Markov Chain Monte Carlo Methods

Nikolas Tsigkas

May 2021

1 Gibbs Sampler for a Normal Model

The given dataset contains daily precipitation data, measured in $\frac{1}{100}$ of an inch, from Snoqualmie Falls, Washington from 1948 to 1983. The proposed model for the data is: $\ln(Y_1), .., \ln(Y_n)|\mu, \sigma \overset{iid}{\sim} N(\mu, \sigma)$, where both μ and σ are unknown in the Bayesian sense. They are in turn modelled as: $\mu \sim N(\mu_0, \tau_0)$ and $\sigma^2 \sim$ Scale-inv-$\chi^2(\nu_0, \sigma_0^2)$. The joint distribution of (μ, σ^2) can be sampled by the Gibbs Sampling Monte Carlo algorithm. The algorithm works as follows:

Algorithm 1: Gibbs Sampling

initialise the parameters $\theta_1^{(0)}, .., \theta_p^{(0)}$;
for *i = 1,..,N* **do**
  for *j = 1,..,p* **do**
    Draw $\theta_j^{(i)}$ from the conditional posterior $p(\theta_j|\theta_1^i, .., \theta_{j-1}^{(i)}, \theta_{j+1}^{(i-1)}, .., \theta_p^{(i-1)})$
  end
end
Result: $(\theta_1^{(1)}, .., \theta_p^{(1)}), .., (\theta_1^{(N)}, .., \theta_p^{(N)})$

The Gibbs Sampling method is a convenient method for when the full conditional distributions are known, which is the case here, as it can be shown that:

$$
\begin{aligned}
w &= \frac{n/\sigma^2}{n/\sigma^2 + 1/\tau_0^2} \\
\mu_n &= w\bar{y} + (1-w)\mu_0 \\
\frac{1}{\tau_n^2} &= \frac{1}{n/\sigma^2 + 1/\tau_0^2} \\
\mu|\sigma, \mathbf{y} &\sim N(\mu_n, \tau_n) \\
\sigma^2|\mu, \mathbf{y} &\sim \text{Scale-Inv-}\chi^2(\nu_0 + n, \frac{\nu_0\sigma_0^2 + \sum_{i=1}^n (y_i - \mu)^2}{\nu_0 + n})
\end{aligned}
$$

The algorithm was performed using the prior parameters: $\mu_0 = 3.7$, $\tau_0 = 100$, $\nu_0 = 1000$ and $\sigma_0 = 1$, as well as using the initial values $\mu^{(0)} = 1$, $\sigma^{(0)} = 1$. The resulting scatter plot of 5000 random draws from the Gibbs-sampler are shown in figure 1.

One possible problem of the Gibbs sampler, as with all Markov Chain Monte Carlo methods, is that the samples can be heavily autocorrelated, resulting in a slower convergence towards the mean. This is due to the fact that the variance of the sample mean of a parameter θ becomes: $\mathbf{V}[\frac{1}{N}\sum_{i=1}^N \theta^{(i)}] = \frac{\sigma^2}{N}[1 + 2\sum_{i=1}^N \text{acf}(i)]$, where acf$(i)$ denotes the autocorrelation at lag i. The term inside the parenthesis is often referred to as the inefficiency factor (IF), and can give a good description of how fast the simulations converges. From the simulations, this quantity was estimated to be IF = 5.168.

Another way to examine the convergence is visually, by plotting the Markov Chain trajectories. Figure 2 shows that the algorithm draws values from what appears to be a stationary distribution.

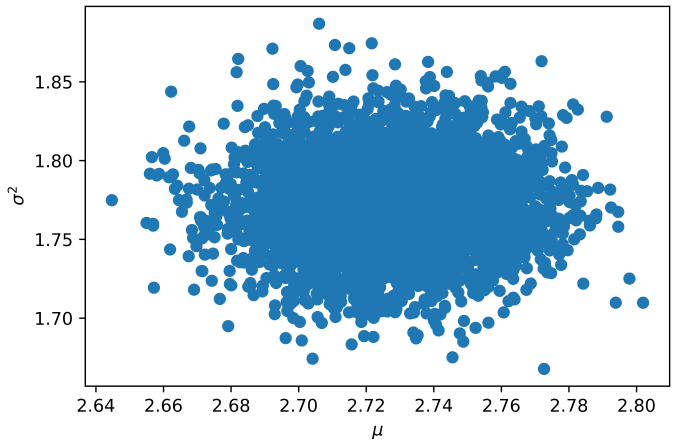

Figure 1: Scatter plot over the simulated μ and σ^2 values

Finally, the posterior predictive density can be evaluated using the Gibbs draws as:

$$p(y_{n+1}|\mathbf{y}) = \iint_{\Theta} p(y_{n+1}|\mathbf{y},\theta)p(\theta|\mathbf{y}) = \mathbf{E}_{\theta|\mathbf{y}}[p(y_{n+1}|\theta)] \approx \frac{1}{N}\sum_{1}^{N} p(y_{n+1}|\theta^{(i)})$$

Where Θ is the domain space of the parameters, N is the number of Gibbs draws and $\theta^{(i)}$ is the vector of Gibbs draws, i.e. $(\mu^{(i)}, (\sigma^2)^{(i)})$, at iteration i. By using the Gibbs draws, one can then draw from the posterior predictive density by drawing random samples from a $N(\mu^{(i)}, \sigma^{(i)})$ distribution. This resulting distribution can then be compared to the observed data. Figure 3 shows the log of the observed data in a histogram together with the posterior predictive draws. The results are not too promising, as there is quite some deviation between the data and the pdf. For better results, one could examine for instance normal mixture distributions, with which could potentially capture the skewness which is apparent in the plot. Drawing from mixture densities is also fairly convenient with the Gibbs sampling algorithm.

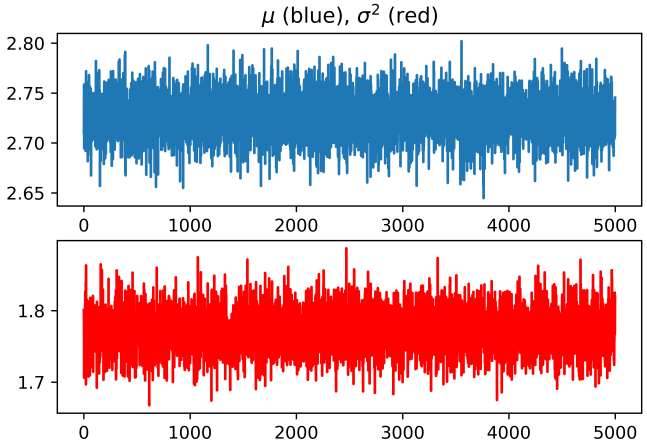

Figure 2: The Markov Chain trajectories for the sampled parameters

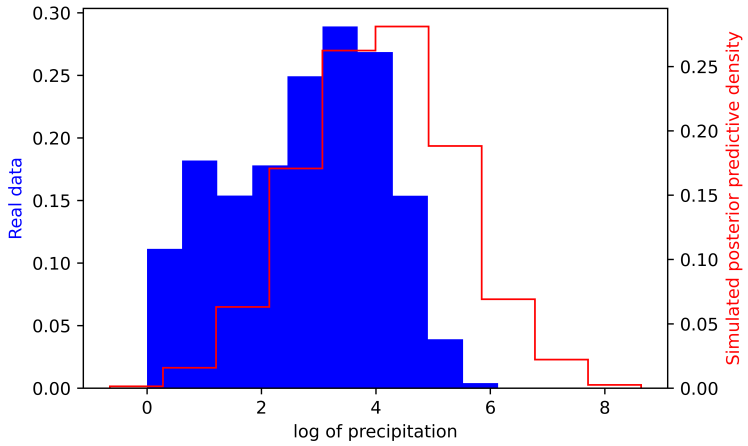

Figure 3: The real data compared to the estimated posterior predictive density

```
""" Some useful packages"""
import pandas as pd
import numpy as np
from scipy import stats as st
import matplotlib.pyplot as plt

np.random.seed(1111) # set seed for reproducability

""" Import and transform the data """
data = pd.read_csv('rainfall.dat', header = None)
log_y = np.log(data.values)

""" Helper functions"""
# Draws 1 sample from the Scale-Inverse-chi2 distribution
def draw_chi2inv(df, scale):
    x = st.chi2.rvs(df-1, scale, size=1)
    sigma_sq = (df-1)*scale/x
    return sigma_sq

# Returns the parameters for the posterior of mu
def mu_post_params(pri_params, sigma, data):
    [mu_0, tau_0, nu_0, sigma_0] = pri_params
    n = len(data)
    w = (n/sigma**2)/(n/sigma**2 + tau_0**(-2))
    mu_n = w*np.mean(data)+(1-w)*mu_0
    tau_n = (1/(n/sigma**2 + 1/tau_0**2))**0.5
    return [mu_n, tau_n] # Mean and sd for the normal distribution

# Returns the parameters for the posterior of sigma^2
def sigma_post_params(pri_params, mu, data):
    [mu_0, tau_0, nu_0, sigma_0] = pri_params
    n = len(data)
    nu_n = nu_0 + n
    scale = (nu_0*sigma_0**2 + np.sum((data-mu)**2) )/(nu_n)
```

```
35      return [nu_n, scale] #DoF and scale parameter for the Scale-Inv-chi2
36
37  # Draws nDraws Gibbs-samples from the posterior
38  def Gibbs(pri_params, post_params1, post_params2, dist1, dist2, iVals, nDraws,
        data):
39      var1 = np.zeros(nDraws+1)
40      var2 = np.zeros_like(var1)
41      # Initialize the Gibbs algorithm with the initial values
42      [var1[0], var2[0]] = iVals
43      # Sample iteratively from the conditional marginal distributions
44      for i in range(1,nDraws+1):
45          [mu_n, tau_n] = post_params1(pri_params, var2[i-1], data)
46          var1[i] = dist1(mu_n, tau_n)
47          [nu_n, scale] = post_params2(pri_params, var1[i],   data)
48          var2[i] = dist2(nu_n, scale)
49      return [var1[1:], var2[1:]]
50
51  # Sample autocorrelation
52  def acf(x):
53      n = len(x)
54      autocorr = []
55      for i in range(1, len(x)-1):
56          autocorr.append(np.corrcoef(x[:n-i],x[i:])[0,1])
57      return np.array(autocorr)
58
59  """ Prior parameters """
60  # pri_params = mu_0, tau_0, nu_0, sigma_0
61  pri_params = np.array([3.7, 100.0, 1000.0, 1.0])
62  iVals = np.ones(2)
63
64  [sample_mu, sample_sigma2] = Gibbs(pri_params,
65                                mu_post_params, sigma_post_params,
66                                st.norm.rvs, draw_chi2inv,
67                                iVals,
68                                5000,
69                                log_y
70                                )
71
72  # Calculate the inefficiency factor
73  autocorr = acf(sample_sigma2)
74  IF = 1 + 2*np.sum(autocorr)
75
76  # Scatterplot of the simulated draws
77  plt.scatter(sample_mu, sample_sigma2)
78  plt.xlabel(r"$\mu$")
79  plt.ylabel(r"$\sigma^2$")
80  plt.show
81
82  # Trajectories of the two Markov chains
83  fig, axs = plt.subplots(2,1)
84  axs[0].plot(sample_mu)
85  axs[1].plot(sample_sigma2, color = "r")
86
87  axs[0].set_title(r"$\mu$ (blue), $\sigma^2$ (red)")
```

```
plt.savefig("Gibbs_Convergence.png", dpi = 800)
plt.show

# Compare the empirical data to the posterior predictive distribution
# from the MCMC simulation
y_pred = np.zeros((5000,1))

for i in range(len(y_pred)):
    y_pred[i] = sample_mu[i] + np.sqrt(sample_sigma2[i])*st.norm.rvs(1)

fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
ax1.hist(log_y, density = True, color = "b")
ax2.hist(y_pred, color="r", histtype = "step", density = True)

ax1.set_xlabel('log of precipitation')
ax1.set_ylabel('Real data', color='b')
ax2.set_ylabel('Simulated posterior predictive density', color='r')
plt.show()
```

2 Random Walk Metropolis for Poisson Regression

The given dataset contains data on from coin auctions on eBay. The goal is to fit a model that explains the number of bids that an auction receives, given the covariates:

- If the seller is a "Power Seller", meaning they put up a high volume of items for auction (0,1)
- If the seller is verified by eBay (0,1)
- If the sold coin was packaged in an unopened envelope (0,1)
- If the item has a minor or major defect (0,1 for each type of defect)
- If the seller has recorded a lot of negative feedback (0,1)
- The log of the book value of the coin, as deemed by experts
- The ratio between the minimal bidding price and the book value

Formally, the model is:

$$Y_i|\mathbf{x}_i, \beta \sim \text{Po}[\exp(\mathbf{x}_i^T \beta)]$$

Where y_i is the recorded number of bids, and $\mathbf{x}_i$ is a vector containing the covariates of the auction, with an intercept term added. Initially, the maximum likelihood estimate of the regression coefficients β can be explored, for instance using the STATSMODEL package in Python. The results from this are shown in the code snippet below.

```
print(poisson_fit.summary())
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 1000
Model:                            GLM   Df Residuals:                      991
Model Family:                 Poisson   Df Model:                            8
Link Function:                    log   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -1796.1
Date:                Thu, 13 May 2021   Deviance:                       867.47
Time:                        19:00:29   Pearson chi2:                     752.
No. Iterations:                     5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          1.0724      0.031     34.848      0.000       1.012       1.133
x1            -0.0205      0.037     -0.558      0.577      -0.093       0.052
x2            -0.3945      0.092     -4.268      0.000      -0.576      -0.213
x3             0.4438      0.051      8.778      0.000       0.345       0.543
x4            -0.0522      0.060     -0.867      0.386      -0.170       0.066
x5            -0.2209      0.091     -2.416      0.016      -0.400      -0.042
x6             0.0707      0.056      1.255      0.210      -0.040       0.181
x7            -0.1207      0.029     -4.166      0.000      -0.177      -0.064
x8            -1.8941      0.071    -26.588      0.000      -2.034      -1.754
==============================================================================
```

One can see that the variables x_1, x_4 and x_6, corresponding to the "Power seller", minor damage and negative feedback parameters, are not statistically significant on a 5% level from the resulting p-values.

Furthermore, one can also draw simulated values for these parameters using for instance the Metropolis Random Walk algorithm. The algorithm works by performing the following steps:

Algorithm 2: Random Walk Metropolis

for *i = 1,..,N* **do**
  Draw a sample proposal $\theta^{(p)}$ from the conditional probability $\Theta^{(p)}|\theta^{(i-1)} \sim N(\theta^{(i-1)}, c\Sigma)$;
  Compute the acceptance probability $\alpha_i = \min(1, \frac{p[\theta^{(p)}|\mathbf{y}]}{p[\theta^{(i-1)}|\mathbf{y}]})$;
  Draw a sample x_i from $X_i \sim \text{Bern}(\alpha_i)$;
  if $x_i = 1$ **then**
    $\theta^{(i)} \leftarrow \theta^{(p)}$
  else
    $\theta^{(i)} \leftarrow \theta^{(i-1)}$
  end
end

Two possible choices for the Σ parameter is the identity matrix, or the negative inverse hessian of the log-posterior, evaluated at the posterior mode. The latter was found through numerical optimisation of the log-posterior pdf, using the Broyden-Fletcher-Goldfarb-Shanno algorithm. The prior used for the β parameters was the so called Zellner g-prior:

$$\beta \sim N(0, 100[\mathbf{X}^T\mathbf{X}]^{-1})$$

This gives a log posterior proportional to:

$$\ln(p(\beta|\mathbf{y})) \propto \sum_{i=1}^{n} \ln[p(y_i|\beta)] + \ln[p(\beta)] = \sum_{i=1}^{n}[-\lambda_i + y_i \ln(\lambda_i) - \sum_{j=1}^{y_i} \ln(j)] + \phi_{\mathbf{0},100\mathbf{X}^T\mathbf{X}^{-1}}(\beta)$$

Where $\lambda_i = \exp(\mathbf{x}_i^T\beta)$ and $\phi_{\mu,\mathbf{\Sigma}}$ is the pdf of the normal distribution with mean vector and covariance matrix μ and $\mathbf{\Sigma}$ respectively. It can be noted that, for the optimisation, the inner sum in the first term is redundant and can thus be omitted, as it is independent of β. The resulting posterior mode and negative inverse hessian of the log posterior was:

$$\tilde{\beta} = \begin{bmatrix} 1.0698 & -0.0205 & -0.3930 & 0.4436 & -0.0525 & -0.2212 & 0.0707 & -0.1202 & -1.8920 \end{bmatrix}^T$$

$$-H^{-1}_{\ln p}(\tilde{\beta}) = \begin{bmatrix} 0.0011 & -0.0007 & 0 & -0.0004 & -0.0003 & 0.0002 & -0.0004 & 0 & 0.0012 \\ -0.0007 & 0.0013 & -0.0002 & -0.0002 & 0.0006 & 0.0008 & 0.0004 & 0 & -0.0002 \\ 0 & -0.0002 & 0.0027 & 0 & 0.0007 & 0.0024 & 0.0007 & -0.0014 & -0.0009 \\ -0.0004 & -0.0002 & 0 & 0.0021 & -0.0005 & -0.0008 & -0.0002 & 0 & -0.0008 \\ -0.0003 & 0.0006 & 0.0008 & -0.0005 & 0.0035 & 0.0038 & 0.0002 & -0.0004 & -0.0007 \\ 0.0002 & 0.0008 & 0.0024 & -0.0008 & 0.0038 & 0.0223 & 0.0027 & -0.0012 & 0.0010 \\ -0.0004 & 0.0004 & 0.0007 & -0.0002 & 0.0002 & 0.0027 & 0.0029 & -0.0004 & -0.0003 \\ 0 & 0 & -0.0014 & 0 & -0.0004 & -0.0012 & -0.0004 & 0.0007 & 0.0005 \\ 0.0012 & -0.0002 & -0.0009 & -0.0008 & -0.0007 & 0.0010 & -0.0003 & 0.0005 & 0.0034 \end{bmatrix}$$

Furthermore, the ratio of posterior probabilities, needed to determine α in the Metropolis algorithm can be calculated using the log-posterior, as: $\exp[\ln(p(\beta^i|\mathbf{y})) - \ln(p(\beta^{i-1}|\mathbf{y}))]$, where once again the terms independent with regards to β can be neglected. This approach is also more numerically sound, as the originally defined ratio can lead to division by very small numbers.

The final terms needed to perform the Monte Carlo algorithm are the initial value of β, which was set to the zero-vector, and the constant c. This parameter can be used to tune the sampling to get satisfactory results. One rule of thumb is to have an average acceptance probability within the region of $25 - 30\%$. With $c = 0.6$, the average acceptance was 27.1%. The resulting marginal trajectories for each element in the β-vector can also be examined to see whether the algorithm has converged. Figure 4 shows that the samples of all the elements follow a stationary process after around 500 iterations.

From these random draws, the sample mean can be evaluated. By eliminating the fist 500 random draws, the so called "burn-in" period, the sample mean was estimated to:

$$\hat{\beta} = \begin{bmatrix} 1.0612 & -0.0097 & 0.0512 & 0.3736 & -0.0517 & -0.2173 & 0.057 & -0.0256 & -1.770 \end{bmatrix}^T$$

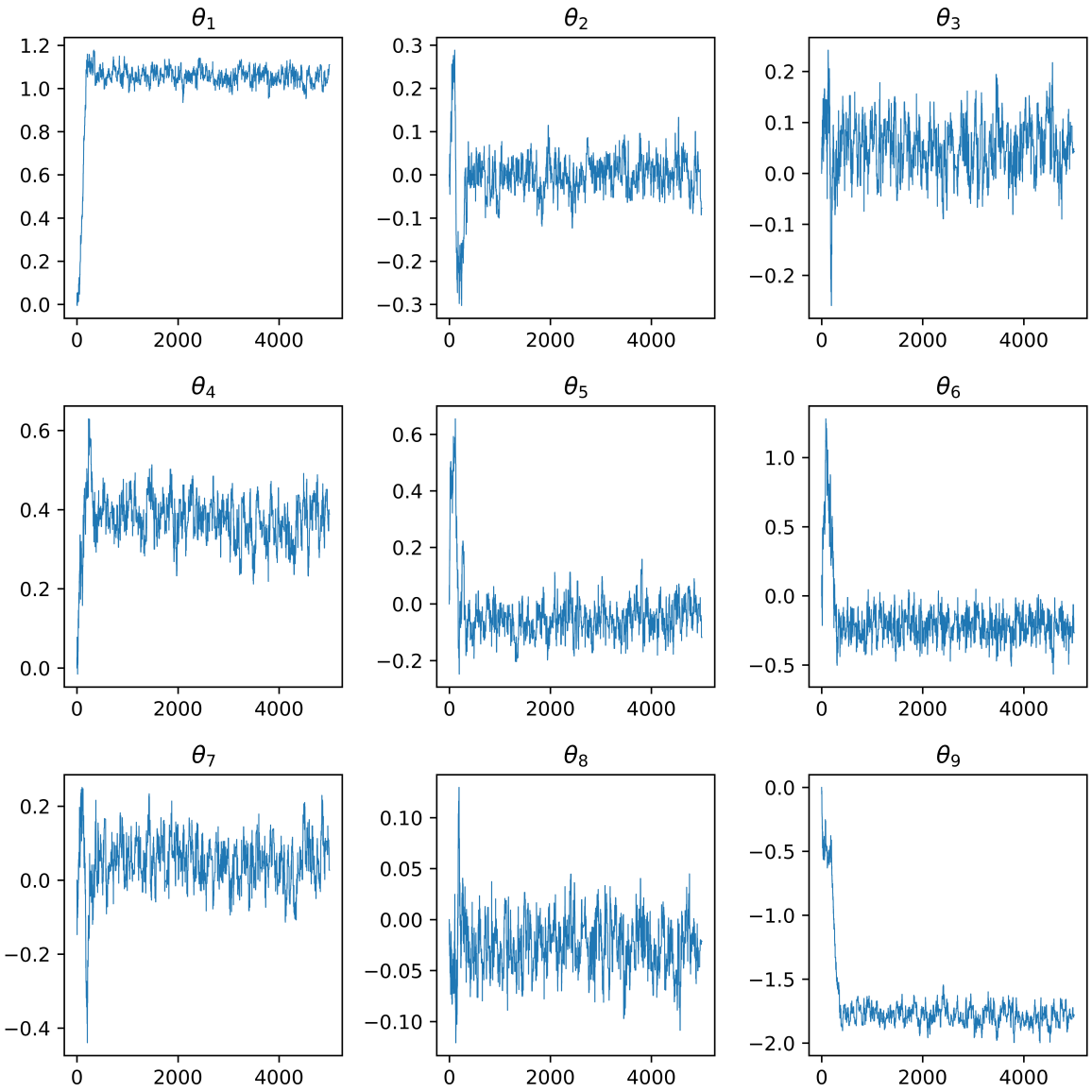

Figure 4: Trajectories of the regression parameters for 5000 simulations

Using the draws from the stationary distribution, one can examine the distribution of a new, unobserved sample. For instance the posterior predictive distribution of an auction with:

- A power seller with a verified ID
- A sealed product with a major defect, but no minor one
- No major negative feedback of the bidder
- A log book value of 1
- A minimum bid to book value of 0.7

Which corresponds to the data vector:

$$\mathbf{x}_s = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0.7 \end{bmatrix}^T$$

Can be evaluated using the same procedure as with the Gibbs sampling algorithm, i.e. drawing random values from a $Po[\exp(\mathbf{x}_s^T \beta^{(i)})]$ distribution for the different $\beta^{(i)}$ draws produced from the Metropolis Random Walk. The histogram

from these draws, together with the pmf corresponding to the maximum likelihood estimate of the statsmodel package is shown in figure 5. From these draws, the probability that this auction will have no bids, $P[Y_s = 0|\beta]$ was also evaluated to be approximately 18.69%

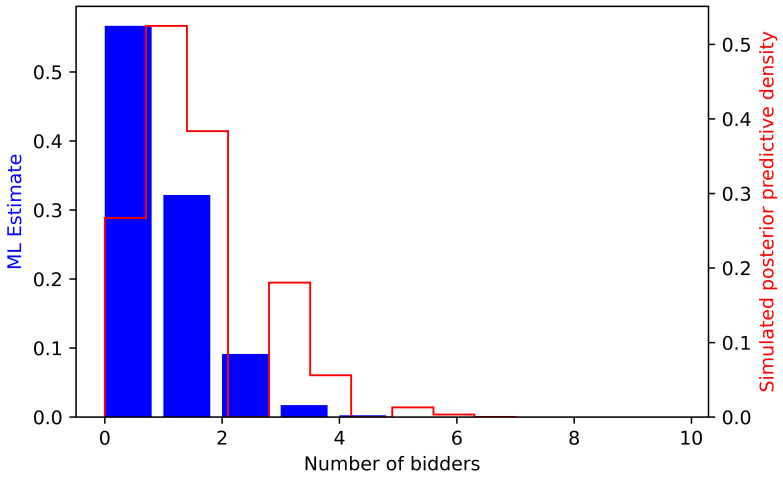

Figure 5: The Maximum Likelihood estimate compared to the posterior predictive density

```python
""" Some useful packages"""
import pandas as pd
import numpy as np
from scipy import stats as st
import matplotlib.pyplot as plt
import statsmodels.api as sm
from scipy.optimize import minimize as fmin

np.random.seed(1111) # set seed for reproducability

""" Helper functions """
# An expression proportional to the log of the posterior, minus the terms
# that are not dependent on beta,
# to be used for the numerical optimization and the MRW
def log_posterior(beta, X, y):
    regr = X@beta
    logL = np.sum(-np.exp(regr) + y*regr)
    prior = np.log(st.multivariate_normal.pdf(beta, mean=np.zeros_like(beta), cov =
        100*np.linalg.inv(X.T@X)))
    return (logL + prior)

class RWM_res:
    def __init__(self, init, samples):
        self.initial = init
        self.iterations = samples
        self.theta = np.zeros((len(init), samples+1))
        self.theta[:,0] = self.initial # Initialize
        self.acc_p = 0.0 # Keep track of the acceptance probability
    def accept(self, theta_p, it):
        self.acc_p += 1.0/(self.theta.shape[1])
        self.theta[:,it] = theta_p
    def reject(self, it):
        self.theta[:,it] = self.theta[:,it-1]
    def draw(self, it, Cov):
        return st.multivariate_normal.rvs(mean=self.theta[:,it-1], cov=Cov)
    def hat(self, burn_in):
        return np.mean(self.theta[:,burn_in:], axis=1)
    def plot(self, x, y):
        order = np.reshape(np.arange(0,x*y),(x,y))
        fig, axs = plt.subplots(x,y, figsize = (8,8))
        for i in range(x):
            for j in range(y):
                axs[i,j].plot(self.theta[order[i,j],:], linewidth=0.5)
                axs[i,j].set_title(r"$\theta_%i$" %(order[i,j]+1))
            fig.tight_layout()
            plt.show
    def iterate(self, log_post_fun, X, y, Cov, c):
        for i in range(1,self.iterations+1):
            theta_p = self.draw(i, c*Cov)
            # Calculate the probability of acceptance
            post_ratio = np.exp(log_post_fun(theta_p, X, y) - log_post_fun(self.theta
                [:,i-1], X, y))
            # Draw a random sample from a Bern(alfa) distribution
            # To determine if the iteration should be accepted or not
            alfa = np.nanmin([1, post_ratio])
            if st.bernoulli.rvs(alfa):
                self.accept(theta_p, i)
            else:
```

```
                self.reject(i)

""" Import and transform the data """
data = pd.read_csv('eBayNumberOfBidderData.csv', sep = ";")

y = data.nBids.values # Response variable
X = data.iloc[:, 1:].values # Covariates

"""Fit a generalized linear model using the statsmodel package"""
exog, endog = X, y
model = sm.GLM(endog, exog,
               family = sm.families.Poisson(link=sm.families.links.log))

poisson_fit = model.fit()
print(poisson_fit.summary())

"""Find the normal approximation of the posterior, to get the inverse hessian"""
# Define the log posterior as a function of beta only, for the optimization
# Also, multiply with -1, as we seek the maximum
# and the algorithm performs minimization
def obj_fun(beta):
    return -1*log_posterior(beta, X, y)

# Find the max of the log posterior using a quasi-Newton method
# Which also returns a numerical approximation of H^-1
x0 = np.zeros(9) # Initial guess
opt_res = fmin(obj_fun, x0, method="BFGS", options={"disp" : True})

# Numerical approximation of -H^-1
H_inv = opt_res["hess_inv"]
# The mode, i.e. the beta vector at the optimum
post_mode = opt_res['x']

"""Draw from the actual posterior,
using the Random Walk Metropolis algorithm"""
RWM_results = RWM_res(np.zeros(9),5000) # Initialize
RWM_results.iterate(log_posterior, X, y, H_inv, 0.6)

RWM_results.acc_p # Check the average acceptance probability

# Plot the marginal trajectories to check convergence visually
RWM_results.plot(3,3)
plt.savefig("MCMC_convergence.png", dpi=800)

# Compute the sample mean, eliminating the first 500 iterations
beta_hat = RWM_results.hat(500)

# Using the iterations to sample from the posterior predictive density
beta_draws = RWM_results.theta[:,501:]

# For the following auction:
"""
The x-vector for an auction with:
    A power seller,
    with verified ID.
    A sealed product with
    a major defect
    No major negative feedback
    A log book-value of 1
```

```
    A minimum bid of 0.7
"""
x_sample = np.array([1,1,1,1,0,1,0,1,0.7])

y_pred = np.zeros(4500)
for i in range(len(y_pred)):
    lamda = np.exp(x_sample@beta_draws[:,i])
    y_pred[i] = st.poisson.rvs(1, lamda)

# The posterior distribution of this sample
lamda_MLE = np.exp(x_sample@poisson_fit.params)
y_MLE = st.poisson.pmf(np.arange(0,10), lamda_MLE)

fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
ax1.bar(np.arange(0,10), y_MLE, color = "b", align = "edge")
ax2.hist(y_pred, color="r", histtype = "step", density = True, align = "mid")

ax1.set_xlabel('Number of bidders')
ax1.set_ylabel('ML Estimate', color='b')
ax2.set_ylabel('Simulated posterior predictive density', color='r')
plt.show()

# The probability that the sample gets no bids
p0 = np.sum(y_pred == 0)/len(y_pred)
```

3 Hamiltonian Monte Carlo for Time Series Analysis

The aim is to use Hamiltonian Monte Carlo, through the PyStan package, to evaluate parameters of an $AR(1)$ process $\{X_t\}_{t=0}^T$, defined in the following way:

$$X_t = \mu + \phi(X_{t-1} - \mu) + \epsilon_t \quad \epsilon_t \overset{iid}{\sim} N(0, \sigma) \quad X_0 = \mu$$

First, two synthetic datasets from this process are generated; $\{X_t\}_{t=0}^T$, which has parameters $\mu = 20$, $\phi = 0.3$, $\sigma = 2$ and $T = 200$, and $\{Y_t\}_{t=0}^T$, which has parameters $\phi = 0.9$ and μ and σ are the same as for X_t. The trajectories for the two processes are shown in figure 6.

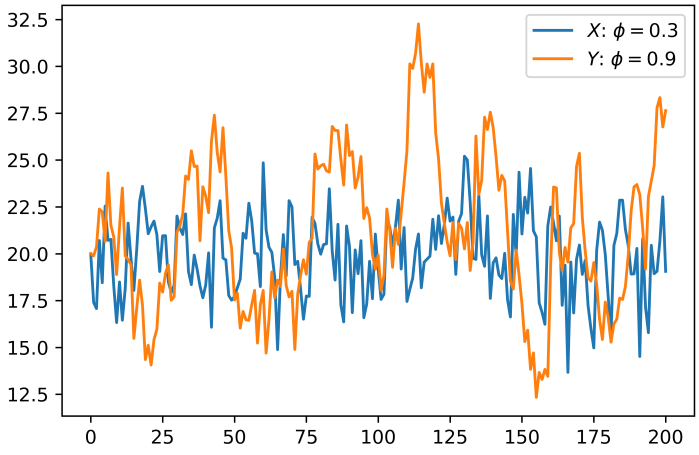

Figure 6: The trajectories of the two AR(1) processes

Next, the parameters μ, σ and ϕ are treated as unknown parameters in the Stan model, using the priors:

$$p(\phi) \propto \begin{cases} 1, & -1 \leq \phi \leq 1 \\ 0 & \text{otherwise} \end{cases} \quad p(\sigma) \propto \begin{cases} c, & 0 < \sigma \\ 0, & \text{otherwise} \end{cases} \quad p(\mu) \propto c$$

As $|\phi| > 1$ will lead to an unstable process (in general, the impact of ϕ is the autocorrelation between two subsequent samples), and σ must be positive. The way this is defined in Stan is showed in the code snippet below.

```
data {
  int<lower=0> T;
  vector[T] x;
}
parameters {
  real<lower=-1, upper=1> phi;
  real mu;
  real<lower=0> sigma;
}
model {
  for (t in 2:T)
    x[t] ~ normal(mu + phi * ( x[t-1] - mu ), sigma);
}
```

Using this model, 5000 iterations per chain and four chains where performed for each process, using the "No U-Turn Sampling" algorithm in Stan. For the first, a warm-up period of 1000 was used while, for the second, the warm-up period was 2500, as a way of increasing the robustness of the model. The results from the Monte Carlo simulations are shown below, in terms of the model output shown below, and the pairwise plots of the posterior parameters shown in figure 7.

```
x_fit

Inference for Stan model: anon_model_bed26d3d7e672f0db82b8a542ddb1573.
4 chains, each with iter=5000; warmup=1000; thin=1;
post-warmup draws per chain=4000, total post-warmup draws=16000.

         mean se_mean     sd    2.5%     25%     50%     75%   97.5%  n_eff   Rhat
phi      0.36  5.7e-4   0.07    0.22    0.31    0.36     0.4    0.49  14043    1.0
mu      19.88  2.1e-3   0.23   19.43   19.73   19.88   20.03   20.33  11258    1.0
sigma    2.01  8.4e-4    0.1    1.83    1.94    2.01    2.08    2.23  14647    1.0
lp__   -239.4    0.01   1.27  -242.6  -239.9  -239.0  -238.4  -237.9   7864    1.0

Samples were drawn using NUTS at Sat May 15 15:55:03 2021.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
y_fit

Inference for Stan model: anon_model_06adc118d3f2b3d73f4283204fec27dc.
4 chains, each with iter=5000; warmup=2500; thin=1;
post-warmup draws per chain=2500, total post-warmup draws=10000.

         mean se_mean     sd    2.5%     25%     50%     75%   97.5%  n_eff   Rhat
phi      0.91  1.8e-3   0.04    0.84    0.89    0.91    0.94     1.0    475    1.01
mu      29.78    5.55  74.35    16.6   20.54   21.57   22.73   46.14    179    1.02
sigma    1.87  1.7e-3    0.1    1.69    1.81    1.87    1.93    2.07   3250    1.0
lp__   -226.6    0.17   2.35  -233.7  -227.3  -225.9  -225.1  -224.4    187    1.03

Samples were drawn using NUTS at Sat May 15 16:27:01 2021.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

From this one can extract the following characteristics of the sample posterior:

	X				Y			
	$\hat{\mu}$	$CI_{0.95}$		n_{eff}	$\hat{\mu}$	$CI_{0.95}$		n_{eff}
ϕ	0.36	0.22	0.49	14043	0.91	0.84	1.0	475
μ	19.88	19.43	20.33	11258	29.78	16.6	46.14	179
σ	2.01	1.83	2.23	14647	1.87	1.69	2.07	3250

Table 1: Parameter data estimated from the HMC simulations

From the results summary, as well as the plots, it can clearly be seen that the samples from X can be used to accurately describe the parameters, while they fail to do so for Y. One possible reason for this is that the subsequent samples of Y have much heavier autocorrelation, which reduces the efficiency of the sampling. This is noted when looking at the number of efficient draws from Y, which is extremely low. Another possible issue was the selected prior for ϕ between -1 and 1, as this prohibits sampling much higher than the true value of ϕ.

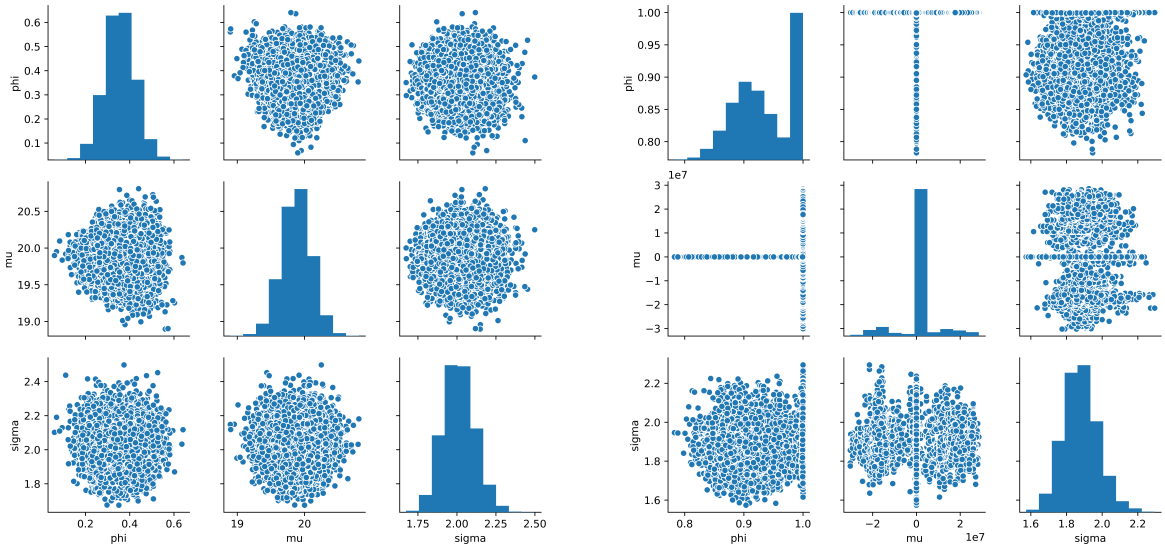

Figure 7: Pairs plots for the parameters of the two processes (left: X, right: Y)

```python
""" Some useful packages"""
import numpy as np
from scipy import stats as st
import matplotlib.pyplot as plt
import pystan as stan
import seaborn as sea

np.random.seed(1111) # Set seed for reproducibility

""" Functions that draws samples from an AR(1)-process """
def AR1(mu, phi, sigma, T, x0):
    x = np.zeros(T+1)
    x[0] = x0
    for t in range(1, T+1):
        x[t] = st.norm.rvs(loc= mu+phi*(x[t-1]-mu), scale=sigma)
    return x

""" Explore the impact of phi on x """
x = AR1(20, 0.3, 2, 200, 20)
y = AR1(20, 0.9, 2, 200, 20)

plt.plot(x)
plt.plot(y)
plt.legend([r"$X$: $\phi=0.3$","$Y$: $\phi=0.9$"])

""" Define the AR-process in PyStan """
AR_sm = stan.StanModel(file='AR1.stan')

""" And treat the previous simulations as the datasets """
AR_x_data = {"T" : 201,
             "x" : x}
AR_y_data = {"T" : 201,
             "x" : y}

x_fit = AR_sm.sampling(data=AR_x_data, iter=5000, warmup=1000)
```

```
# Increase warmup to get a better n_eff
y_fit = AR_sm.sampling(data=AR_y_data, iter=5000, warmup=2500)

x_fit_df = x_fit.to_dataframe()[['phi', 'mu', 'sigma']]
y_fit_df = y_fit.to_dataframe()[['phi', 'mu', 'sigma']]

sea.pairplot(x_fit_df)
sea.pairplot(y_fit_df)
```