2, \sigma_1, \sigma_2)\mathbb{P}(s, \mu_1, \mu_2, \sigma_1, \sigma_2) \\ &= \Pi_i \mathbb{P}(y_i|s, \mu_1, \mu_2, \sigma_1, \sigma_2)\mathbb{P}(s, \mu_1, \mu_2, \sigma_1, \sigma_2)\end{aligned}$$

Now let's think about the two parts in that equation

The Switchpoint Problem

Part 1: $\Pi_i \mathbb{P}(y_i | s, \mu_1, \mu_2, \sigma_1, \sigma_2)$

Remember the picture from before?

We can write that down in maths.

$$\mathbb{P}(y_i | s, \mu_1, \mu_2, \sigma_1, \sigma_2) = \begin{cases} y_i \sim N(\mu_1, \sigma_1) & \text{if } i \leq s \\ y_i \sim N(\mu_2, \sigma_2) & \text{if } i > s \end{cases}$$

The Switchpoint Problem

Part 2: $\mathbb{P}(s, \mu_1, \mu_2, \sigma_1, \sigma_2)$

This part of the equation describes our prior belief; what do we know about the parameters before looking at any data?

- We know that the switchpoint s is an integer between 0 and the length of the timeseries.
- The variance is always positive ie. $\sigma > 0$

Recipe

Again, let's repeat the concept.

1. We have data D and unknown parameters $s, \mu_1, \mu_2, \sigma_1, \sigma_2$.

2. We are going to sample around the $[s, \mu_1, \mu_2, \sigma_1, \sigma_2]$-space, markov style.
   2a. If the next parameter sample is more likely, move.
   2b. If not, apply a coin flip and decide to stay or move on.

3. Repeat a whole lotta times.

PyMC3 to the rescue

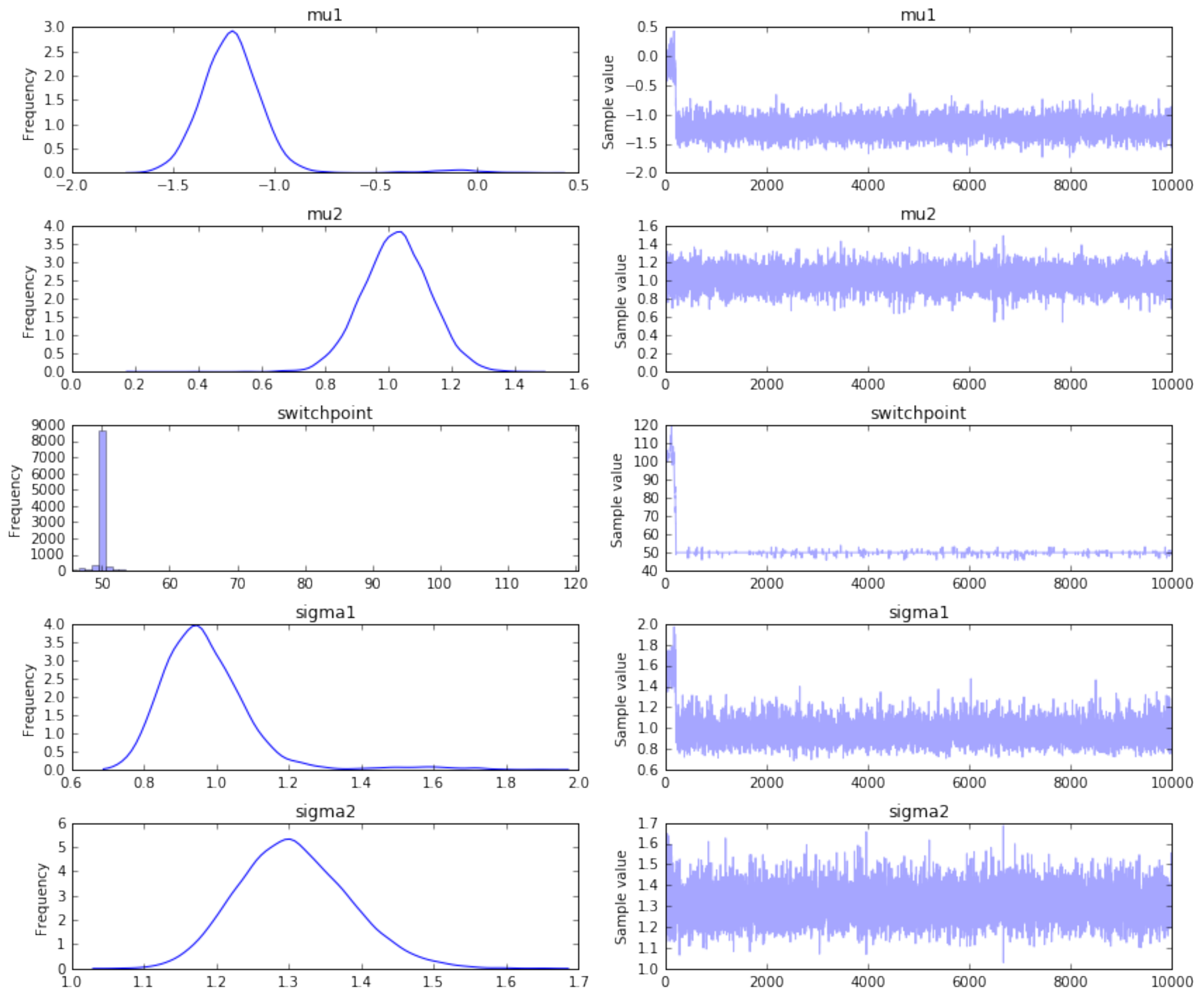
```
import pymc3 as pm
basic_model = pm.Model()

with basic_model:
    mu1 = pm.Normal('mu1', mu=0, sd=2)
    mu2 = pm.Normal('mu2', mu=0, sd=2)
    sigma1 = pm.HalfNormal('sigma1', sd=2)
    sigma2 = pm.HalfNormal('sigma2', sd=2)
    switchpoint = pm.DiscreteUniform('switchpoint', 0, time.max())
    ...
```

PyMC3 to the rescue

Continued from last slide.

```
with basic_model:
    ...
    tau_mu = pm.switch(time >= switchpoint, mu2, mu1)
    tau_sigma = pm.switch(time >= switchpoint, sigma2, sigma1)

    y = pm.Normal('y1', mu=tau_mu, sd=tau_sigma, observed=x)
    trace = pm.sample(10000)

pm.traceplot(trace)
```

The View



Some results

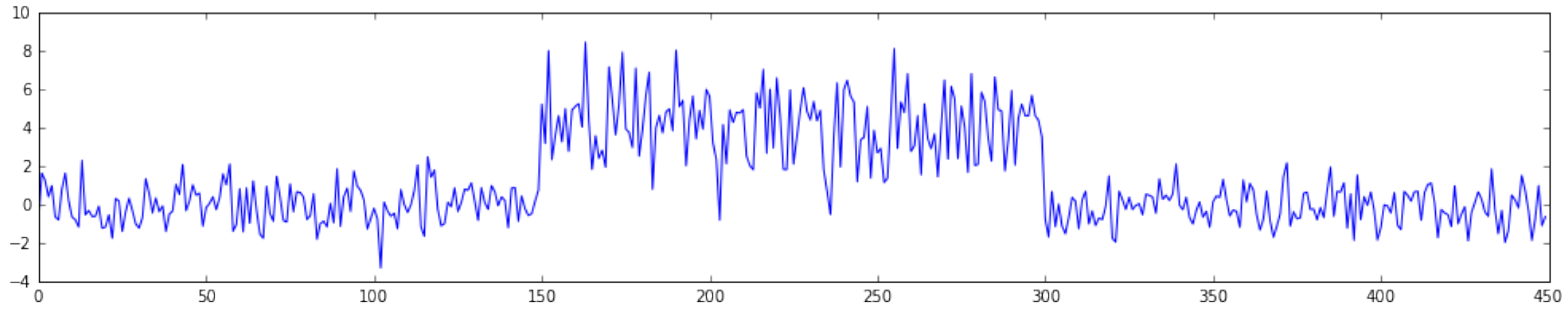
You can also ask for the best parameters...

```
> pm.find_MAP(model=basic_model)
{'mu1': array(-0.10742096434465985),
 'mu2': array(1.023212217472886),
 'sigma1_log_': array(0.44914063604250837),
 'sigma2_log_': array(0.2821837005109984),
 'switchpoint': array(99)}
```

... but! Note that `pm.find_MAP` can get stuck in a hill! It seems to misunderstand integers.

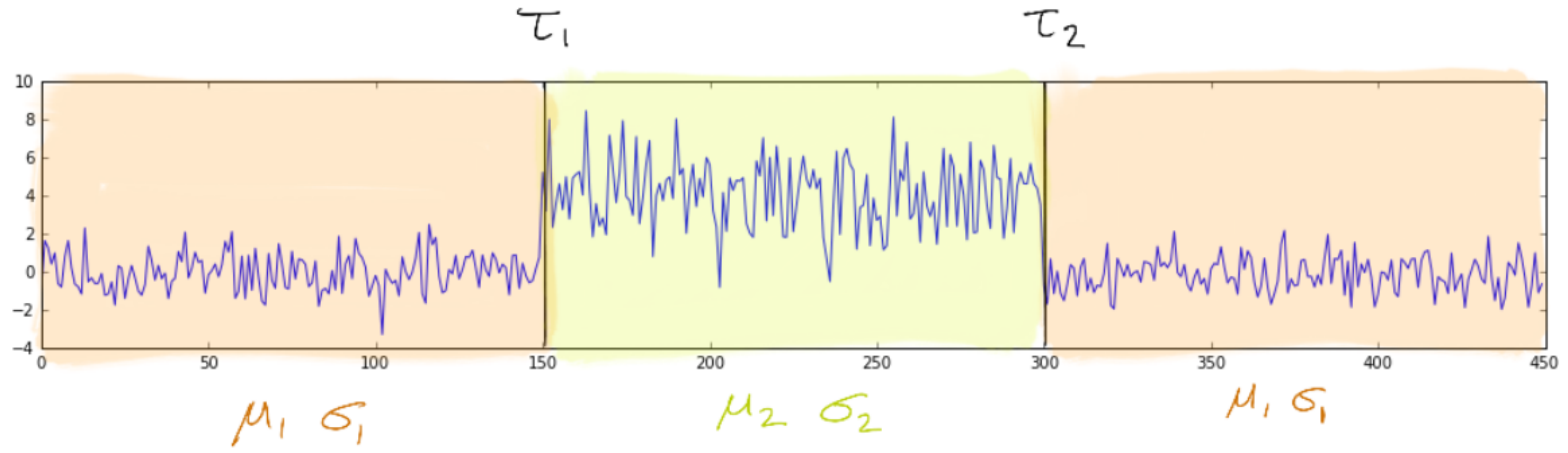
An Extension

We can extend the problem by adding another switchpoint. In this case we assume that there is a reversion. As in; we switch back to our original state.



An Extension

The model changes only slightly.



New Code

Setup for two switchpoints.

```
adv_model = pm.Model()
with adv_model:

    mu1 = pm.Normal('mu1', mu=0, sd=4)
    mu2 = pm.Normal('mu2', mu=0, sd=4)
    sigma1 = pm.HalfNormal('sigma1', sd=4)
    sigma2 = pm.HalfNormal('sigma2', sd=4)
    switchpoint1 = pm.DiscreteUniform('switchpoint1', 0, time.max() - 1)
    switchpoint2 = pm.DiscreteUniform('switchpoint2', switchpoint1, time.max())
    ...
```

New Codw

Continued from previous slide.

```
with adv_model:
    ...
    # formally define the effect from the switchpoints, be careful about order!
    tau_mu1 = pm.switch(time >= switchpoint1, mu2, mu1)
    tau_mu2 = pm.switch(time >= switchpoint2, mu1, tau_mu1)
    tau_sigma1 = pm.switch(time >= switchpoint1, sigma2, sigma1)
    tau_sigma2 = pm.switch(time >= switchpoint2, sigma1, tau_sigma1)

    # define the relationship between observed data and input
    y = pm.Normal('y1', mu=tau_mu2, sd=tau_sigma2, observed=x)
    adv_trace = pm.sample(5000)
```

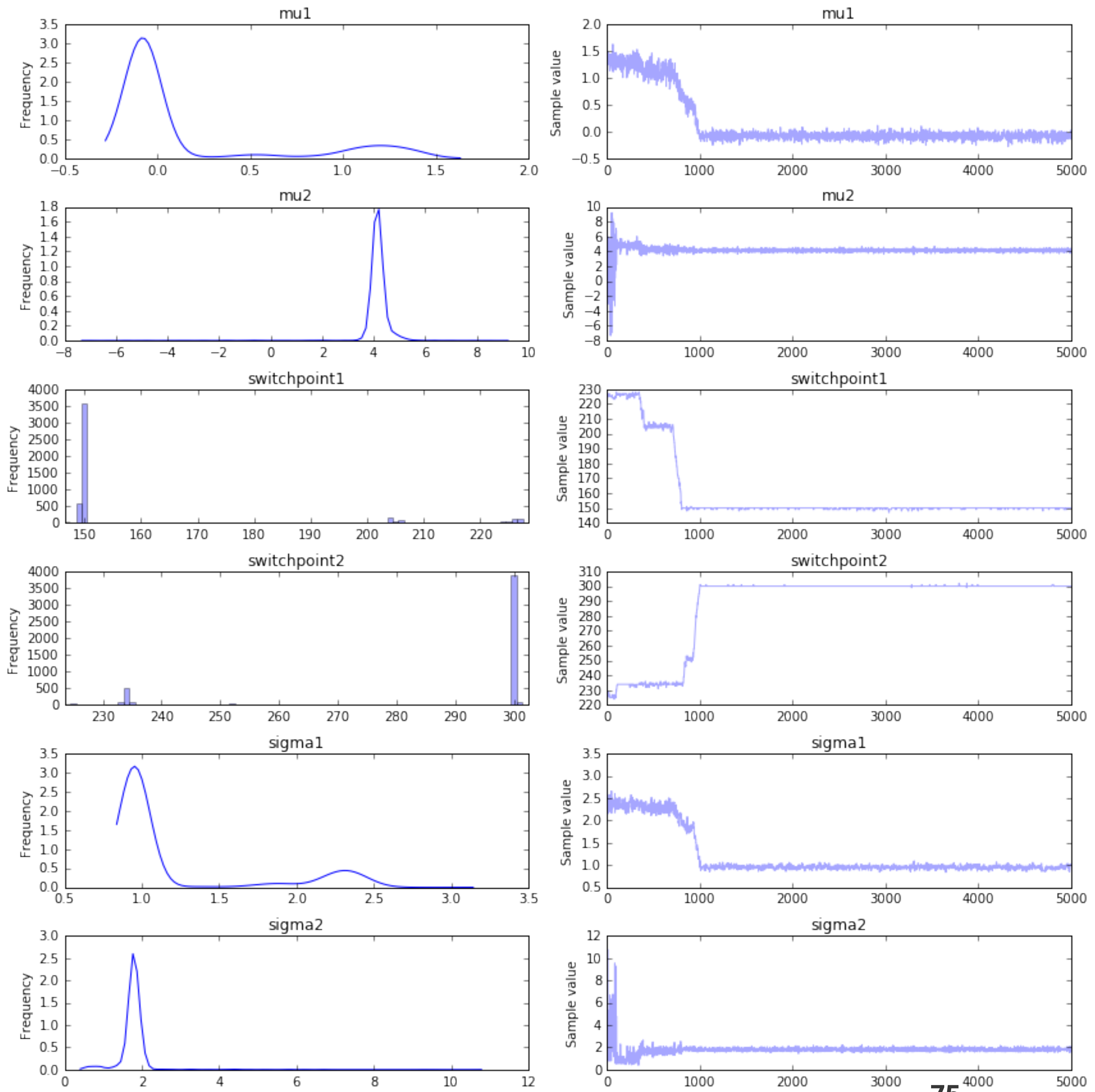
Another View

We can plot the trace again via;

```
pm.traceplot(adv_trace)
```

Notice that the road is more bumpy.

It may be good to ignore the first 20% of the samples.

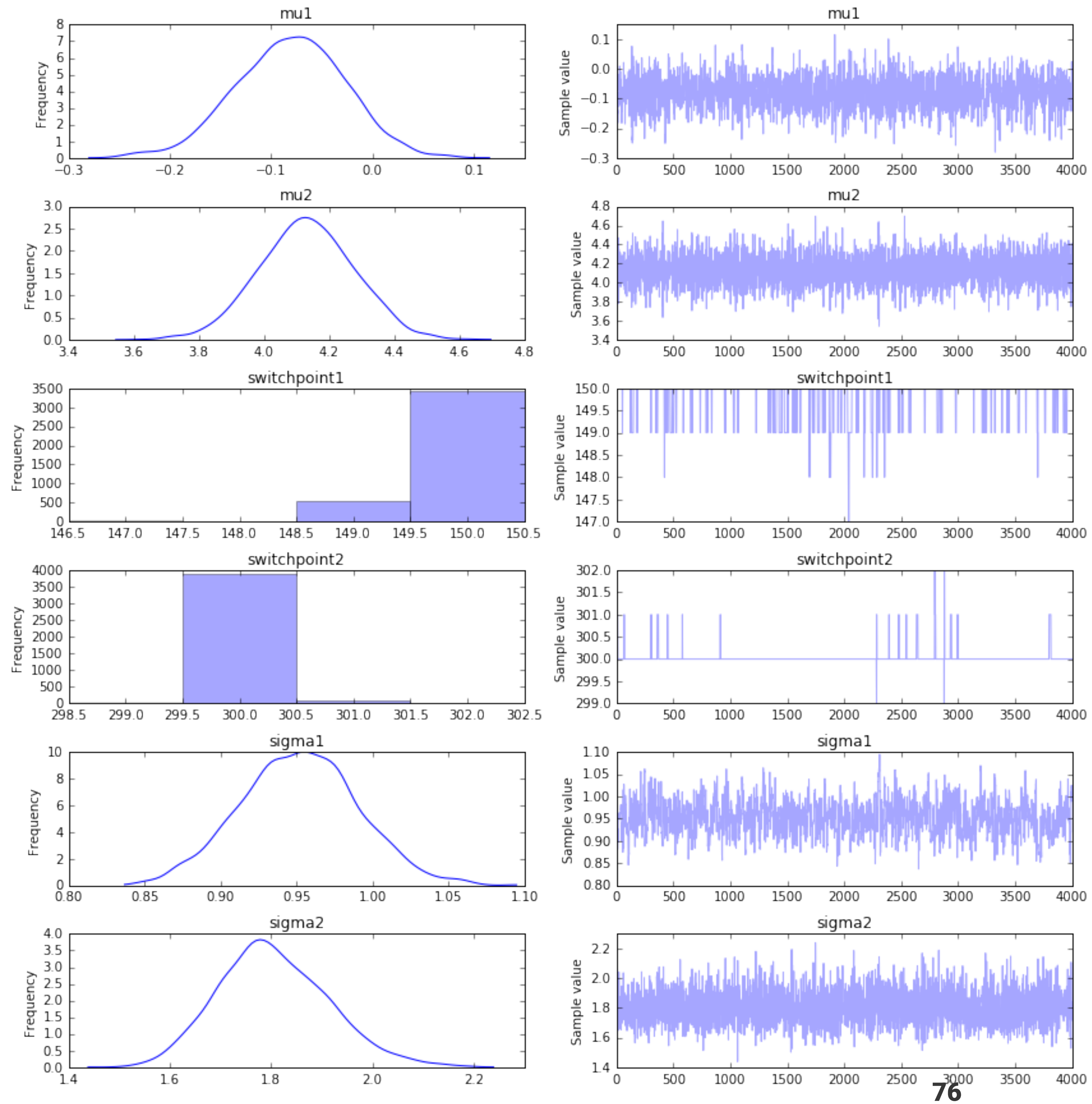


Another [Better] View

```
pm.traceplot(adv_trace[1000:])
```

This looks a bit better. Also the `pm.findMap` works a bit better now. Although integers are still an issue.

```
> pm.find_MAP(model=adv_model)
{'mu1': array(1.3253008730440974),
 'mu2': array(0.0),
 'sigma1_log_': array(0.8639975314985741),
 'sigma2_log_': array(1.3862943591402144),
 'switchpoint1': array(224),
 'switchpoint2': array(224)}
```



Conclusion and a Final Experiment

Let's start with some of the benefits;

- Sampling is a rather general algorithm in terms of models.
- Sampling is parallizable.
- You get an impression of uncertainty of your parameters.
- PyMC3 is kind of a nice library [context managers!]

Conclusion and a Final Experiment

Let us remain critical though. Do we need sampling?

Conclusion and a Final Experiment

Let us remain critical though. Do we need sampling?

We can also do probibalistic programming without sampling, we can use gradients too.

$$\mathbb{P}(y_i|s, \mu_1, \mu_2, \sigma_1, \sigma_2) = \begin{cases} y_i \sim N(\mu_1, \sigma_1) & \text{if } i \leq s \\ y_i \sim N(\mu_2, \sigma_2) & \text{if } i > s \end{cases}$$

Conclusion and a Final Experiment

Let us remain critical though. Do we need sampling?

We can also do probibalistic programming without sampling, we can use gradients too.

$$\mathbb{P}(y_i|s, \mu_1, \mu_2, \sigma_1, \sigma_2) = \begin{cases} y_i \sim N(\mu_1, \sigma_1) & \text{if } i \leq s \\ y_i \sim N(\mu_2, \sigma_2) & \text{if } i > s \end{cases}$$

Is gradient descent maybe not faster?

$$\text{argmax}_{\tau, \mu_1, \mu_2, \sigma_1, \sigma_2} \Pi_i \mathbb{P}(y_i|s, \mu_1, \mu_2, \sigma_1, \sigma_2)$$

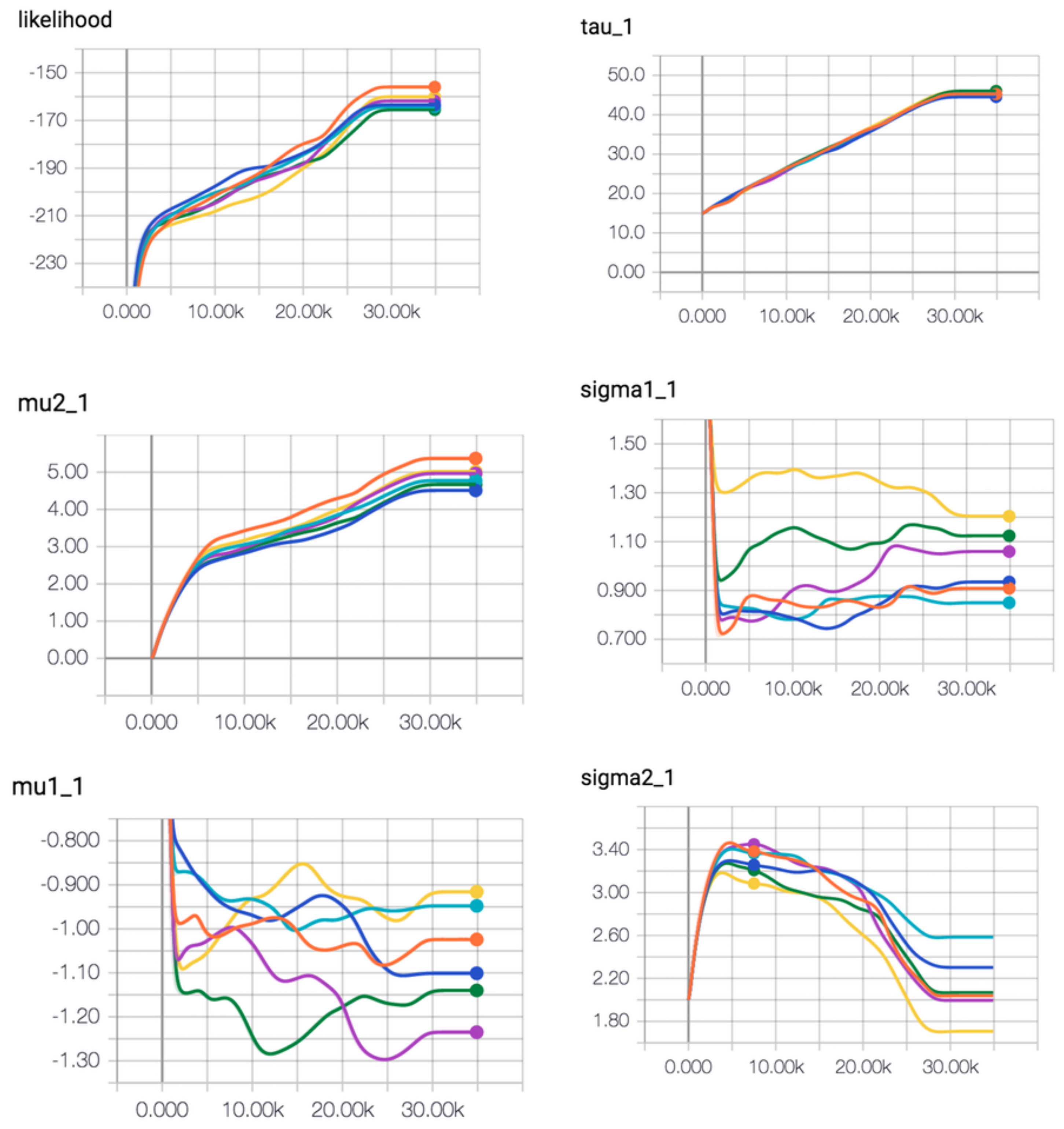
Let's be hip and use tensorboard.

```
import tensorflow as tf
import numpy as np
import os
import uuid

TENSORBOARD_PATH = "/tmp/tensorboard-switchpoint-two-opt"
# tensorboard --logdir=/tmp/tensorboard-switchpoint

x1 = np.random.randn(45)-1
x2 = np.random.randn(45)*2 + 5
x_all = np.hstack([x1, x2])
len_x = len(x_all)
time_all = np.arange(1, len_x + 1)

mu1 = tf.Variable(0, name="mu1", dtype=tf.float32)
mu2 = tf.Variable(0, name = "mu2", dtype=tf.float32)
sigma1 = tf.Variable(2, name = "sigma1", dtype=tf.float32)
sigma2 = tf.Variable(2, name = "sigma2", dtype=tf.float32)
tau = tf.Variable(15, name = "tau", dtype=tf.float32)
switch = 1./(1+tf.exp(tf.pow(time_all - tau, 1)))

mu = switch*mu1 + (1-switch)*mu2
sigma = switch*sigma1 + (1-switch)*sigma2

likelihood_arr = tf.log(tf.sqrt(1/(2*np.pi*tf.pow(sigma, 2)))) - tf.pow(x_all - mu, 2)/(2*tf.pow(sigma, 2))
total_likelihood = tf.reduce_sum(likelihood_arr, name="total_likelihood")

if np.random.random() < 0.5:
    opt_name = "adam"
    optimizer = tf.train.AdamOptimizer()
else:
    opt_name = "momentum"
    optimizer = tf.train.MomentumOptimizer(0.01, 0.01)

opt_task = optimizer.minimize(-total_likelihood)
init = tf.global_variables_initializer()

tf.summary.scalar("mu1", mu1)
tf.summary.scalar("mu2", mu2)
tf.summary.scalar("sigma1", sigma1)
tf.summary.scalar("sigma2", sigma2)
tf.summary.scalar("tau", tau)
tf.summary.scalar("likelihood", total_likelihood)
merged_summary_op = tf.summary.merge_all()

with tf.Session() as sess:
    sess.run(init)
    print("these variables should be trainable: {}".format([_.name for _ in tf.trainable_variables()]))
    uniq_id = os.path.join(TENSORBOARD_PATH, "switchpoint-{}-{}".format(opt_name, str(uuid.uuid1())[:4]))
    summary_writer = tf.summary.FileWriter(uniq_id, graph=tf.get_default_graph())
    for step in range(35000):
        lik, opt, summary = sess.run([total_likelihood, opt_task, merged_summary_op])
        if step % 100 == 0:
            variables = {_.name:_.eval() for _ in [total_likelihood]}
            summary_writer.add_summary(summary, step)
            print("i{}: {}".format(str(step).zfill(5), variables))
```

Why tensorflow?

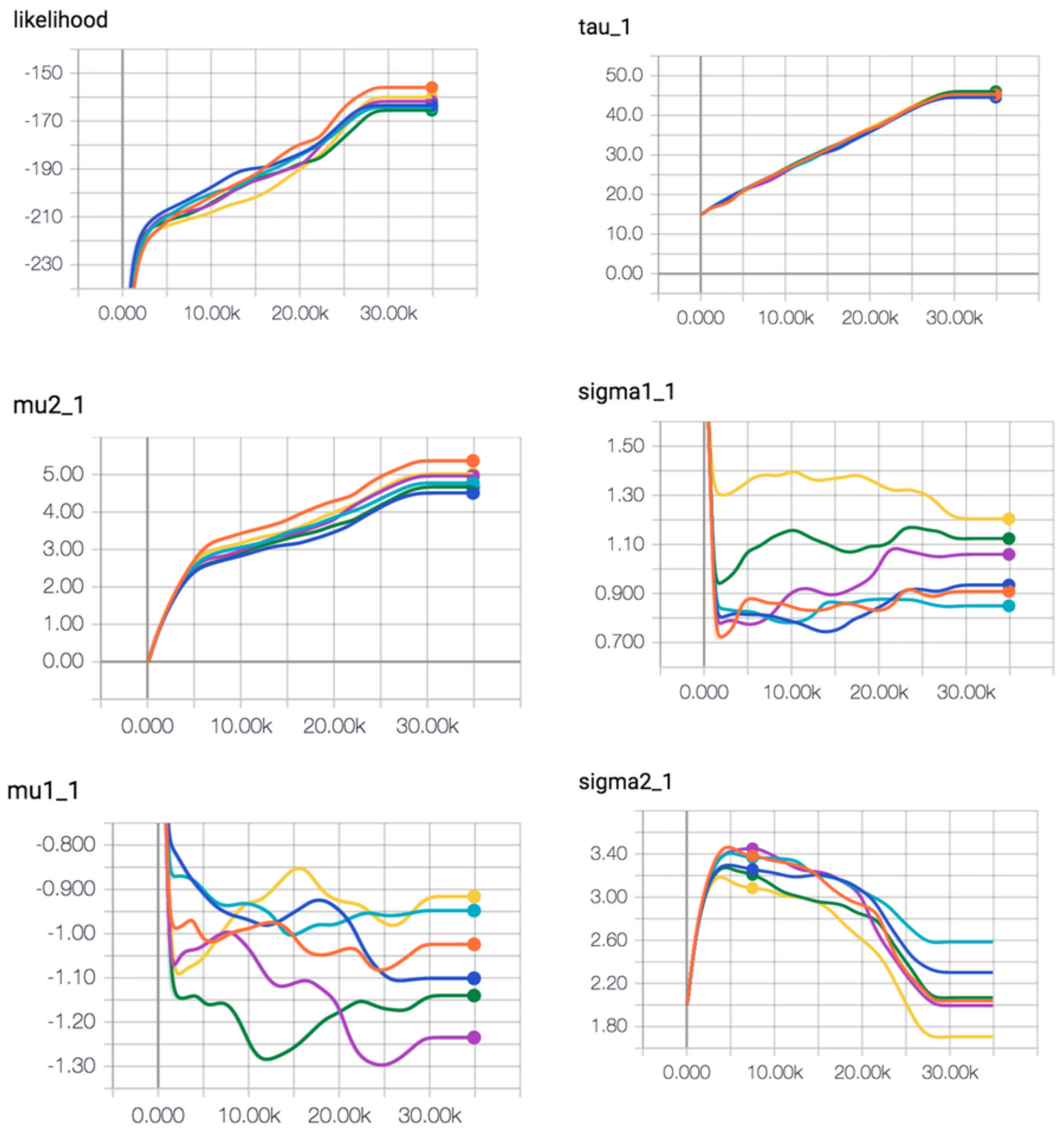
Tensorflow is more than just neural networks people. It is best compared to numpy that is easier to distribute. I used tensorflow for two main reasons;

- Tensorflow has great gradient methods. For free!
- Tensorflow has great support for logging variables. For free!



What did I find?

Gradient descent did work for this problem!



What did I find?

Gradient descent did work for this problem!

It may not always work though. It can also get stuck in hills instead of mountains but you may be able to covercome this with gradient tricks.

Note; all randomness shown here is because I am running simulations.

When to pick?

If you don't have lot's of data, you have uncertainty. You want a posterior. #PYMC3-usecase

If you need to make one single decision, which is very important, you may prefer to be explicit about uncertainty too.

Probibalistic programming in general is super practical in industry. Pymc3 is nice but other tools exist [emcee, stan] and you may be able to use gradient too.