ENSAE ParisTech

Bayesian Statistics Porject

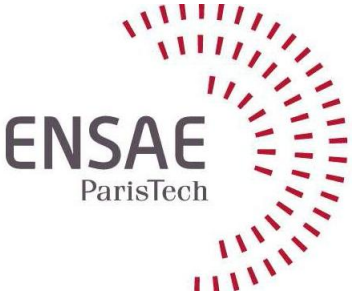
A study of the 1987 paper by Martin A. Tanner and Wing Hung Wong:

The Calculation of Posterior Distribution by Data Augmentations

Alexis Rosuel
Thomas Seleck
Gaston Bizel-Bizellot

Professors:
Anna Simoni
Rémi Bardenet

January 30th 2016



Introduction

Let's start by giving a quick overview of the algorithm presented in this paper from 1987: the main idea here is to artificially generate some new data from the available observations in order to get a better estimation of the posterior distribution.

1 Presentation of the algorithm

1.1 The intuition behind the algorithm

In this section, we will detail the proofs presented in the article, and stick to the notations used by the authors. The following bullet points give the 'real' interpretation of the variables :

- $\theta \in \Theta$ is the parameter of interest, ie. the one that we will want to estimate
- $y \in \mathcal{Y}$ is the available data
- $z \in \mathcal{Z}$ is the artificially generated data (using several methods detailed below)
- $x \in \mathcal{X}$ represents the union between the y and the z, ie. the "augmented data"

In the Bayesian framework, the quantity we are looking for is the posterior distribution $p(\theta|y)$. It can be written as follows:

$$\begin{aligned} p(\theta|y) &= \int_{\mathcal{Z}} p(\theta, z|y)dz \text{ (marginalisation)} \\ &= \int_{\mathcal{Z}} p(\theta|z, y)p(z|y)dz \text{ (conditional probability)} \end{aligned}$$

In this last expression, the posterior distribution is expressed as function of the "real" data and the generated data. We now need to define which process allows the generation of the z:

$$\begin{aligned} p(z|y) &= \int_{\Theta} p(z, \phi|y)d\phi \text{ (marginalisation)} \\ &= \int_{\Theta} p(z|\phi, y)p(\phi|y)d\phi \text{ (conditional probability)} \end{aligned}$$

Here, we understand that z is drawn from a "continuous mixture" of laws, each weighted by the posterior probability we are looking for. We can also note the recursion: the weights used for the data augmentation is the distribution we want to find.

Now let's focus on this recursion. We use the computed expression of $p(z|y)$ in $p(\theta|y)$:

$$\begin{aligned} p(\theta|y) &= \int_{\mathcal{Z}} p(\theta|z, y)p(z|y)dz \\ &= \int_{\mathcal{Z}} p(\theta|z, y) \left\{ \int_{\Theta} p(z|\phi, y)p(\phi|y)d\phi \right\} dz \\ &= \int_{\Theta} p(\phi|y) \left\{ \int_{\mathcal{Z}} p(\theta|z, y)p(z|\phi, y)dz \right\} d\phi \end{aligned}$$

Thus, by denoting $K(\theta, \phi) = \int_{\mathcal{Z}} p(\theta|z, y)p(z|\phi, y)dz$ and $g(\theta) = p(\theta|y)$, we get:

$$g(\theta) = \int_{\mathcal{Z}} K(\theta, \phi)g(\phi)d\phi$$

So g is the fixed point of the application T defined by

$$T(f(\theta)) = \int_{\mathcal{Z}} K(\theta, \phi)f(\phi)d\phi$$

The goal here is to compute the posterior distribution. We will now be able to do that, not with the usual method using the Bayes formula, but by finding the fixed point of the application f. We already know how to solve this problem using iterations:

- we start with a random value of g, let's call it g_0
- we senquentially compute $g_{i+1} = T(g_i)$

This way, the sequence (g_i) converges to the fixed point g. Although this result remains true in most cases, some mild conditions have to be satisfied. They will be detailed later.

1.2 The final algorithm

The final algorithm has two main parts:

- the first part generates artificial data z
- the second part uses the initial data y and the new data z to suggest a new form for the desired posterior distribution

More precisely, here are the steps to implement :

- generate $\phi \sim g_i(\theta)$
- generate $z^{(j)} \sim p(z|\phi, y)$
- compute $g_{i+1}(\theta) = m^{-1} \sum_{j=1}^{m} p(\theta|z^{(j)}, y)$

The sequence (g_i) under some hypothesis will converge to the desired distribution g.

2 Comparison with the EM algorithm

The data augmentation algorithm can be seen as a Bayesian view of the EM algorithm. Indeed:

- the EM algorithm makes accessible the maximum of a likelihood (the Maximum A Posteriori)
- the data augmentation algorithm tries to recover the whole a posteriori distribution

This is the same dichotomy as the Frequentist vs Bayesian point of view (point estimation vs whole distribution recovery).

Moreover, both algorithm rely on unobserved data in order to improve the estimations :

- the EM algorithm uses latent variables
- the data augmentation algorithm generates its own data

Finally, both algorithms use an iterative process with a fixed point theorem.

3 Proof of the algorithm

In this section, we will focus on the main arguments provided in the proof of the algorithm presented in the paper. We will use the same notations, which are the following:

- $K(\theta, \phi) = \int p(\theta|z, y)p(z|\phi, y)dz$
- T is the application defined by $T(f(\theta)) = \int K(\theta, \phi)f(\phi)d\phi$
- g is a function in L_1 (space of Lebesgue integrable functions)
- g_* is the true posterior density

The proof is articulated around these 4 major arguments :

- g_* is the only density that satisfies $Tg = g$
- $\|g_{i+1} - g_*\| \leq \|g_i - g_*\|$
- the distance is strictly decreasing
- the decrease is geometric : $\alpha \in]0, 1[, \|g_{i+1} - g_*\| \leq \alpha^i \|g_0 - g_*\|$

4 Implementation of three examples

We implemented three examples presented in the paper in Python 3.5. These examples are:

- The genetic linkage example: this example was presented by Rao in 1973 and described as follows: from a genetic linkage model, we believe that 197 animals are distributed multinomially into four categories, $y = (y_1, y_2, y_3, y_4) = (125, 18, 20, 34)$ with cell probabilities defined by

$$(\frac{1}{2} + \frac{\theta}{4}, \frac{1-\theta}{4}, \frac{1-\theta}{4}, \frac{\theta}{4})$$

- The estimation of the correlation coefficient from a bivariate normal distribution
- The application of the Dirichlet Sampling Process to approximately sample from the posterior distribution of parametric models for multinomial data when we can't draw samples directly from the posterior distribution. This process will be illustrated by the previous genetic linkage example.

4.1 Genetic Linkage model

In this example, we want to estimate the parameter θ which influences the distribution of animals between classes. We've implemented the "basic" algorithm presented in the paper which works as follows:

Algorithm 1 Basic algorithm for estimating the value of θ for Genetic Linkage example

Require:
iterations: Number of algorithm's iterations
m: Number of samples
y: Observed data
Ensure:
θ: This is what we want to estimate
$\theta \leftarrow \theta_0$
$x_3 \leftarrow y_2$
$x_4 \leftarrow y_3$
$x_5 \leftarrow y_4$
for *iter* `in` $1 : iterations$ **do**
# Step 1: Imputation step
$\theta \leftarrow (\theta_1, ..., \theta_m)$ where $\theta_i \sim p(\theta|y) \forall i \in 1, ..., m$
$x_2 \leftarrow (b_1, ..., b_m)$ where $b_i \sim \mathcal{B}(y_1, \frac{\theta}{\theta+2}) \forall i \in 1, ..., m$
Step 2: Posterior step, update the current approximation of $p(\theta|y)$
$\nu_1^{(i)} \leftarrow x_2^{(i)} + x_5 + 1$
$\nu_2^{(i)} \leftarrow x_3 + x_4 + 1$
$p(\theta|y) \leftarrow \frac{1}{m} \sum_{i=1}^{m} Be(\nu_1^{(i)}, \nu_2^{(i)})(\theta)$ where $Be(\nu_1^{(i)}, \nu_2^{(i)})$ is the Beta distribution
end for
return θ

We were able to reproduce the results of the paper for this algorithm. We ran it on the three observed data sets provided in the paper, which are $y = (125, 18, 20, 34), y = (13, 2, 2, 3)$ and $y = (14, 0, 1, 5)$. We chose to keep m and *iterations* constant regardless of the chosen data set. So, we took $m = 1600$ and $iterations = 100$.

The three following Figures show the results we have obtained. For each plot, the blue curve represents the output distribution of the algorithm, the red one is the beta distribution $Be(\nu_1, \nu_2)$ and the green one is the true posterior distribution.

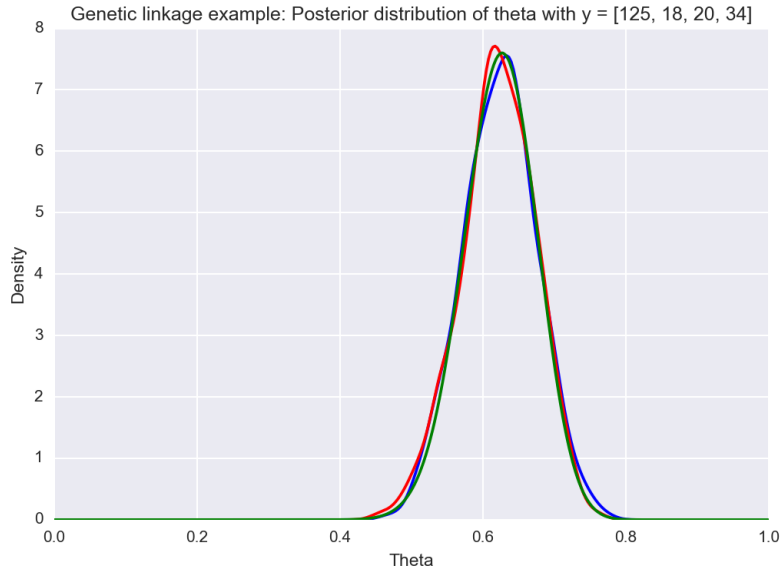


Figure 1: Genetic linkage example — Posterior distribution of θ with $y = (125, 18, 20, 34)$

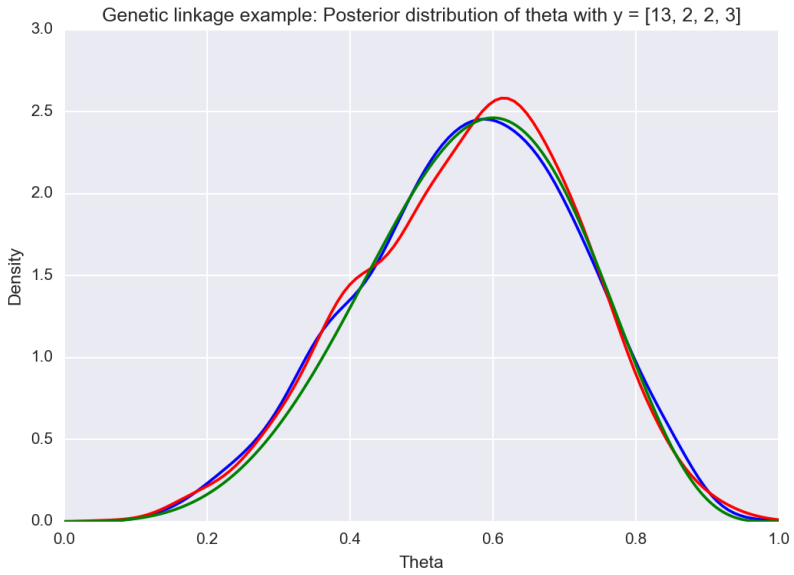


Figure 2: Genetic linkage example — Posterior distribution of θ with $y = (13, 2, 2, 3)$

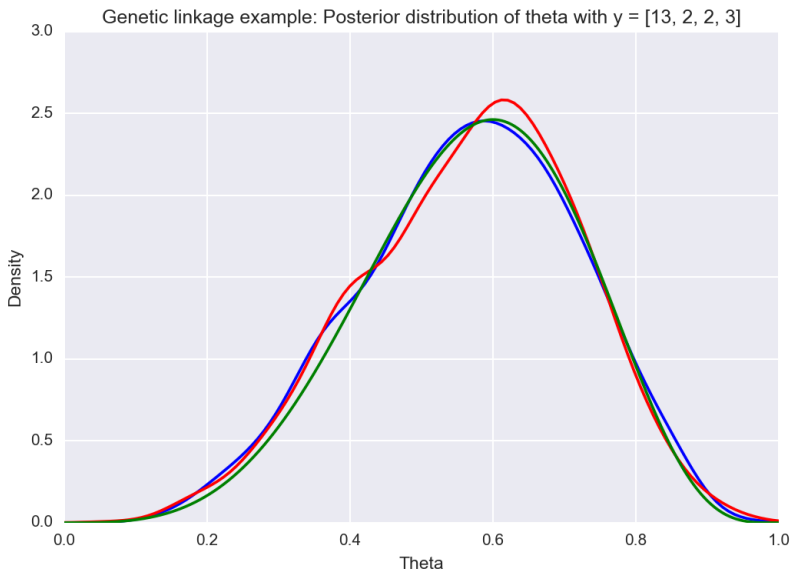


Figure 3: Genetic linkage example — Posterior distribution of theta with $y = (13, 2, 2, 3)$

As we can see, the blue and green curves are really close, meaning that this algorithm successfully manages to compute the posterior distribution in each case.

4.2 Estimation of the correlation coefficient from a bivariate normal distribution

In this example, we want to estimate the correlation coefficient from a bivariate normal distribution. We have 12 samples from the bivariate normal distribution with $\mu_1 = \mu_2 = 0$, a correlation coefficient ρ and variances σ_1^2 and σ_2^2. However, the dataset contains 8 incomplete observations. The dataset is visible in Table 1.

Table 1: Twelve observations from a bivariate normal distribution

Distribution 1 —	1	1	-1	-1	2	2	-2	-2	*	*	*	*
Distribution 2 —	1	-1	1	-1	*	*	*	*	2	2	-2	-2

The implementation of the algorithm is as follows:

- Given the covariance matrix Σ, generate missing data from two gaussian distributions, depending on which series the observation is missing.
- Then, we compute the covariance matrix from the current guess of the posterior distribution $p(\Sigma|y)$.
- Next, we update the current approximation of $p(\Sigma|y)$ by drawing m samples from the mixture of inverse Wishart distributions.
- We update the values of σ_1, σ_2 et ρ.

The following plot shows the posterior distribution we've computed using the algorithm above in blue. The green curve shows the true posterior distribution computed with the formula given in the paper. The parameters of the algorithm we took are the following: 4 iterations and $m = 6,400$.

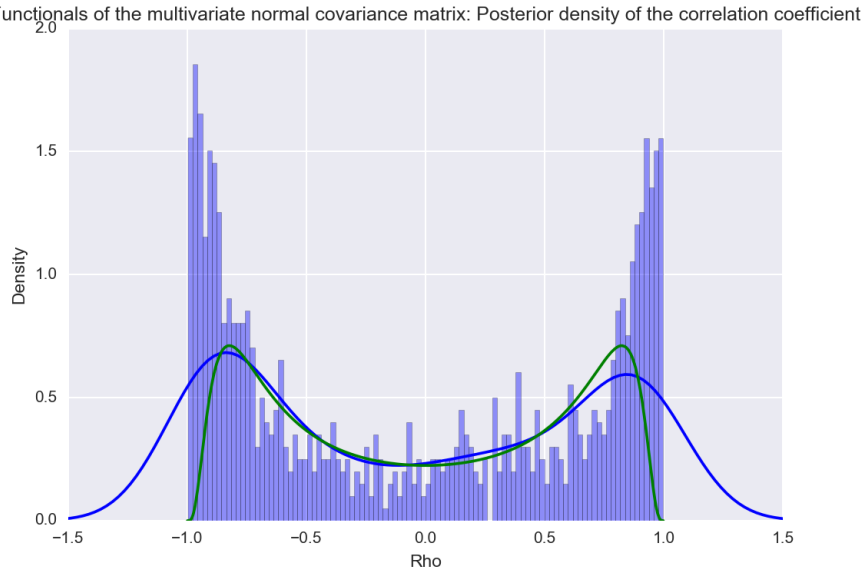


Figure 4: Genetic linkage example — Posterior distribution of theta with $y = (13, 2, 2, 3)$

The shape of the posterior distribution we've computed is similar to the paper's, but we can notice that our curve ranges from -1.5 to 1.5 on the x axis whilst the paper's one ranges from -1 to 1. We conclude that there must exist an error in our code, but we couldn't figure out where.

Indeed, we tried to find how the true posterior distribution was found : $p(\rho|y) = [(1-\rho^2)^{4.5}]/[(1.25-\rho^2)^8]$. We wrote :

$$
\begin{aligned}
p(y|\rho) &= \prod_{i=1}^{4} p(y_i|\rho) && (1)\\
&= \prod_{i=1}^{4} \int_{\sigma_1^2,\sigma_2^2=0}^{+\infty} p(y_i, \sigma_1^2, \sigma_2^2|\rho) d\sigma_1^2 d\sigma_2^2 && (2)\\
&= \prod_{i=1}^{4} \int_{\sigma_1^2,\sigma_2^2=0}^{+\infty} p(y_i|\sigma_1^2, \sigma_2^2, \rho) p(\sigma_1^2, \sigma_2^2) d\sigma_1^2 d\sigma_2^2 && (3)
\end{aligned}
$$

with

$$
p(y_i|\sigma_1^2, \sigma_2^2, \rho) \propto \frac{1}{\sigma_1^2\sigma_2^2 - \rho^2} e^{y_i^T \Sigma^{-1} y_i}
$$

$$
p(\sigma_1^2, \sigma_2^2) = \sigma_1^{-2}\sigma_2^{-2}
$$

and :

$$
p(\rho|y) = \frac{p(y|\rho)p(\rho)}{p(y)} \qquad (4)
$$

but unfortunately, the expressions found were intractable, and we couldn't find a justification about the choice of the true posterior distribution in the paper.

However, the data we used gives evidence for $\theta = 1$ or $\theta = -1$, not $\theta = 0.8$ or $\theta = -0.8$ as the green curve appears to show (the one computed in the paper). As a consequence, our distribution seems to be more natural... We can try with the data of Table 2, which should lead to a posterior distribution concentrated near $+1$.

Table 2: Twelve observations from a bivariate normal distribution ($\rho = +1$)

Distribution 1 —	1	1	-1	-1	2	2	-2	-2	*	*	*	*
Distribution 2 —	1	1	-1	-1	*	*	*	*	2	2	-2	-2

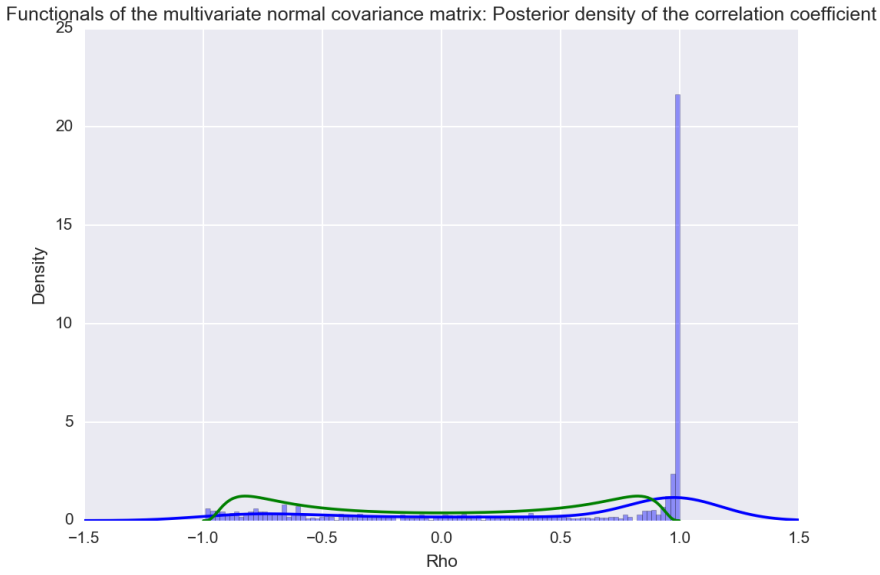


Figure 5: Posterior density of the correlation coefficient

Indeed, we observe that almost all the weight of the posterior distribution of θ is near $+1$.

Before ending this section, we stress the fact that the Bayesian point of view is very informative in the first example of the estimation of the correlation coefficient. Because the posterior is bi-modal, with each mode having the same intensity, a point estimation (in frequentist) would answer 0 ! Retrieving the whole posterior distribution enables the statistician to understand better what are the evidence for each possible value of θ.

4.3 Dirichlet sampling applied to Genetic Linkage example

In the first example we've implemented in Python, the posterior is a Beta distribution we can easily sample from. But in some cases, sampling θ from $p(\theta|x)$ is hard. So, we'll use the Dirichlet sampling to sample from the posterior distribution. In order to show that Dirichlet sampling is working properly, we'll use the Genetic Linkage example again, replacing the Beta distribution by a Dirichlet distribution this time.

The algorithm then becomes:

- The first step where we generate z from $p(z|\phi, y)$ is the same as in the former genetic linkage example.
- We generate m samples from the Dirichlet distribution $D(\frac{z+1}{2}, \frac{y_2+1}{2}, \frac{y_3+1}{2}, \frac{y_4+1}{2})$.
- Then, we compute $\hat{\theta} = 2(p_2 + p_5)$ for each sample where (p_2, p_3, p_4, p_5) is a sample generated by the previous Dirichlet distribution.
- We compute $\hat{p} = (\frac{\hat{\theta}}{4}, \frac{1}{4} - \frac{\hat{\theta}}{4}, \frac{1}{4} - \frac{\hat{\theta}}{4}, \frac{\hat{\theta}}{4})$ for each sample.
- We keep a sample only if it's located near the parametric curve C defined by

$$C = \{(\frac{\hat{\theta}}{4}, \frac{1}{4} - \frac{\hat{\theta}}{4}, \frac{1}{4} - \frac{\hat{\theta}}{4}, \frac{\hat{\theta}}{4}) : \theta \in [0, 1]\}$$

We actually do this by keeping the sample if

$$\sqrt{\sum_{i=2}^{5}(p_i - \hat{p_i})^2} < \varepsilon$$

The parameters of the algorithm we took are the following: 20 iterations, $m = 10,000$, $y = (3, 2, 2, 3)$ and $\varepsilon = .20$. We obtain the following curves: the blue one is the algorithm's output (about 3,000 samples kept at the end) whilst the green one is the true posterior distribution.

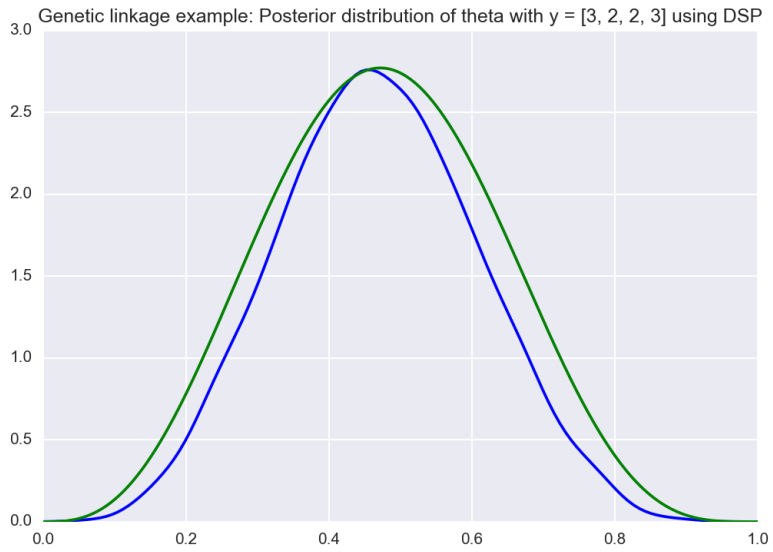


Figure 6: Genetic linkage example — Posterior distribution of θ with $y = (3, 2, 2, 3)$ using DSP

Conclusion

Missing data traditionally doesn't have much use and can generally create many problems when it comes to analysing the data. So when confronted to missing values, one of the following options is usually chosen:

- Deleting the whole row
- Filling all missing values using medians or clustering methods such as kNN

However, this paper suggests an alternative method that enables us to "use" the missing data: using artificially augmented data, we can compute the posterior distribution with better precision.

Appendices

Appendice 1: Python Code

```
#!/usr/bin/env python

"""project_code.py: Main file of the project."""

__author__ = "Thomas SELECK, Alexis ROSUEL, Gaston BIZEL"
__credits__ = ["Thomas SELECK", "Alexis ROSUEL", "Gaston BIZEL"]
__license__ = "GPL"
__version__ = "1.0.0"
__status__ = "Production"

from numpy.random import binomial
from numpy.random import uniform
from numpy.random import chisquare
from numpy.random import normal
from numpy.random import randint
from scipy import stats
import numpy as np
import seaborn as sns
import time
from operator import itemgetter
from scipy.stats import gaussian_kde

def BasicAlgorithmLinkageExample(iterations, m, y, displayPlot):
    """
    This function implements the basic algorithm presented in the paper (Section 2).
    It's basically an EM algorithm to compute the posterior distribution of a parameter theta.

    Parameters
    ----------
    iterations : positive integer
            This number is the number of iterations we want for our algorithm.

    m : positive integer
            This represents the number of sample we'll use.

    y : list
            This represents the observed data.

    displayPlot : boolean
            If true, the plot is displayed. Otherwise, it is save as a png file in the current directory.

    Returns
    -------
    theta : float
            This is the posterior estimation of theta we've computed.
    """

    print("Computing posterior distribution for the Genetic Linkage example with y = " + str(y) + " ...")

    # Step 1: Generate a sample z_1, ...z_m from the current approximation of theta
```

```
    ## Step 1.1. Generate theta from g_i(theta)
    theta = uniform(size = m)

    for i in range(iterations):
        ## Step 1.2. Generate z from p(z|phi, y) where phi is the value obtained in Step 1.1.
        z = binomial(y[0], theta / (theta + 2), m)

        # Step 2: Update the current approximation of p(theta|y)
        nu_1 = z + y[3] + 1
        nu_2 = y[1] + y[2] + 1

        # Select a distribution from the mixture of beta distributions and do it m times to get m samples
        idx = randint(0, m, size = m)

        # Draw a sample for theta from the mixture of beta and do it m times
        theta = stats.beta.rvs(nu_1[idx], nu_2, size = m)

    # Compute the true posterior distribution
    x = uniform(size = m) # Silent variable to plot the true posterior.
    truePosterior = (((2 + x) ** y[0]) * ((1 - x) ** (y[1] + y[2])) * (x ** y[3]))

    # Scale the true posterior distribution: Quick and dirty way to do this
    truePosterior *= np.max(gaussian_kde(theta).pdf(x)) / np.max(truePosterior)

    x, truePosterior = [list(x) for x in zip(*sorted(zip(x, truePosterior), key=itemgetter(0)))]

    sns.distplot(theta, hist = False, kde = True, color = "b").set(xlim = (0, 1))
    sns.distplot(stats.beta.rvs(nu_1, nu_2, size = m), hist = False, kde = True, color = "r").set(xlim = (0, 1), xlabel = "Theta", ylabel = "Density")
    sns.plt.plot(x, truePosterior, color = "g")
    sns.plt.title("Genetic linkage example: Posterior distribution of theta with y = " + str(y))

    if displayPlot:
        sns.plt.show()
        sns.plt.cla()
    else:
        sns.plt.savefig("genetic_linkage_example_" + "_".join([str(i) for i in y]) + ".png", dpi = 150)
        sns.plt.cla()

    return theta

def DirichletSamplingProcessLinkageExample(iterations, m, y, epsilon, displayPlot):
    """
    This function implements the Dirichlet sampling process presented in the paper (Section 4).
    It's basically an EM algorithm to compute the posterior distribution of a parameter theta. We use
    the Dirichlet sampling when the sampling of theta from p(theta|x) is not simple.
```

```
96
97      Parameters
98      ----------
99      iterations : positive integer
100             This number is the number of iterations we want for our algorithm.
101
102     m : positive integer
103             This represents the number of sample we'll use.
104
105     y : list
106             This represents the augmented data.
107
108     displayPlot : boolean
109             If true, the plot is displayed. Otherwise, it is save as a png
    ↪ file in the current directory.
110
111     Returns
112     -------
113     theta : float
114             This is the posterior estimation of theta we've computed.
115     """
116
117     print("Computing posterior distribution for the Genetic Linkage example
        ↪ with y = " + str(y) + " using DSP...")
118
119     # Step 1: Generate a sample z_1, ...z_m from the current approximation of
        ↪ theta
120     ## Step 1.1. Generate theta from g_i(theta)
121     theta = uniform(size = m)
122
123     for i in range(iterations):
124         ## Step 1.2. Generate z from p(z|phi, y) where phi is the value
            ↪ obtained in Step 1.1.
125         z = binomial(y[0], theta / (theta + 2), m)
126
127         # Step 2: Update the current approximation of p(theta|y)
128         ## Sample observations from Dirichlet distribution
129         dirichletSamples = []
130         for i in range(m):
131             augmented_data = [z[i] + 1, y[1] + 1, y[2] + 1, y[3] + 1] # We add
                ↪ 1 to ensure each number is greater than 0
132             dirichletSamples.append(stats.dirichlet.rvs(augmented_data)[0] /
                ↪ 2)
133
134         dirichletSamples = np.array(dirichletSamples)
135
136         ## Compute theta_hat
137         theta_hat_array = []
138         for sample in dirichletSamples:
139             theta_hat = 2 * (sample[0] + sample[3])
140             p_hat = np.array([theta_hat / 4, 1 / 4 - theta_hat / 4, 1 / 4 -
                ↪ theta_hat / 4, theta_hat / 4])
141
142             if np.sqrt(np.sum((np.array(sample) - p_hat) ** 2)) < epsilon:
143                 theta_hat_array.append(theta_hat)
144
145         m = len(theta_hat_array)
146         theta_hat_array = np.array(theta_hat_array)
147         theta = theta_hat_array
```

```
148
149     # Compute the true posterior distribution
150     x = uniform(size = m) # Silent variable to plot the true posterior.
151     truePosterior = (((2 + x) ** y[0]) * ((1 - x) ** (y[1] + y[2])) * (x **
        ↪ y[3]))
152
153     # Scale the true posterior distribution: Quick and dirty way to do this
154     truePosterior *= np.max(gaussian_kde(theta).pdf(x)) /
        ↪ np.max(truePosterior)
155
156     x, truePosterior = [list(x) for x in zip(*sorted(zip(x, truePosterior),
        ↪ key=itemgetter(0)))]
157
158     sns.distplot(theta, hist = False, kde = True, color = "b").set(xlim = (0,
        ↪ 1))
159     sns.plt.plot(x, truePosterior, color = "g")
160     sns.plt.title("Genetic linkage example: Posterior distribution of theta
        ↪ with y = " + str(y) + " using DSP")
161
162     if displayPlot:
163         sns.plt.show()
164         sns.plt.cla()
165     else:
166         sns.plt.savefig("genetic_linkage_example_DSP" + "_".join([str(i) for i
            ↪ in y]) + ".png", dpi = 150)
167         sns.plt.cla()
168
169     return theta
170
171 def BasicAlgorithmMultivariateCovarianceMatrix(iterations, m, displayPlot):
172     """
173     This function implements the basic algorithm presented in the paper
    ↪ (Section 2).
174     It's basically an EM algorithm to compute the posterior distribution of a
    ↪ parameter theta.
175
176     Parameters
177     ----------
178     iterations : positive integer
179             This number is the number of iterations we want for our algorithm.
180
181     m : positive integer
182             This represents the number of sample we'll use.
183
184     displayPlot : boolean
185             If true, the plot is displayed. Otherwise, it is save as a png
    ↪ file in the current directory.
186
187     Returns
188     -------
189     Sigma : numpy array
190             This is the posterior estimation of the covariance matrix we've
    ↪ computed.
191     """
192
193     print("Computing posterior distribution for the functionals of the
        ↪ multivariate normal covariance matrix example...")
194
```

```
    # x1 and x2 contain the original censored data while x contains both, duplicated m times
    x1 = np.array([1, 1, -1, -1, 2, 2, -2, -2, np.nan, np.nan, np.nan, np.nan])
    x2 = np.array([1, -1, 1, -1, np.nan, np.nan, np.nan, np.nan, 2, 2, -2, 2])
    x = np.array([[x1.astype(np.float), x2.astype(np.float)] for i in range(m)])

    # We initialize the the parameters in the same way of the paper
    rho = uniform(-1, 1, m)
    sigma1 = np.sqrt(chisquare(7, m))
    sigma2 = np.sqrt(chisquare(7, m))
    Sigma = np.array([[[sigma1[i] ** 2, rho[i] * sigma1[i] * sigma2[i]], [rho[i] * sigma1[i] * sigma2[i], sigma2[i] ** 2]] for i in range(m)])

    for iter in range(iterations):
        # Step 1: We impute the missing data by drawing samples from a normal distribution. Its variance depends on which series have missing values.
        for obs_idx in range(x.shape[2]):
            if not np.isnan(x1[obs_idx]) and np.isnan(x2[obs_idx]):
                for i in range(m):
                    x[i, 1, obs_idx] = normal(rho[i] * (sigma2[i] / sigma1[i]) * x1[obs_idx], (sigma2[i] ** 2) * (1 - (rho[i] ** 2)))
            elif not np.isnan(x2[obs_idx]) and np.isnan(x1[obs_idx]):
                for i in range(m):
                    x[i, 0, obs_idx] = normal(rho[i] * (sigma1[i] / sigma2[i]) * x2[obs_idx], (sigma1[i] ** 2) * (1 - (rho[i] ** 2)))

        # Compute the covariance matrix
        covarianceMatrix = np.array([np.cov(x[i]) / x1.shape[0] for i in range(m)])

        # Step 2: Update the current approximation of p(Sigma|y)
        # Select a distribution from the mixture of inverted Wishart distributions and do it m times to get m samples
        idx = randint(0, m, size = m)

        # Draw a sample for Sigma from the mixture of inverted Wishart distributions and do it m times
        Sigma = np.array([stats.invwishart.rvs(df = 4, scale = covarianceMatrix[idx[i]]) for i in range(m)])

        ## We update the values of sigma1, sigma2 and rho
        sigma1 = np.sqrt(Sigma[:, 0, 0])
        sigma2 = np.sqrt(Sigma[:, 1, 1])
        ### Compute the associated correlation coefficient for each observation
        rho = Sigma[:, 1, 0] / (sigma1 * sigma2)

    # Compute the true posterior distribution
    x = uniform(low = -1, size = m)
    truePosterior = ((1 - (x ** 2)) ** 4.5) / ((1.25 - (x ** 2)) ** 8)

    # Scale the true posterior distribution: Quick and dirty way to do this
    truePosterior *= np.max(gaussian_kde(rho).pdf(x)) / np.max(truePosterior)

    x, truePosterior = [list(x) for x in zip(*sorted(zip(x, truePosterior), key=itemgetter(0)))]
```

```
240
241     sns.distplot(rho, hist = False, kde = True, color = "b").set(xlim = (-1.5,
        ↪ 1.5), xlabel = "Rho", ylabel = "Density")
242     sns.plt.plot(x, truePosterior, color = "g")
243     sns.plt.title("Functionals of the multivariate normal covariance matrix:
        ↪ Posterior density of the correlation coefficient")
244
245     if displayPlot:
246         sns.plt.show()
247         sns.plt.cla()
248     else:
249         sns.plt.savefig("multivariate_normal_covariance_matrix_example.png",
            ↪ dpi = 150)
250         sns.plt.cla()
251
252     return Sigma
253
254
255 if __name__ == "__main__":
256     # Set the PRNG's seed
257     np.random.seed(45)
258
259     # Start the timer
260     startTime = time.time()
261
262     # Do you want to see plots or save them to files?
263     displayPlots = True
264
265     # For linkage example
266     iterations = 100
267     m = 1600
268     data = [[125, 18, 20, 34], [13, 2, 2, 3], [14, 0, 1, 5]]
269
270     for y in data:
271         res = BasicAlgorithmLinkageExample(iterations, m, y, displayPlots)
272
273     # For multivariate covariance matrix
274     iterations = 15
275     m = 6400
276     res = BasicAlgorithmMultivariateCovarianceMatrix(iterations, m,
        ↪ displayPlots)
277
278     # For linkage example
279     iterations = 20
280     m = 10000
281     y = [3, 2, 2, 3]
282     epsilon = 0.20
283
284     res = DirichletSamplingProcessLinkageExample(iterations, m, y, epsilon,
        ↪ displayPlots)
285
286     # Stop the timer and print the exectution time
287     print("Exec: --- %s seconds ---" % (time.time() - startTime))
```