From Scratch: Bayesian Inference, Markov Chain Monte Carlo and Metropolis Hastings, in python

Joseph Moukarzel

Credit: Japan Times

I've been an avid reader on medium/towards data science for a while now, and I've enjoyed the diversity and openness of the subjects tackled by many authors. I wish to contribute to this awesome community by creating my own series of articles "From Scratch", where I explain and implement/build anything from scratch (not necessarily in data science, you need only propose!)

Why do I want to do that? In the current state of things, we are in possession of such powerful libraries and tools that can do a lot of the work for us. Most experienced authors are well aware of the complexities of implementing such tools. As such, they make use of them to provide short, accessible and to the point reads to users from diverse backgrounds. In many of the articles that I enjoyed, I failed to understand how **this** or **that** algorithm is implemented in practise. What are their limitations? Why were they invented? When should they be used?

As Hilary Mason puts it:

> " When you want to use a new algorithm that you don't deeply understand, the best approach is to implement it yourself to learn how it works, and then use a library to benefit from robust code."

That's why, in my very first article, I propose to explain and implement from scratch: Bayesian Inference (somewhat briefly), Markov Chain Monte Carlo and Metropolis Hastings, in Python.

The notebook, and a pdf version can be found on my repository at: joseph94m

Prerequisites: Basic probabilities, calculus and Python.

1- Introduction

In one of the courses during my data science degree, I came across a technique called Markov Chain Monte Carlo, or as it's more commonly referred to, MCMC. The description for this method stated something along the lines of: MCMC is a class of techniques for sampling from a probability distribution and can be used to estimate the distribution of parameters given a set of observations.

Back then, I did not think much of it. I thought, "oh it's just another sampling technique", and I decided I'd read on it when I'd practically need it. This need never emerged, or perhaps it did and I wrongly used something else.

1.1- So why the interest now?

Recently, I have seen a few discussions about MCMC and some of its implementations, specifically the Metropolis-Hastings algorithm and the PyMC3 library. **Markov Chain Monte Carlo in Python A Complete Real-World Implementation** was the article that caught my attention the most. In this great article, William Koehrsen explains how he was able to learn the approach by applying it to a real world problem: to estimate the parameters of a logistic function that represents his sleeping patterns.

$$P(sleep/t, \alpha, \beta) = \frac{1}{1 + e^{\beta t + \alpha}}$$

The assumed model

Mr. Koehrsen uses the PyMC3 implementation of the Metropolis-Hastings algorithm to compute the distribution space of α and β, thus deriving the most likely logistic model.

1.2- So why am I talking about all that?

In this article, I propose to implement from scratch, my own version of the Metropolis-Hastings algorithm to find parameter distributions for a dummy data example and then of a real world problem.

I figured that if I get my hands dirty, I might finally be able to understand it. I will only use numpy to implement the algorithm, and matplotlib to draw pretty things. Alternatively, scipy can be used to compute the density functions (which I will talk about later), but I will also show how to implement them using numpy.

1.3- Flow of the article

In part 1, I will introduce Bayesian inference, MCMC-MH and their mathematical components. In part 2, I will explain the MH algorithm using dummy data. Finally, part 3 will provide a real world application for MCMC-MH.

2- Part 1: Bayesian inference, Markov Chain Monte Carlo, and Metropolis-Hastings

2.1- A bird's eye view on the philosophy of probabilities

In order to talk about Bayesian inference and what follows it, I shall first explain what the Bayesian view of probability is, and situate it within its historical context.

2.1.1- Frequentist vs Bayesian thinking

There are two major interpretations to probabilities: **Bayesian** and **Frequentist**.

From a **Frequentist's** perspective, probabilities represent long term frequencies with which events occur. A frequentist can say that the probability of having tails from a coin toss is equal to 0.5 on the long run. Each new experiment, can be considered as one of an infinite sequence of possible repetitions of the same experiment. The main idea is that there is no belief in a frequentist's view of probability. The probability of event x happening out of n trials is approximately the following :

$$P(x) = \frac{n_x}{n}$$

and the true probability is reached when $n \longrightarrow \infty$. Frequentists will never say "I am 45% (0.45) sure that there is lasagna for lunch today", since this does not happen on the long run. Commonly, a frequentist approach is referred to as the *objective* approach since there is no expression of belief and/or prior events in it.

On the other hand, in **Bayesian** thinking, probabilities are treated as an expression of **belief**. Therefore it is perfectly reasonable for a Bayesian to say "I am 50% (0.5) sure that there is lasagna for lunch today". By combining **prior** beliefs, and current events (the **evidence**), one can compute the **posterior**, i.e. the probability that there is lasagna today. The idea behind Bayesian thinking is to keep updating the beliefs as more evidence is provided. Since this approach deals with belief, it is usually referred to as the *subjective* view on probability.

2.1.2- Bayesian inference

In the philosophy of decision making, Bayesian inference is closely related to the Bayesian view on probability, in the sense that it manipulates **priors**, **evidence**, and **likelihood** to compute the **posterior**. Given some event B, what is the probability that event A occurs? This is answered by Bayes' famous formula:

$$P(A/B) = \frac{P(B/A)P(A)}{P(B)}$$

With:

- $P(A/B)$ is the **posterior**. What we wish to compute.
- $P(B/A)$ is the **likelihood**. Assuming A occured, how likely is B.
- $P(A)$ is the **prior**. How likely the event A is regardless of evidence.
- $P(B)$ is the **evidence**. How likely the evidence B is regardless of the event.

In our case, we are mostly interested in the specific formulation of Bayes' formula:

$$P(\theta/D) = \frac{P(D/\theta)P(\theta)}{P(D)}$$ where, $P(\theta/D)$ is the **posterior**, $P(D/\theta)$ is the **likelihood**, $P(\theta)$ is the **prior** and $P(D)$ is the **evidence**.

That is, we would like to find the most likely distribution of θ, the parameters of the model explaining the data, D.

Computing some of these probabilities can be tedious, especially the evidence P (D). Also, other problems can arise such as those of ensuring **conjugacy** which I will not dive into in this article. Luckily, some techniques, namely MCMC, allow us to sample from the posterior, and a draw distributions over our parameters without having to worry about computing the evidence, nor about conjugacy.

2.1.3- Markov Chain Monte Carlo

MCMC allows us to draw samples from a distribution even if we can't compute it. It can be used to sample from the posterior distribution (what we wish to know) over parameters. It has seen much success in many applications, such as computing the distribution of parameters given a set of observations and some prior belief, and also computing high dimensional integrals in physics and in digital communications.

Bottom line: **It can be used to compute the distribution over the parameters given a set of observations and a prior belief.**

2.1.4- Metropolis-Hastings

MCMC is a class of methods. Metropolis-Hastings is a specific implementation of MCMC. It works well in high dimensional spaces as opposed to Gibbs sampling and rejection sampling.

This technique requires a simple distribution called the proposal distribution (Which I like to call transition model) $Q(\theta'/\theta)$ to help draw samples from an intractable posterior distribution $P(\Theta = \theta/D)$.

Metropolis-Hastings uses Q to randomly walk in the distribution space, accepting or rejecting jumps to new positions based on how likely the sample is. This "memoriless" random walk is the "Markov Chain" part of MCMC.

The "likelihood" of each new sample is decided by a function f. That's why f must be proportional to the posterior we want to sample from. f is commonly chosen to be a probability density function that expresses this proportionality.

To get a new position of the parameter, just take our current one, θ, and propose a new one , θ', that is a random sample drawn from Q(θ'/θ). Often this is a symmetric distribution. For instance, a normal distribution with mean θ and some standard deviation σ: Q(θ'/θ) = N(θ, σ)

To decide if θ' is to be accepted or rejected, the following ratio must be computed for each new proposed θ':

$$\frac{P(\theta'/D)}{P(\theta/D)}$$

Using Bayes' formula this can be easily re-formulated as:

$$\frac{P(D/\theta')P(\theta')}{P(D/\theta)P(\theta)}$$

The evidence P(D) is simply crossed out during the division

Which is also equivalent to:

$$\frac{\prod_i^n f(d_i/\Theta = \theta')P(\theta')}{\prod_i^n f(d_i/\Theta = \theta)P(\theta)}$$

Where f is the proportional function mentioned previously.

The rule for acceptance, can then be formulated as:

$$P(\text{accept}) = \begin{cases} \dfrac{\prod_i^n f(d_i/\Theta = \theta')P(\theta')}{\prod_i^n f(d_i/\Theta = \theta)P(\theta)}, & \prod_i^n f(d_i/\Theta = \theta)P(\theta) > \prod_i^n f(d_i/\Theta = \theta')P(\theta') \\ 1, & \prod_i^n f(d_i/\Theta = \theta)P(\theta) \le \prod_i^n f(d_i/\Theta = \theta')P(\theta') \end{cases}$$

Note: The prior components are often crossed if there is no preference or restrictions on the parameters.

This means that if a θ' is more likely than the current θ, then we always accept θ'. If it is less likely than the current θ, then we might accept it or reject it randomly with decreasing probability, the less likely it is.

So, briefly, the Metropolis-Hastings algorithm does the following:

- given:
    - f, the PDF of the distribution to sample from
    - Q, the transition model
    - θ_0, a first guess for θ
    - $\theta = \theta_0$
- for n iterations
    - $p = f(D/\Theta = \theta)P(\theta)$
    - $\theta' = Q(\theta_i)$
    - $p' = f(D/\Theta = \theta')P(\theta')$
    - $ratio = \dfrac{p'}{p}$
    - generate a uniform random number r in [0,1]
    - if $r < ratio$:
        - set $\theta_i = \theta'$

Metropolis-Hastings algorithm

3- Part 2: Dummy data example

3.1- Step 1: Data generation

We generate 30,000 samples from a normal distribution with mean μ= 10, and standard deviation σ= 3, but we can only observe 1000 random samples from them.

```
1    mod1=lambda t:np.random.normal(10,3,t)

2

3    #Form a population of 30,000 individual, with average=10 and scale=3

4    population = mod1(30000)

5    #Assume we are only able to observe 1,000 of these individuals.

6    observation = population[np.random.randint(0, 30000, 1000)]

7

8    fig = plt.figure(figsize=(10,10))

9    ax = fig.add_subplot(1,1,1)

10   ax.hist( observation,bins=35 ,)

11   ax.set_xlabel("Value")

12   ax.set_ylabel("Frequency")

13   ax.set_title("Figure 1: Distribution of 1000 observations sampled from a population of 30,

14   mu_obs=observation.mean()

15   mu_obs
```


Figure 1: Distribution of 1000 observations sampled from a population of 30,000 with $\mu=10$, $\sigma=3$
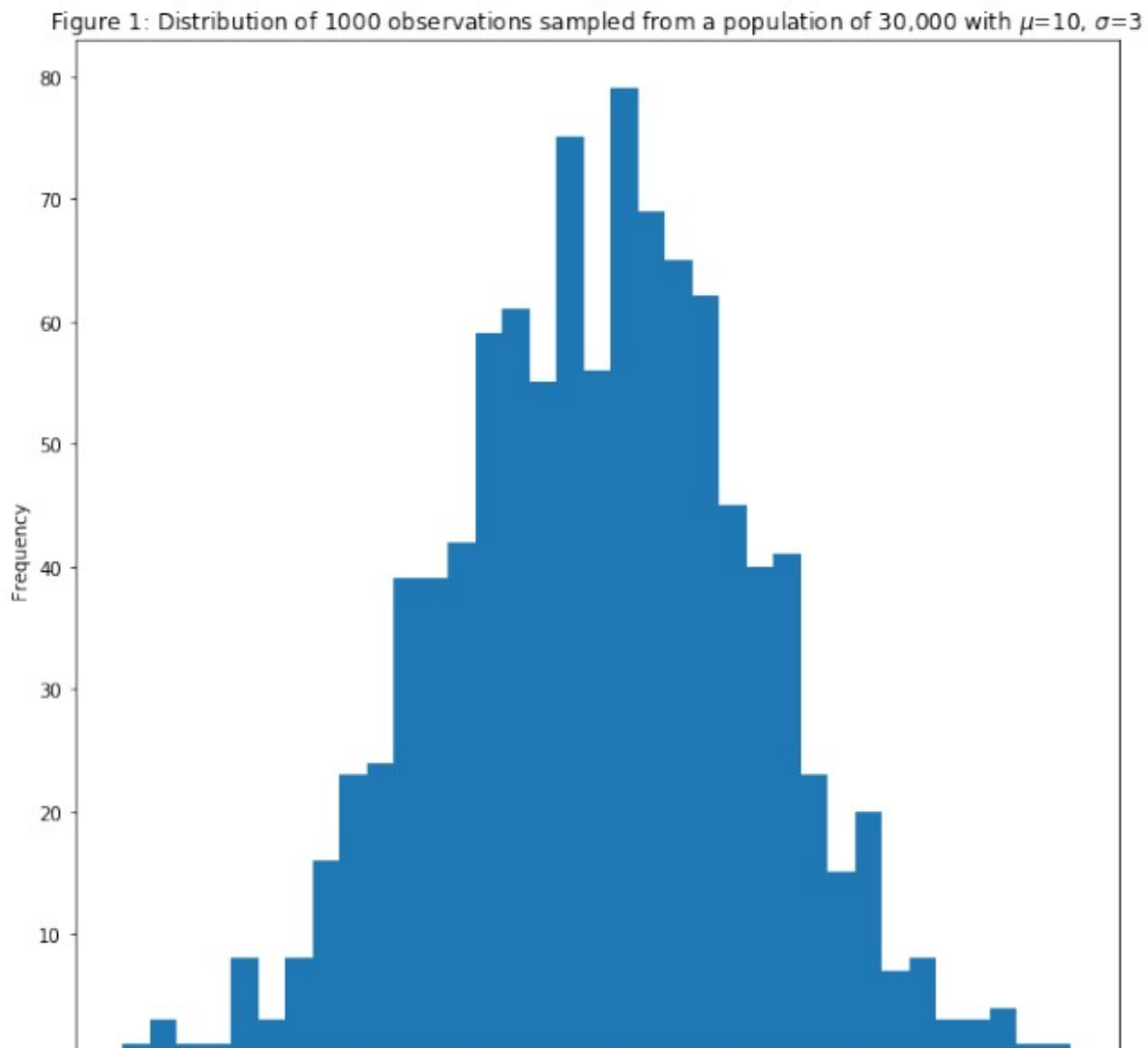
3.2- Step 2: What do we want?

We would like to find a distribution for σ{observed} using the 1000 observed samples. Those of you who are avid mathematicians will say that there is a formula for computing σ:

$$\sigma = \sqrt{\frac{1}{n} \Sigma_i^n (d_i - \mu)^2}$$

Why do we want to sample and whatnot? Well, this is just a dummy data example, the real problem is in part 3, where it's hard to compute the parameters directly. Plus here, we are not trying to find a value for σ, but rather, we are trying to compute a probability distribution for σ.

3.3- Step 3: Define the PDF and the transition model

From Figure 1, we can see that the data is normally distributed. The mean can be easily computed by taking the average of the values of the 1000 samples. By doing that, we get that μ{observed} = 9.8 (although on a side note, we could have also assumed μ to be unknown and sampled for it just like we are doing for σ. However, I want to make this starting example simple.)

3.3.1- For the transition model/ proposal distribution

I have no specific distribution in mind, so I will choose a simple one: the Normal distribution!

$$Q(\sigma_{new}/\sigma_{current}) = N(\mu = \sigma_{current}, \sigma' = 1)$$

Note that σ' is unrelated to σ{new} and σ{current}. It simply specifies the standard deviation of the parameter space. It can be any value desired. It affects the convergence time of the algorithm and the correlation between samples, which I talk about later.

3.3.2- For the PDF

Since *f* should be proportional to the posterior, we choose *f* to be the following Probability Density Function (PDF), for each data point di in the data set D:

$$f(d_i/\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(d_i - \mu)^2}{2\sigma^2}}$$

In our case, θ is made up of two values: $[\mu, \sigma]$, and that μ is a constant, $\mu = \mu_{obs}$.

Since μ is constant, we can practically consider that σ is equivalent to θ

3.4- Step 4: Define when we accept or reject σ{new}

3.4.1- The acceptance formula

We accept σ_{new} if:

$$\frac{Likelihood(D/\mu_{obs}, \sigma_{new}) * prior(\mu_{obs}, \sigma_{new})}{Likelihood(D/\mu_{obs}, \sigma_{current}) * prior(\mu_{obs}, \sigma_{current})} > 1 \qquad (1)$$

If this ratio is not larger than 1, then we compare it to a uniformly generated random number in the closed set [0,1]. If the ratio is larger than the random number, we accept σ{new}, otherwise we reject it. This ensures that even if a sample is less likely than the current, we might still want to try it. (Similar notion to simulated annealing)

3.4.2- The likelihood

The total likelihood for a set of observation D is:

$$Likelihood(D/\mu_{obs}, \sigma_a) = \prod_i^n f(d_i/\mu_{obs}, \sigma_a), \text{ where } a = new \text{ or } current.$$

This must be computed for both new and current sigma in order to compute the ratio in equation (1)

3.4.3- The Prior P(μ,σ)

We don't have any preferences for the values that σ{new} and σ{current} can take. The only thing worth noting is that they should be positive. Why? Intuitively, the standard deviation measures dispersion. Dispersion is a distance, and distances cannot be negative.

Mathematically:

$$\sigma = \sqrt{\frac{1}{n} \sum_i^n (d_i - \mu)^2}$$

and the square root of a number cannot be negative, so σ is always positive. We strictly enforce this in the prior.

3.4.4- The final acceptance form

In our case, we will log both the prior and the likelihood function. Why log? Simply because it helps with numerical stability, i.e. multiplying thousands of small values (probabilities, likelihoods, etc..) can cause an underflow in the system's memory, and the log is a perfect solution because it transforms multiplications to additions and transforms small positive numbers into non-small negative numbers.

Our acceptance condition from equation (1) becomes:

Accept σ_{new} if:

$$Log(Likelihood(D/\mu_{obs}, \sigma_{new})) + Log(prior(\mu_{obs}, \sigma_{new})) - (Log(Likelihood(D/\mu_{obs}, \sigma_{current})) + Log(prior(\mu_{obs}, \sigma_{current}))) > 0$$

Equivalent to:

$$\sum_i^n Log(f(d_i/\mu_{obs}, \sigma_{new})) + Log(prior(\mu_{obs}, \sigma_{new})) - \sum_i^n Log(f(d_i/\mu_{obs}, \sigma_{current})) - Log(prior(\mu_{obs}, \sigma_{current})) > 0$$

Equivalent to:

$$\sum_i^n Log(f(d_i/\mu_{obs}, \sigma_{new})) + Log(prior(\mu_{obs}, \sigma_{new})) > \sum_i^n Log(f(d_i/\mu_{obs}, \sigma_{current})) + Log(prior(\mu_{obs}, \sigma_{current}))$$

Equivalent to:

$$\sum_i^n -n Log(\sigma_{new}\sqrt{2\pi}) - \frac{(d_i - \mu_{obs})^2}{2\sigma_{new}^2} + Log(prior(\mu_{obs}, \sigma_{new})) \quad >$$

$$\text{(2)}$$

$$\sum_i^n -n Log(\sigma_{current}\sqrt{2\pi}) - \frac{(d_i - \mu_{obs})^2}{2\sigma_{current}^2} + Log(prior(\mu_{obs}, \sigma_{current}))$$

This form can be reduced even more by taking the square root and the multiplication out of the log, but never mind that now, there's already enough maths!

3.4.5- The implementation of that rant

```
1    #The tranistion model defines how to move from sigma_current to sigma_new
2    transition_model = lambda x: [x[0],np.random.normal(x[1],0.5,(1,))]
3
4    def prior(x):
5        #x[0] = mu, x[1]=sigma (new or current)
6        #returns 1 for all valid values of sigma. Log(1) =0, so it does not affect the summati
7        #returns 0 for all invalid values of sigma (<=0). Log(0)=-infinity, and Log(negative n
8        #It makes the new sigma infinitely unlikely.
9        if(x[1] <=0):
10            return 0
11       return 1
12
13   #Computes the likelihood of the data given a sigma (new or current) according to equation
14   def manual_log_like_normal(x,data):
15       #x[0]=mu, x[1]=sigma (new or current)
16       #data = the observation
17       return np.sum(-np.log(x[1] * np.sqrt(2* np.pi) )-((data-x[0])**2) / (2*x[1]**2))
18
19   #Same as manual log like normal(x,data), but using scipy implementation. It's pretty slow.
```

The implementation is quite simple, right ?!

3.6- Step 6: Run the algorithm with initial parameters and collect accepted and rejected samples

The algorithm accepted 8317 samples (which might be different on each new run). The last 10 samples contain the following values for σ:

[2.87920187, 3.10388928, 2.94469786, 3.04094103, 2.95522153, 3.09328088, 3.07361275, 3.08588388, 3.12881964, 3.03651136]

Let's see how the algorithm worked its way to these values:

Figure 2: MCMC sampling for σ with Metropolis-Hastings. First 50 samples are shown.

Figure 3: MCMC sampling for σ with Metropolis-Hastings. All samples are shown.
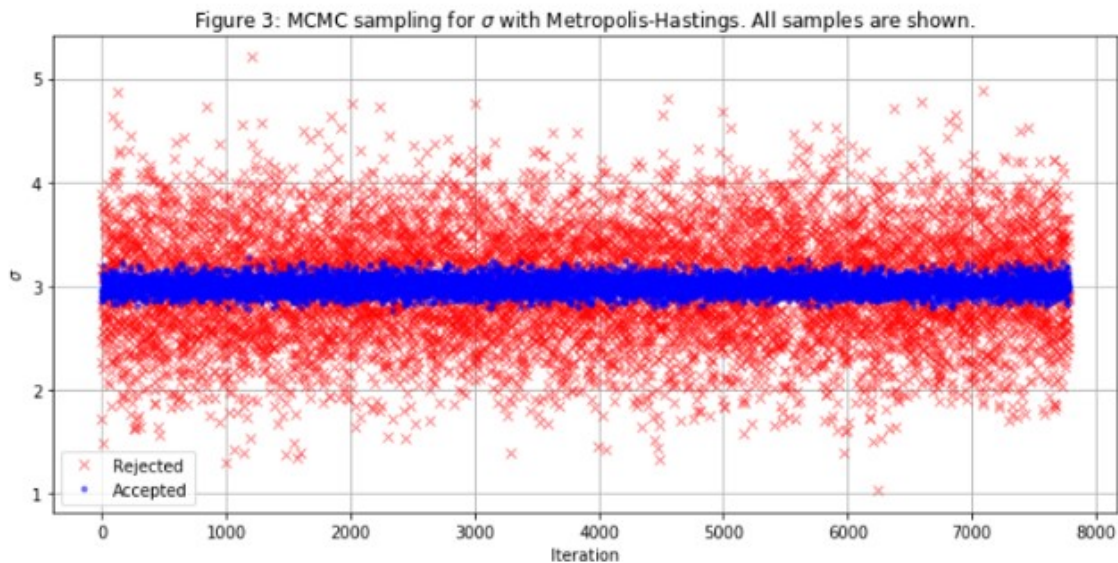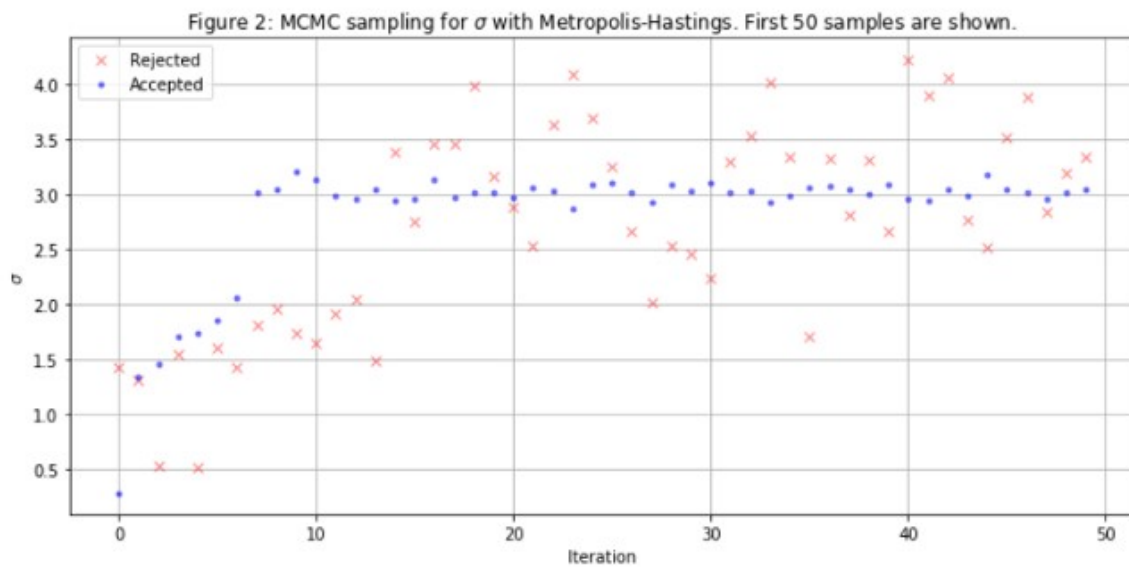
So, starting from an initial σ of 0.1, the algorithm converged pretty quickly to the expected value of 3. That said, it's only sampling in a 1D space…. so it's not very surprising.

Still, we will consider the initial 25% of the values of σ to be "burn-in", so we drop them.

3.6.2- Let's visualise the trace of σ and the histogram of the trace
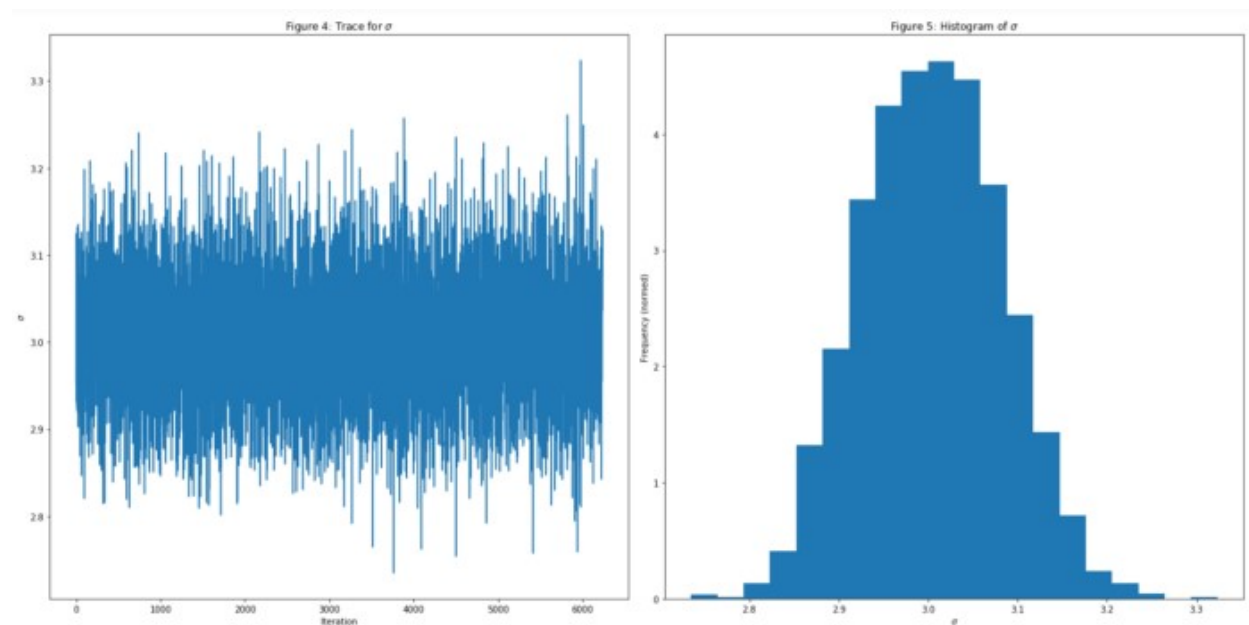
The most likely value for σ is around 3.1. This is a wee bit more than the original value of 3.0. The difference is due to us observing only 3.33% of the original population (1,000 out of 30,000)

3.6.3- Predictions: How would our model fare in predicting the original population of 30,000?

First, we average the last 75% of accepted samples of σ, and we generate 30,000 random individuals from a normal distribution with μ=9.8 and σ=3.05 (the average of the last 75% of accepted samples) which is actually better than the most likely value of 3.1.

And voilà:

Figure 6: Posterior distribution of predicitons
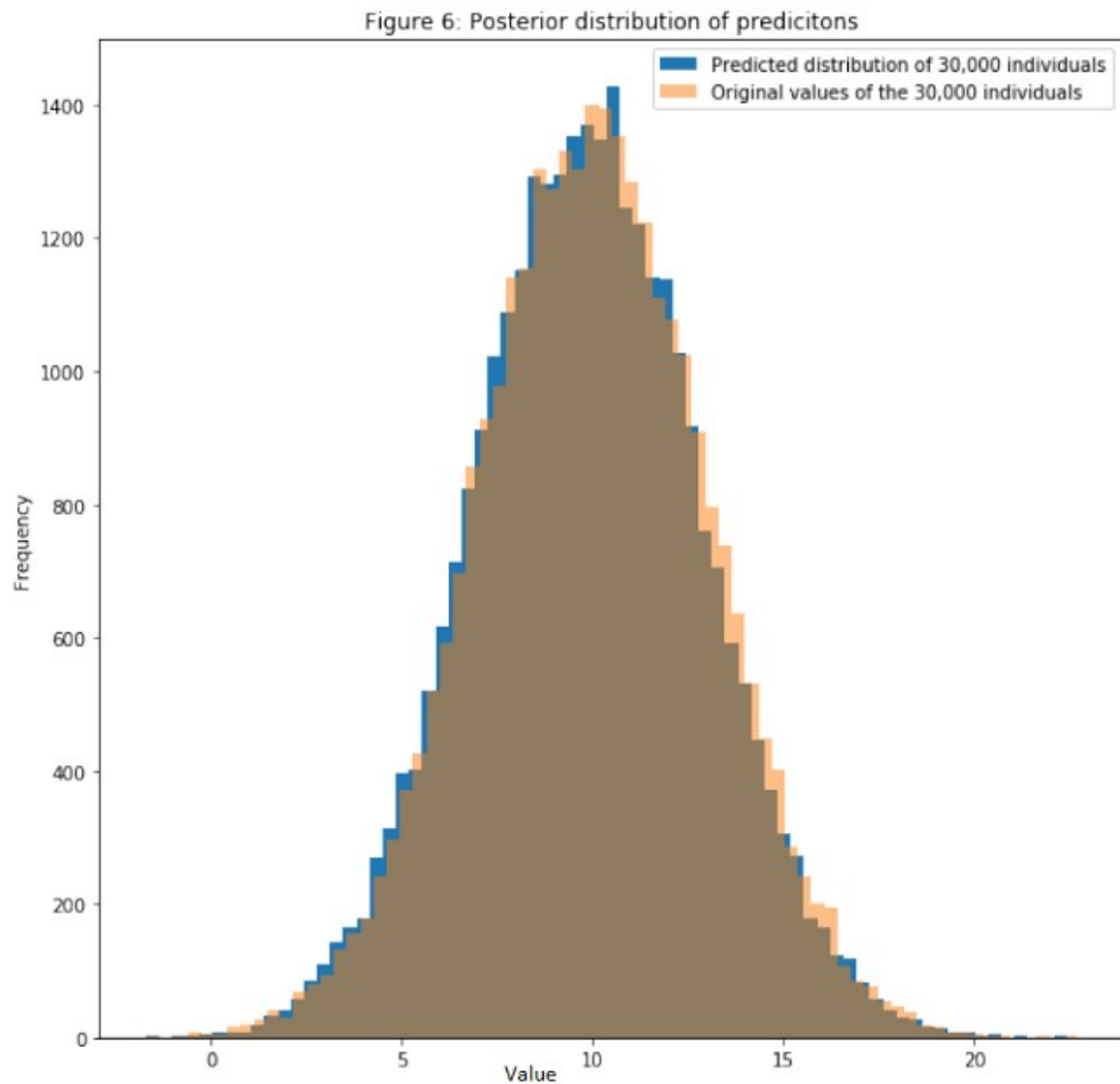
And now, onto the real stuff!

4- Part 3: A real world example
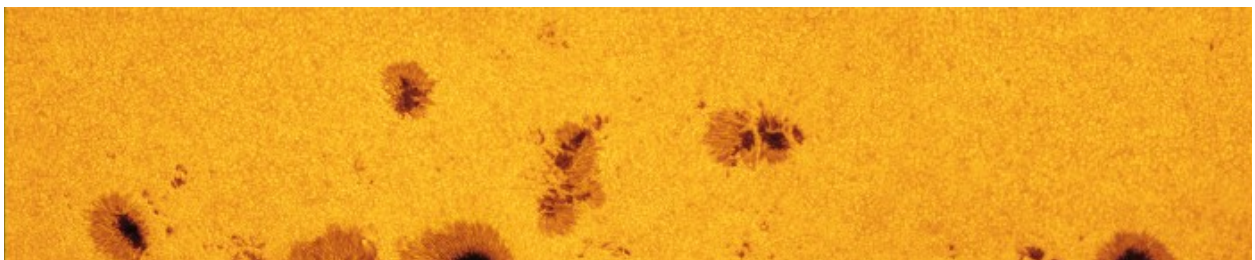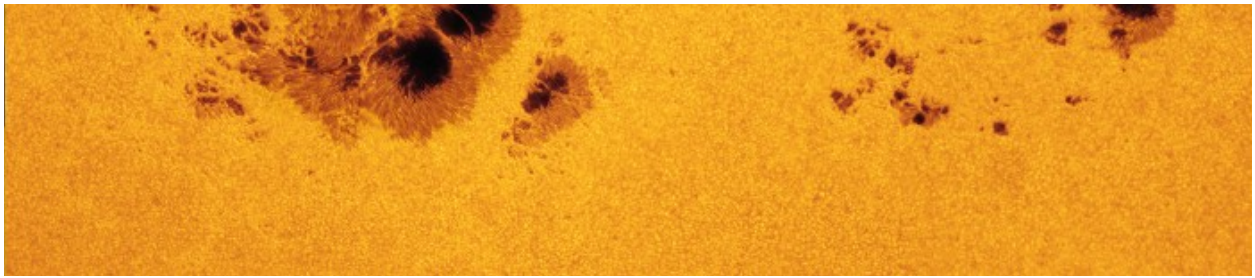
A sunspot is a region on the Sun's surface (photo-sphere) that is marked by a lower temperature than its environment. These reduced temperatures are caused by concentrations of magnetic field flux that inhibit convection by an effect similar to eddy current brakes. Sunspots usually appear in pairs of opposite magnetic polarity. Their number varies according to the approximately 11-year solar cycle.

The data we will be working on is the "Monthly mean total sunspot number", for each month from January 1749 to November 2018. This data is collected, curated and made publicly available by the World Data Center for the production, preservation and dissemination of the international sunspot number.

4.1- Let's plot the data over the years to see what the distribution might be like

Figure 7: Sunspots, 1749-2018

Figure 8: Sunspots, 1749-1785
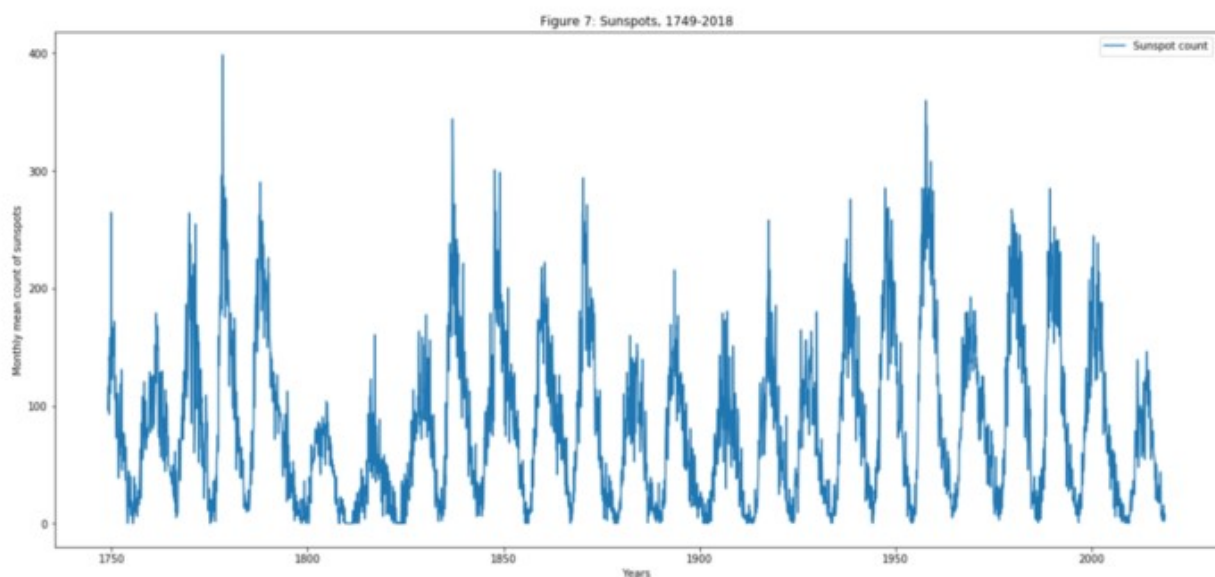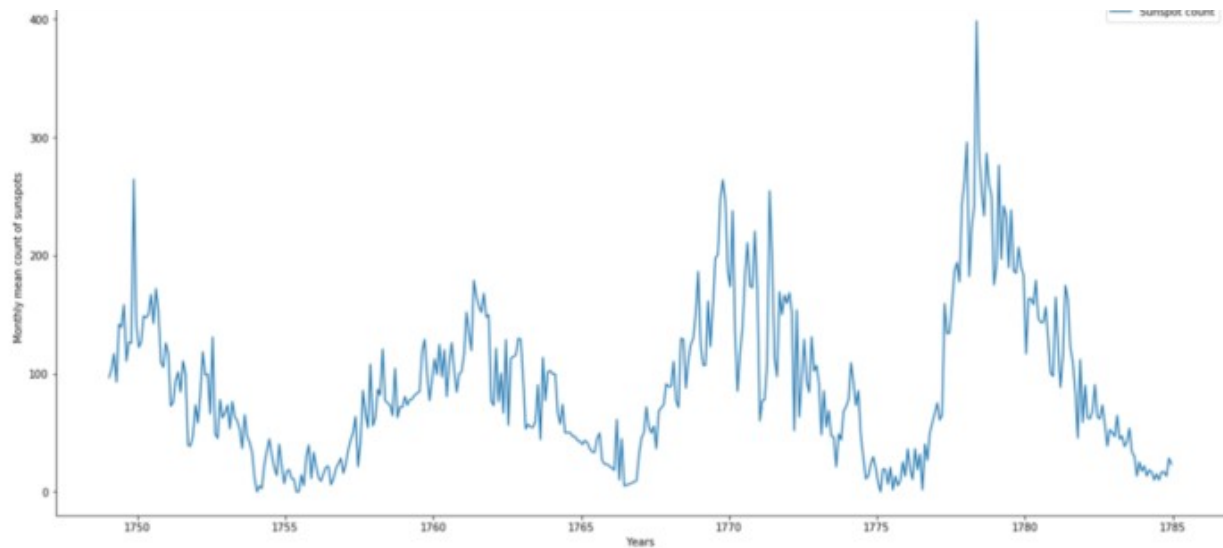
4.2- It seems like we could model this phenomenon with a gamma distribution, with a new cycle resetting every 12 years

A gamma distribution Γ is a two-parameter family of continuous probability distributions. The parameters are the shape a and the scale b. A random variable X that is gamma-distributed is noted $X \sim \Gamma(a, b)$, and in our case X is the count of sunspots. The two parameters a and b are the unknowns that we would like to calculate distributions for.
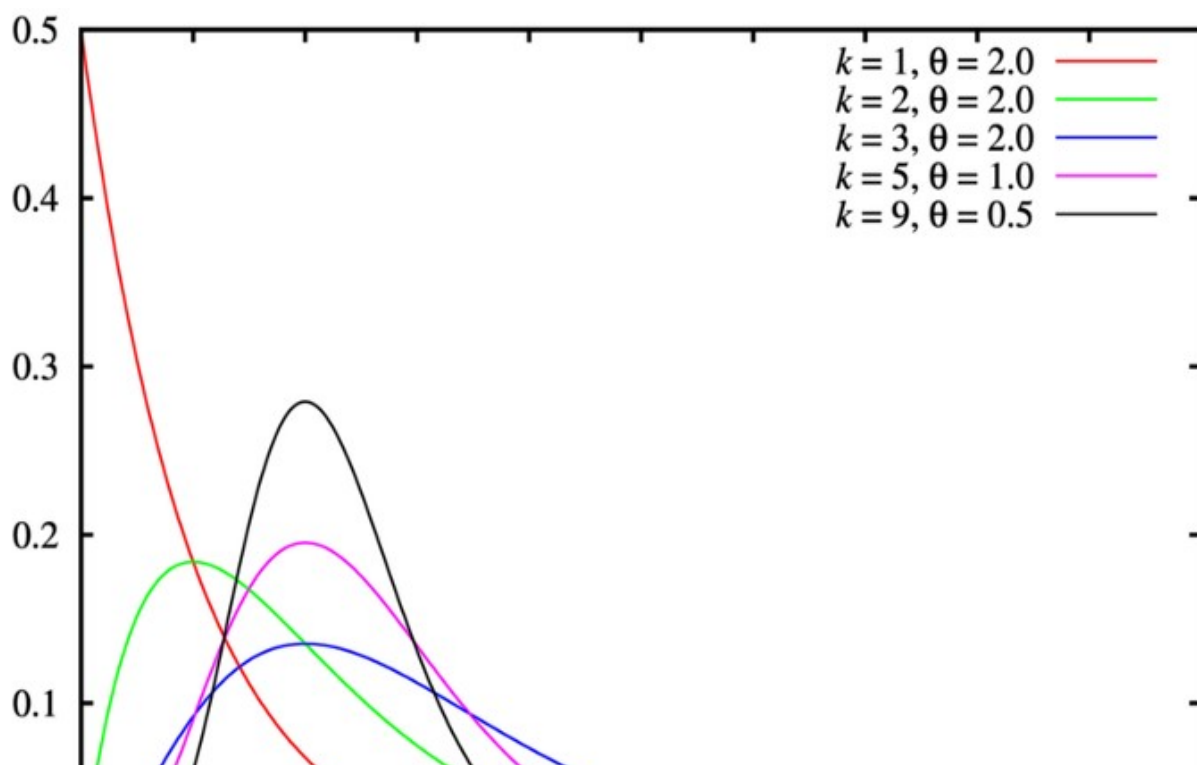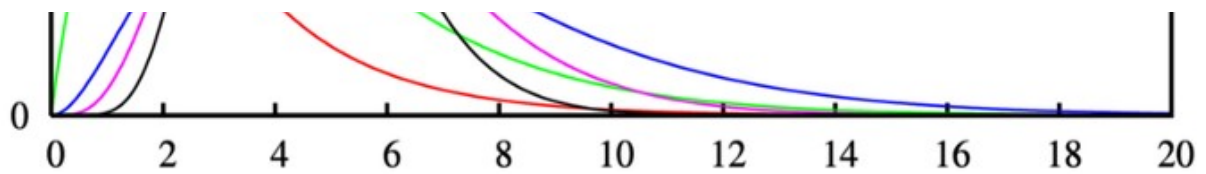
Gamma function with different parameters. Credit: Wikipedia Commons

For example, in the first cycle, the sunspot counts start from their highest at about 300 at the end of 1749, and fall to their lowest 6 years after, during 1755. Then the number rises up again to it's maximum during 1761 and 1762 before falling again during 1766 and so on. . .

Let's make sure by plotting a histogram of sunspot counts:

Figure 9: Histogram showing the frequency of sunspot counts over 270 years (1749-2018)
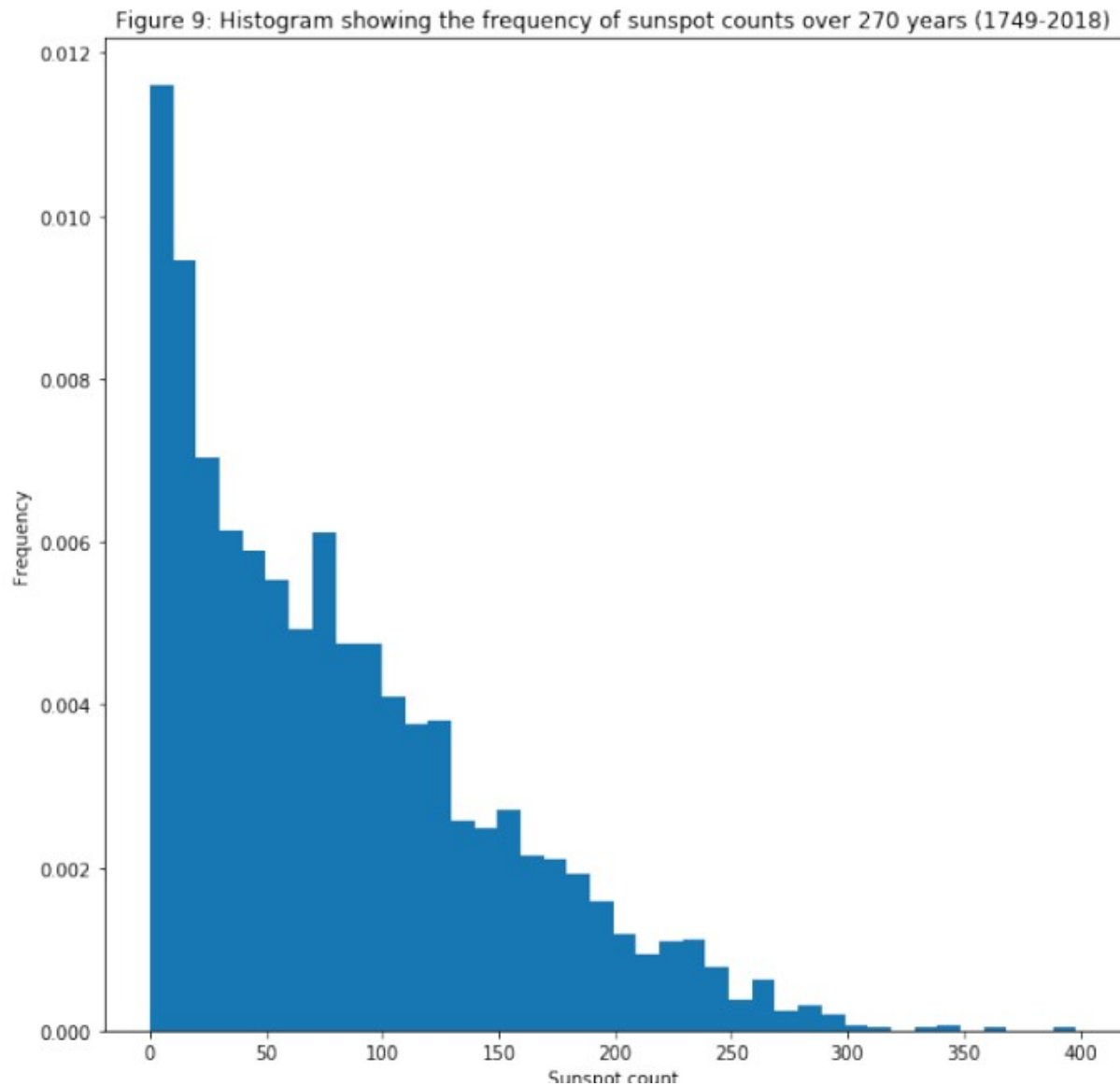
4.3- Indeed, it does seem like the frequency of counts follows a gamma distribution

The gamma distribution, has for PDF, f such that:

$$f(x; a, b) = \frac{b^a x^{a-1} e^{-bx}}{\Gamma(a)}$$

where Γ is the gamma function: $\Gamma(a) = (a-1)!$ (not to be confused with the gamma distribution!)

Following the same procedure as in the dummy data example, we can write down the log likelihood from this pdf (see code below). Alternatively, one could use the scipy.stats.gamma(a, b).pdf(x) function to compute it. However, beware that scipy's implementation is several orders of magnitudes slower than the one I implemented.

Since a and b must be strictly positive, we enforce this in the prior:

```
1    transition_model = lambda x: np.random.normal(x,[0.05,5],(2,))
2    import math
3    def prior(w):
4        if(w[0]<=0 or w[1] <=0):
5            return 0
6        else:
7            return 1
8
9    def manual_log_lik_gamma(x,data):
10       return np.sum((x[0]-1)*np.log(data) - (1/x[1])*data - x[0]*np.log(x[1]) - np.log(math.
11
12   def log_lik_gamma(x,data):
13       return np.sum(np.log(scipy.stats.gamma(a=x[0],scale=x[1],loc=0).pdf(data)))
```

Run the code and collect samples:

Starting from a=4, and b =10, the algorithm accepted 8561 pairs of samples, the last value for a is 0.98848982 and the last value for b is 84.99360422, which are pretty far off the initial values.

As with the dummy data example, let's see how the algorithm worked its way to these values:

Figure 10: MCMC sampling for a and b with Metropolis-Hastings. First 50 samples are shown.
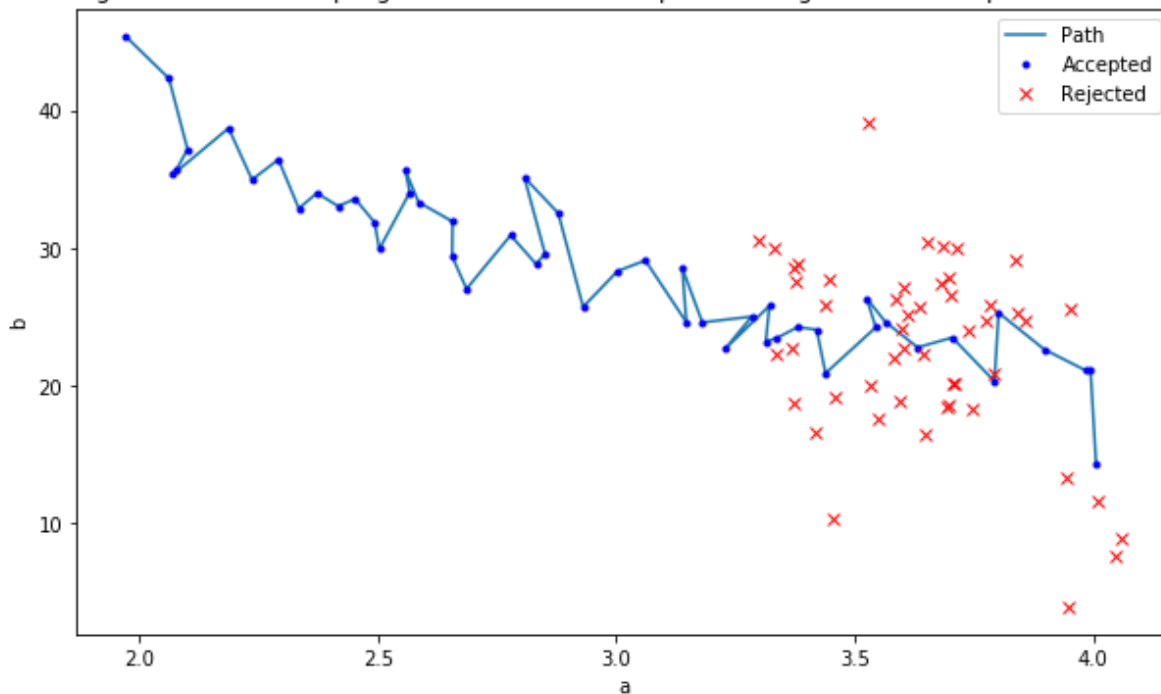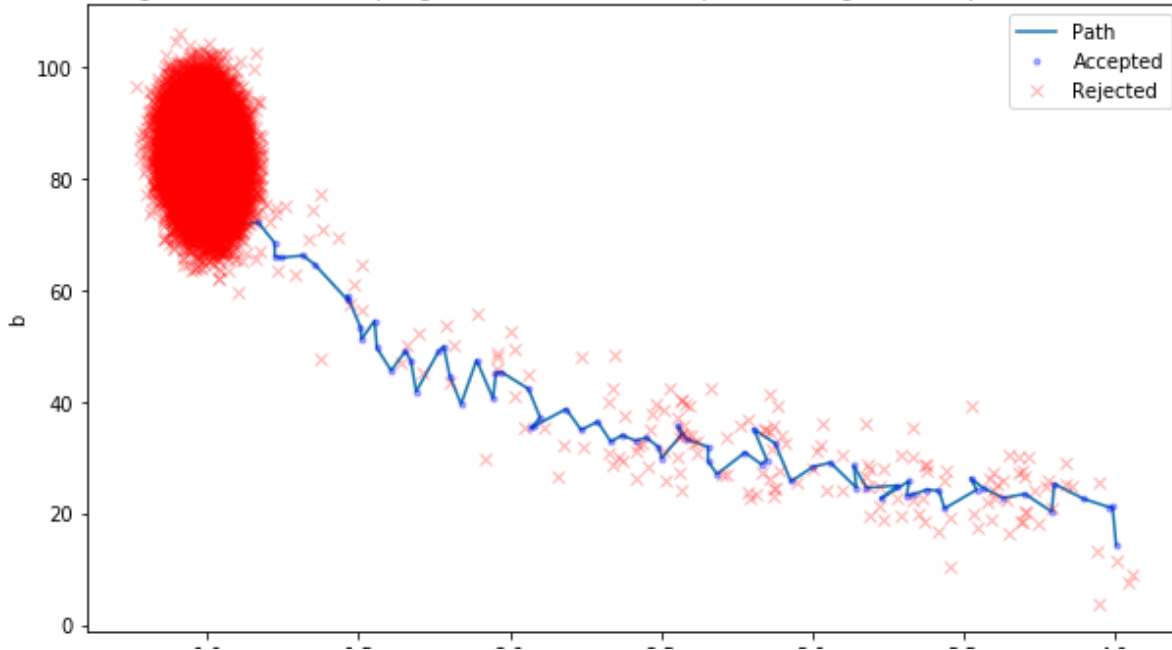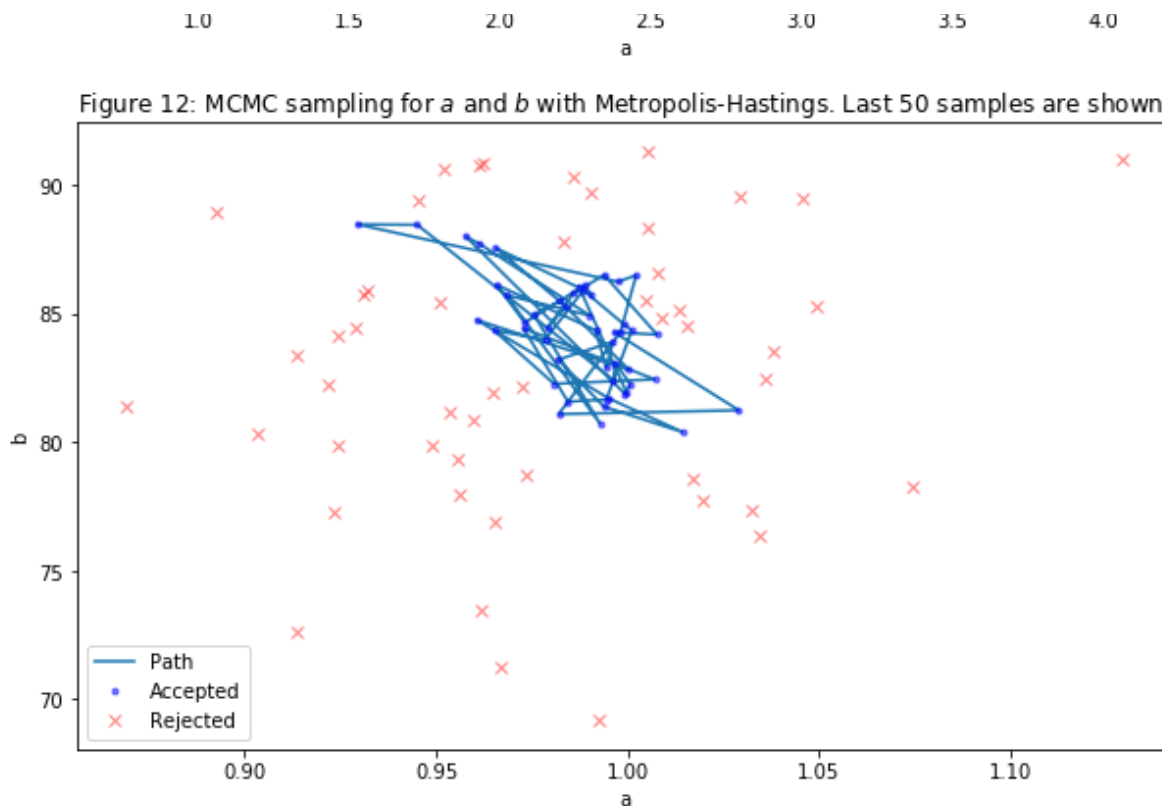
Figure 11: MCMC sampling for a and b with Metropolis-Hastings. All samples are shown.

Figure 12: MCMC sampling for a and b with Metropolis-Hastings. Last 50 samples are shown.

As we can see from figures 10, 11, and 12, the algorithm converges quickly to the [a=1,b=85] zone.

Tip: when the algorithm starts to heavily reject samples, that means that we have reached a zone of saturation of the likelihood. Commonly, this can be interpreted as having reached the optimal parameter space from which we can sample, i.e. there is very little reason for the algorithm to accept new values. This is marked in figures 11, and 12 where the algorithm no longer accepts any values outside of a small range.

4.3.1- We consider the initial 50% of the values of a and b to be "burn-in", so we drop them. Let's visualise the traces of and b and the histogram of the traces.
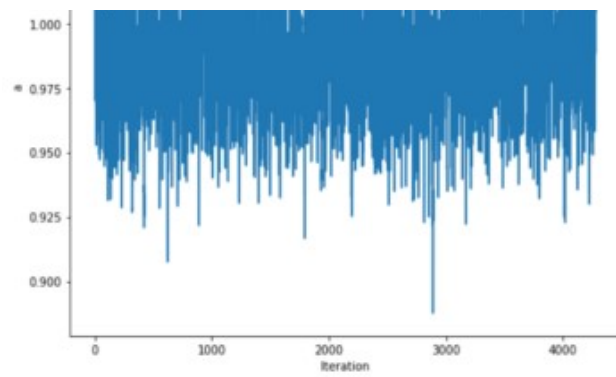

Figure 13: Trace for a
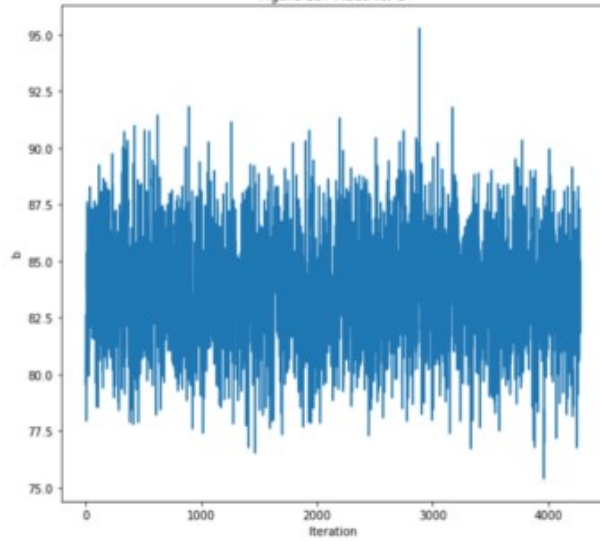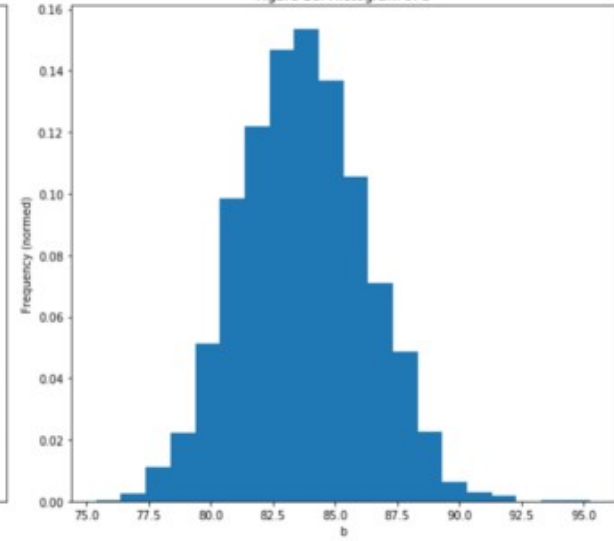
Figure 14: Histogram of a
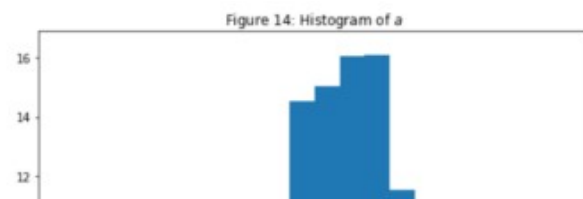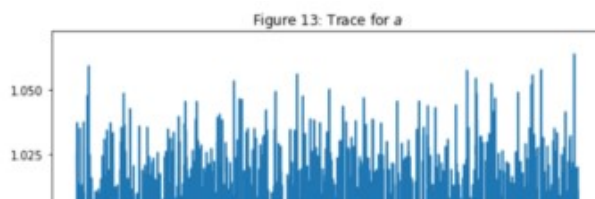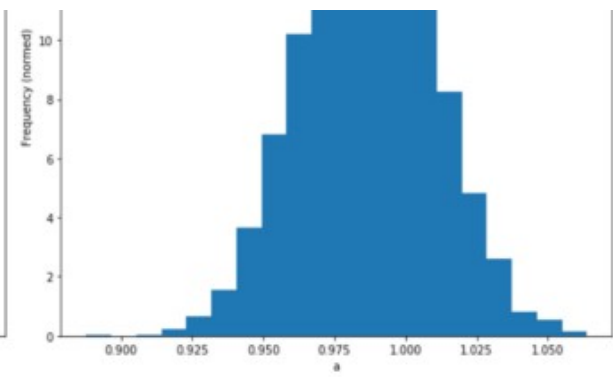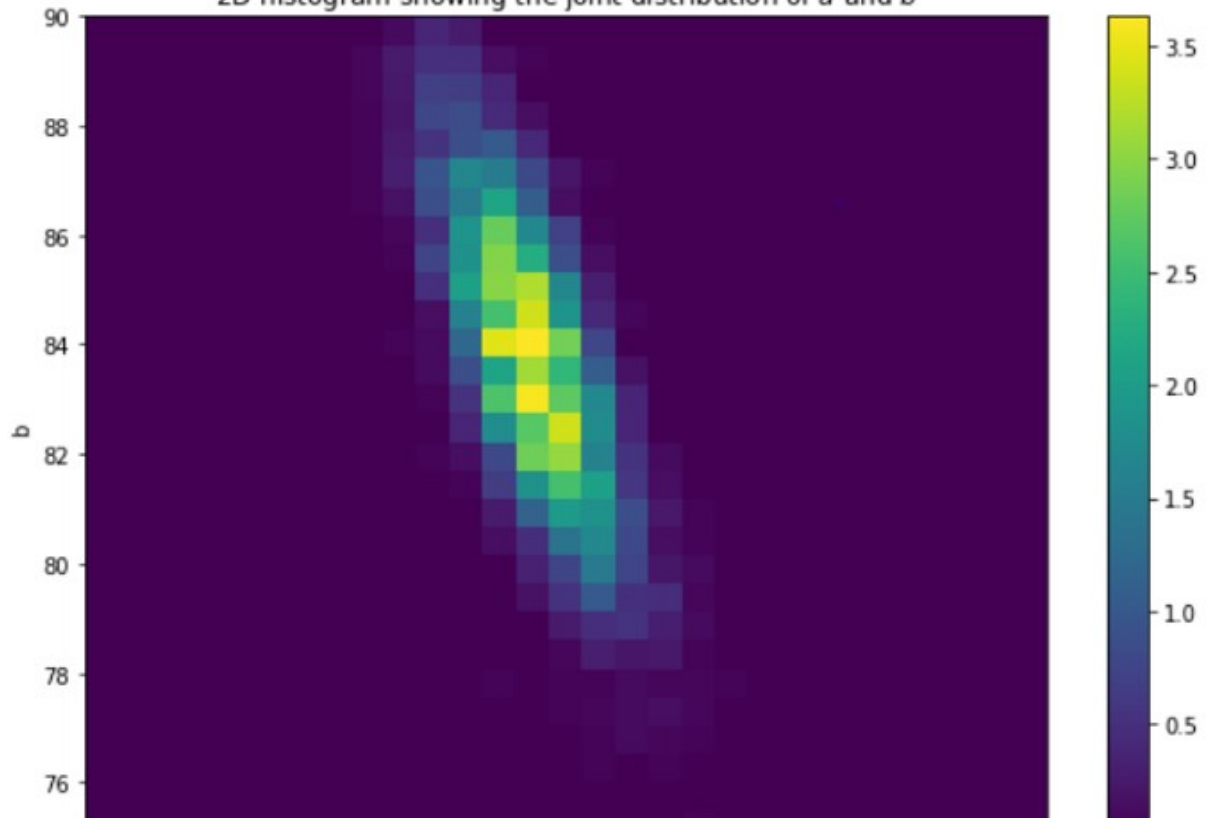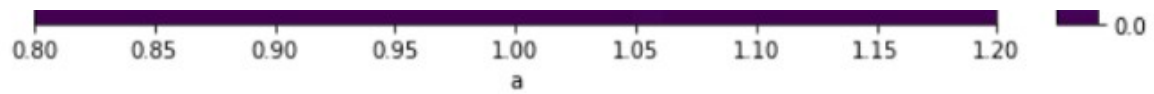
Figure 15: Trace for b

Figure 16: Histogram of b

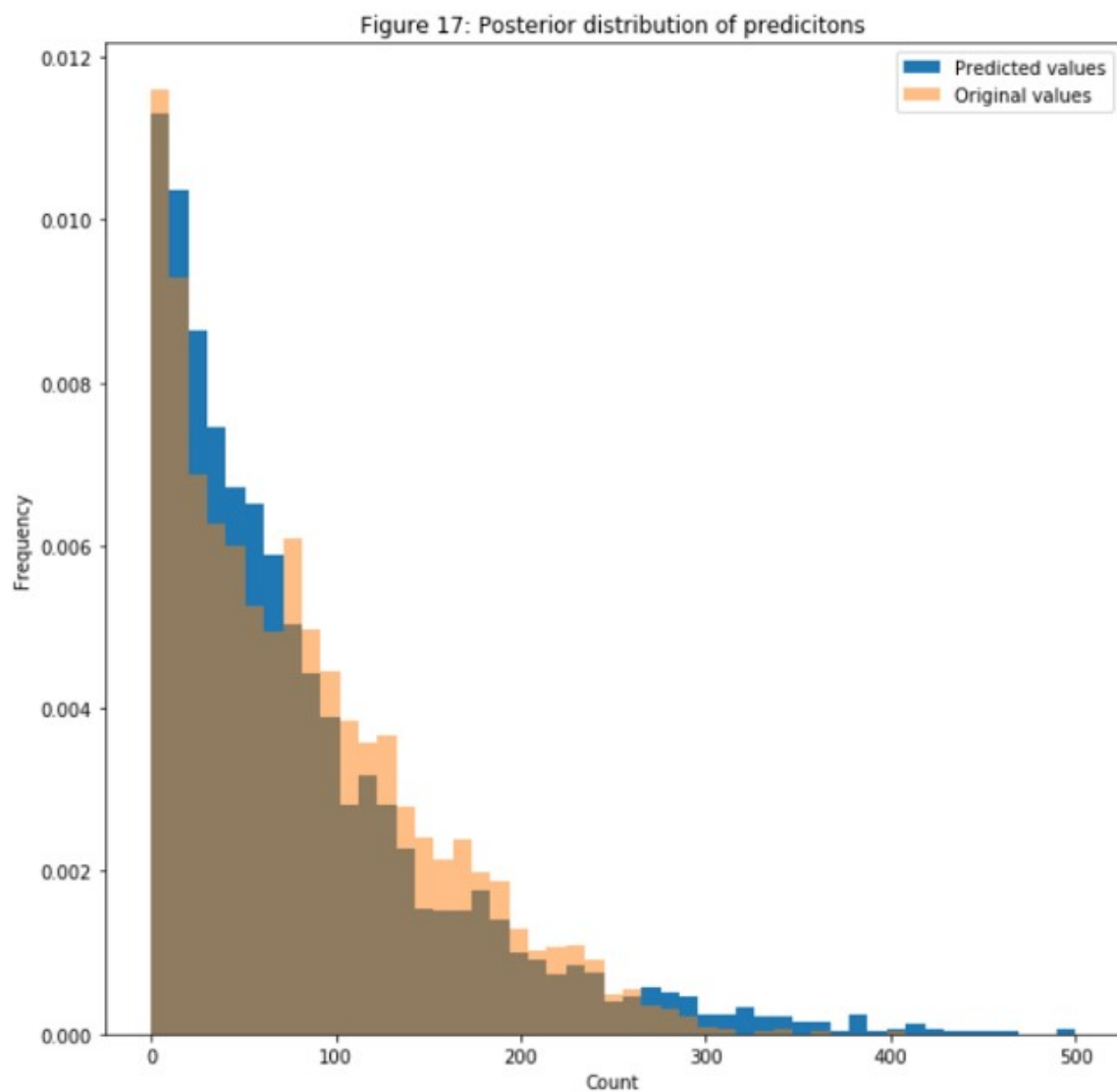2D histogram showing the joint distribution of a and b

4.4- Prediction time

First, we average the last 50% of accepted samples of a and b, and we generate random individuals from a Γ distribution. a{average}=0.9866200759935773 and b{average}=83.70749712447888.

And the predictions:

Figure 17: Posterior distribution of predicitons

# 4- Evaluation

4.1- Evaluation of the proposal distribution

How do we specify the parameters for the distribution Q? Should we move far from the current sample θ, or stay relatively close? These questions can be answered by measuring the auto-correlation between accepted samples. We don't want distant samples to be too correlated as we are trying to implement a Markov Chain, i.e. a sample should only depend on its previous sample, and the auto-correlation plot should show a quick, exponential decrease between the correlation of sample i and i-1,i-2,...i-k

The auto-correlation is defined by computing the following function for each lag k:

$$r_k = \frac{\sum_{i=1}^{N-k}(Y_i - Y_{avg})(Y_{i+k} - Y_{avg})}{\sum_{i=1}^{N}(Y_i - Y_{avg})^2}$$

The lag k, is basically the range ahead of sample Y_i in which we would like to measure the correlation.

The plots below show the auto-correlation for a, b for k going from 1 to 100. A lag of k=0 means that we are measuring the correlation of a sample with itself, so we expect it to be equal to 1. The higher k goes, the lower that correlation ought to be.

Figure 18: Auto-correlation for a and b, for k=1..100
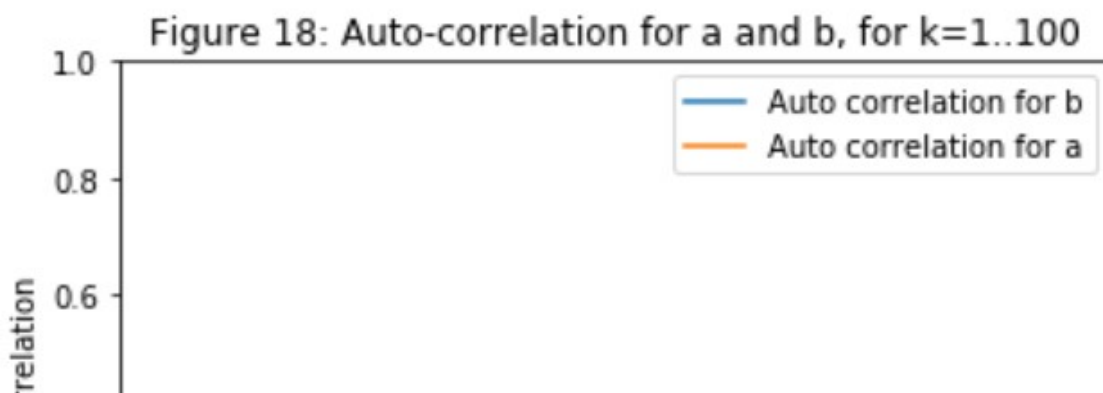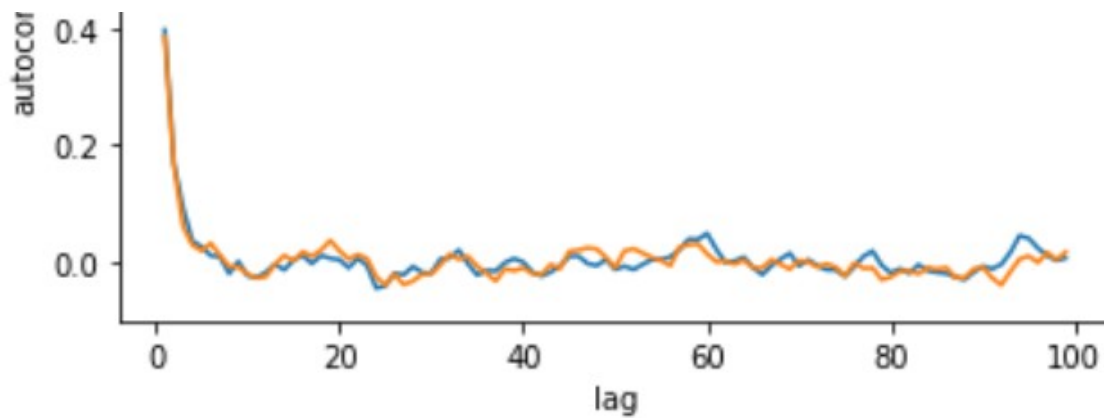
In our case, we are lucky to have a low enough correlation. In general, we might want to setup the parameters of the proposal distribution, Q, automatically. A common method is to keep adjusting the proposal parameters so that more than 50% proposals are rejected. Alternatively, one could use an enhanced version of MCMC called Hamiltonian Monte Carlo, which reduces the correlation between successive sampled states and reaches the stationary distribution quicker.

6- Conclusion

While the abstraction behind this algorithm may seem out of grasp at first, the implementation is actually pretty simple, and gives awesome results. In fact, the great thing about probabilistic programming, notably MCMC is that you only need to write down the model and then run it. There is no need to compute the evidence, or ensure some constraining mathematical properties.

I hope that everyone reading this article found it enjoyable and insightful. If there's positive feedback, there will be more articles of this series "From Scratch", where I explain and implement stuff from scratch (obviously!), so if you like it, please do suggest what you want me to talk about next!

Any questions are welcome, and I will do my best to answer them! Feedback is more than welcome as this is my first article and I wish to improve.