Gibbs Sampling for the Multivariate Gaussian mixture model

Computational Statistics Project 3

Leonardo Ruggieri

1 Introduction

Mixture models are simple yet powerful instruments when it comes to capture complex distribution and model a wide variety of data. One common task is to perform clustering by means of a hierarchical model with latent variables. The basic element is the mixture of Gaussian distributions, that considers a superposition of Gaussian densities according to some prior mixing coefficients. In the following, a simple Gibbs sampler is motivated and implemented with synthetic data in a Python code, for which convergence diagnostics is provided. Finally, an application to color image segmentation is delivered.

2 The model

2.1 The Gaussian mixture distribution

The Gaussian mixture model provides a richer class of density models than a single Gaussian distribution. The mixture distribution can be defined as follows:

$$p(x) = \sum_{z=1}^{K} \pi_z \mathcal{N}(x|\mu_z, \Sigma_z)$$

where π_k are the mixing coefficients [1]. Given that their sum must be equal to one, that is, $\sum_{z=1}^{K} = 1$, and that $\mathcal{N}(x|\mu_z, \Sigma_k)$ is non-negative, $0 \leq \pi_k \leq 1 \; \forall k = 1, ..., K$ must hold, so that we have $p(x) \geq 0$ on its support. Hence, the mixing coefficients satisfy the requirements to be probabilities.

2.2 Formulation in terms of latent variables z

We can equivalently represent the mixture distribution by means of a latent (unobserved) variable z. Hence, with multiple data points x_i, there is a corresponding latent variable z_i for each one of them, which identifies its cluster assignment. This latent variables will be critical in our applications: we want to make Bayesian inference on it, so to discover which group every data point belongs to. Thus, the marginal density will be:

$$p(x) = \sum_{z} p(z)p(x|z)$$

2.3 The model

By specifying the priors for the cluster means and by fixing a common variance σ^2 for the observations, the hierarchical model is the following:

$$\mu_z \sim \mathcal{N}(0, \lambda^2 I_d) \quad z = 1, ..., K \tag{1}$$

$$z_i \sim \textit{Categorical}(1/K, ..., 1/K) \quad i = 1, ..., n \tag{2}$$

$$x_i | z_i, \mu \sim \mathcal{N}(\mu_{z_i}, \sigma^2 I_d) \quad i = 1, ..., n \tag{3}$$

The mean parameters are independently drawn from a common prior, expressed by a multivariate normal centered in zero. In addition, d is the dimension of the space of the observation x_i's. Moreover, μ_{z_i} is the mean vector of the cluster to which x_i belongs.

3 Gibbs sampling

3.1 Motivation

The MCMC approach, in general, requires to simulate a Markov chain that is invariant with respect to the target distribution. The Gibbs sampler is a technique for indirectly generating random variables from a marginal distribution without having to calculate the density explicitly[3]. Specifically, it can be useful when dealing with multidimensional target distributions, for which the analytical formulation can be hard to compute. With the Gibbs sampler, we only deal with full (or complete) conditional distributions, for which it is typically easier to obtain. Then, the Ergodic theorem, the very same realizations of the Markov chains are used to obtain an approximation of the posterior distribution and the Bayesian estimates of interest. The Markov chain is built by using the complete conditionals as transition densities: thus, each of the generated subsequences is a Markov chain, which has the corresponding marginal as invariant distribution. In general, the process is as follows:

1. Initialize $X_0 = x_0$
2. For t=1, 2, ... , given $\mathbf{x}=(x_1^{(t)}, ..., x_d^{(t)})$ draw:

   $X_1^{t+1} \sim f_1(x_1 | x_2^{(t)}, ..., x_d^{(t)})$

   $X_2^{t+1} \sim f_2(x_2 | x_1^{(t+1)}, x_3^{(t)}, ..., x_d^{(t)})$

   ...

   $X_d^{t+1} \sim f_d(x_d | x_1^{(t+1)}, ..., x_{d-1}^{(t+1)})$

Hybrid Gibbs sampling The Gibbs sampler is a Metropolis-Hastings algorithm, where the acceptance probability of each step is equal to 1. The two methods are often used combined: for instance, when the full conditionals of all the variables and parameters of interest are not analytically tractable, the

Metropolis-Hastings can be incorporated within a Gibbs sampler to draw samples from the variables whose full conditionals cannot be analytically determined. This scheme is often called *Metropolis-Hastings-within-Gibbs* algorithm. One example of the use of such algorithm is the *latent position cluster model* for social network [4], which builds on our hierarchical model: the observations are mapped into a latent space, whose coordinates are drawn from a Gaussian mixture.

3.2 Gibbs sampling for the Gaussian mixture model

Let's go back to our original mixture of multivariate Gaussians. For a sample of size n, the joint density of x, z and μ, where z and μ are the latent variables, is as follows:

$$p(x, z, \mu) = p(\mu) \prod_{i=1}^{N} p(z_i) p(x_i|z_i, \mu)$$

In order to get the posterior distributions of μ, we wish to compute $p(\mu|x) = \frac{p(\mu,x)}{p(x)}$. In order to do so, we need to work out the marginal distribution $p(x)$. However, such a distribution, shown in 4, is rather intractable from a computational standpoint, as one must consider all the possible configurations of cluster assignment, with the computational complexity increasing exponentially in K [2].

$$p(x) = \sum_{z} \int p(\mu) \prod_{i=1}^{n} p(x_i|z_i, \mu) d\mu \tag{4}$$

We set a uniform prior over the cluster assignment for each data point, so that $\pi_{z_i} = \frac{1}{K}$. In order to obtain the Gibbs sampler, the full conditionals must be worked out. As far as the complete conditionals of z are concerned, by exploiting the conditional independence, one can easily get:

$$p(z_i|\mu, z_{-i}, x) = p(z_i|\mu, x_i) \propto p(z_i) p(x_i|\mu) = \pi_{z_i} \phi(x_i; \mu_{z_i}, \sigma^2 I_d)$$

where ϕ is a Gaussian density evaluated at x_i and μ_{z_i} is the mean vector. Similarly, one can obtain the complete conditional for μ_k:

$$p(\mu_k|\mu_{-k}, z, x) = p(\mu_k|z, x) \sim \mathcal{N}(\hat{\mu}_k, \hat{\lambda}_k)$$

where $\hat{\mu}_k$ and $\hat{\lambda}_k$ are easily obtained by conjugacy results.

The final algorithm is shown in the pseudocode below.

Algorithm 1: Gibbs Sampling algorithm for Gaussian mixture

Initialize mixture locations μ and mixture assignments z.
for t = *1,...,T* **do**
  for i = *1,...n* **do**
    for z = *1,...,K* **do**
      Compute $c(z_i = z) = \pi_z \phi(x_i; \mu_z, \sigma^2 I_d)$
    end
    for z = *1,...,K* **do**
      Normalize $p(z_i = z) = \frac{c(z_i=z)}{\sum_{z=1}^{K} \pi_z \phi(x_i;\mu_z,\sigma^2 I_d)}$
    end
    Draw $z_i|\mu, x_i$ from a Categorical with weights $p(z_i)$
  end
  for k = *1,...,K* **do**
    Compute $n_k = \sum_{i=1}^{n} z_i^k$. Compute $\bar{x}_k = \frac{\sum i=1^n z_i^k x_i}{n_k}$
    Compute $\bar{\mu}_k = \frac{n_k/\sigma^2}{n_k/\sigma^2+1/\lambda}\bar{x}_k$. Compute $\hat{\lambda} = (n_k/\sigma^2 + 1/\lambda^2)^{-1}$
    Draw $\mu_k|z, x$ from $\mathcal{N}(\hat{\mu}_k, \hat{\lambda}_k)$
  end
end

4 Application on synthetic data

4.1 Description

We now turn to fit the model to some generated data. In particular, we generate 50 observations coming from each of the 3 different bivariate Normal distributions, respectively centered in the origin, in $(1, 2)$ and in $(3, -4)$ of the Euclidean space, with the same isotropic covariance matrix, as shown in 1.

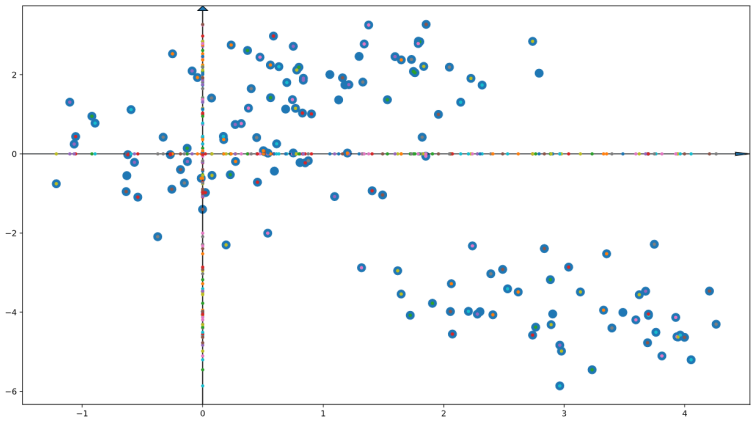

Fig. 1: Synthetic data coming from 3 different clusters.

4.2 Ergodicity and convergence diagnostics

Thanks to the convergence results, we know that, eventually, sampling a value from the generated Markov chain is equivalent to sampling a value from the invariant distribution. In addition, thanks to ergodicity, one can take more values from the same Markov chain as to obtain a sample from the distribution of interest.

However, one may want to check if the the hypotheses of such results are true. In particular, the convergence to the stationary distribution, the convergence of averages and the convergence to i.i.d. sampling must be assessed. In the following, trace plots and tests are provided after 1000 iterations of the algorithm.

Trace plots In Figure 2, the trace plots for the two component of the mean parameter for one of the cluster is shown. By graphical inspection, the chains seem to have reached stationarity. However, more formal checks are provided below.

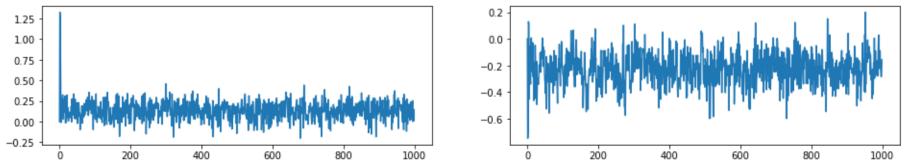

Fig. 2: Trace plots of the means of one cluster.

Effective sample size On the basis of the autocorrelation function, a Python function that computes the effective sample size has been implemented. The effective sample size is the number of independent Monte Carlo elements to get the same precision of the MCMC procedure [5], and it is computed according to Equation 5. In particular, the effective sample size oscillates between 320 and 340, suggesting that the number of iteration run might be sufficient for our purpose.

$$s_{eff} = \frac{s}{1 + 2\sum_{l=1}^{\infty} corr(X^t, X^{t+l})} \tag{5}$$

Kolmogorov-Smirnov test The Kolmogorov-Smirnov test is a nonparametric test for stationarity: it takes two samples and checks whether the two are generated from the same distribution. In our case, after a *thinning* procedure to reduce the autocorrelation of the chains [6], we divide the chains in three parts: the first part can be considered the *burn-in* phase; the second and the third segments are given as input to the test. This tests provides an evidence supporting or rejecting the stationarity of our chains.

The test has been conducted on the chains of μ and z at the end of the procedure. The p-value of the tests conducted, whose null hypothesis is that both chains come from the same distributions, are always high and above 30%, providing another evidence that the chain has reached stationarity. Similar results show for the cluster assignments.

4.3 Results

We consider the first third of the chain as the *burn-in*. Hence, we take the successive values in order to obtain the Bayesian estimates for the cluster assignments and for the mean vectors. In particular, we take the average of the values obtained from the chain and we use these values as our estimates, relying on the Ergodic theorem.

In the following graph, after having obtained the estimates of the variables of interest, 50 points for each cluster were sampled according to the generative process.

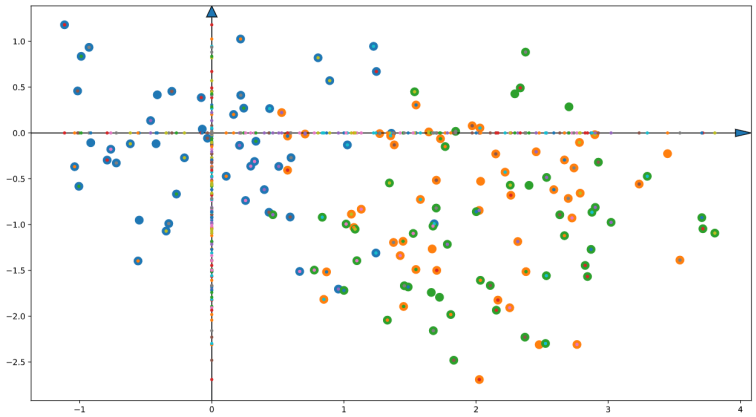

Fig. 3: Data generated with the estimated parameters (1000 iterations).

4.4 Implementation issues

Some issues may arise on the choice of the starting values of the chains. Indeed, this aspect has an impact on the mixing of chain, ultimately. Sometimes one may choose to start from the Maximum Likelihood estimate. another choice is to start from the prior: the cluster assignments have been initialized with values drawn from the prior distribution. Nevertheless, we can notice that even with different starting values, we obtain very similar results at the end of the procedure.

Another issue may arise in the evaluation of the convergence: the methods implemented above indicate quite clearly that the convergence has been reached. However, these kind of results are not always straightforward to interpret. One can then decide to deploy further tools based on the comparison of multiple chains, such as the Gelman-Rubin convergence diagnostics. However, such methods might be computationally demanding, as multiple chains need to be generated for each parameter. For more complex application, like the following, the opportunity cost of running such a test is high: indeed, one might simply prefer to run the chain for a longer time.

The Gibbs sampling is a very convenient method to deal with high-dimensional densities. However, particular attention should also be provided to the hyperparameters specification, which may severely affect the mixing time of the chain and its convergence.

5 Application to Image Segmentation

The second application is to show the great potential of this simple model. Image segmentation algorithms aim at partitioning a given image into homogeneous pattern classes. In the context of our model, this task is accomplished by assigning each pixel to one of the K possible classes, which corresponds to one of the K different mixture components. [7] [8]

5.1 Description

An image of $32x32$ pixels has been provided as our data set. Each pixel is characterized by 3 dimensions, that express the intensity of each of the RGB channels. For each channel, there are 256 possible color intensities, for a total of 256^3 possible combinations. In our application, 10 clusters are specified, which means that all the 256^3 colors will be grouped into 10 clusters. This application is critical also as a pre-processing step for data compression purposes, or to prepare an image data set for other machine learning algorithms.

Then, a new matrix that contains the estimated mean vectors of the clusters assigned to each pixel of the figure is retrieved. This allows us to represent the original picture and the cluster version.

The Python code that implements the image segmentation is also reported in the Appendix and slightly differs from the original one, since some modifications are required in order to deal with an image data input.

The following figures qualitatively show the improvement of the procedure with an increasing number of iterations.

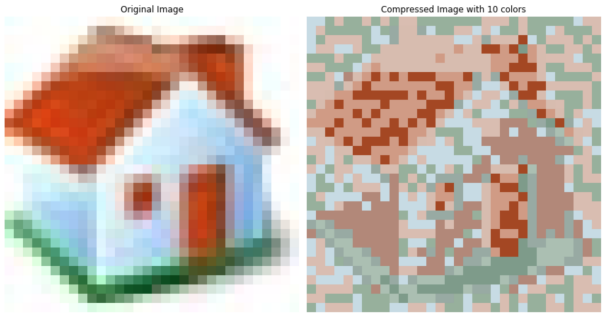

Fig. 4: Original and segmented image, after 10 iterations.

The same convergence diagnostic tools can be implemented in this application as well. In Figure 7, some trace plots of mean vectors are illustrated[1].

[1] Note that the trace plots show value between 0 and 1, rather than between 0 and 255. This is because input data were normalized to 1 and converted in floats prior to training. After that, the cluster means are converted back to an integer scale from 0 to 255.

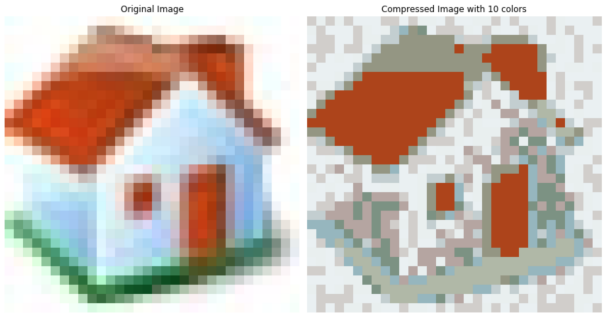

Fig. 5: Original and segmented image, after 100 iterations.

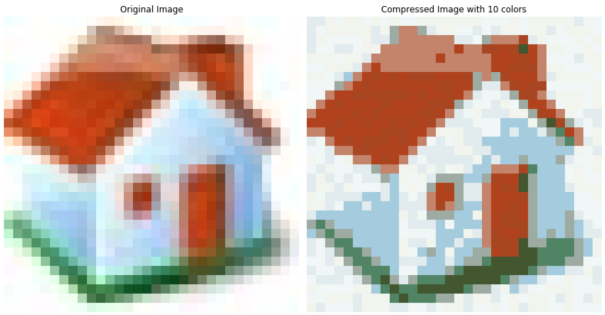

Fig. 6: Original and segmented image, after 1000 iterations.

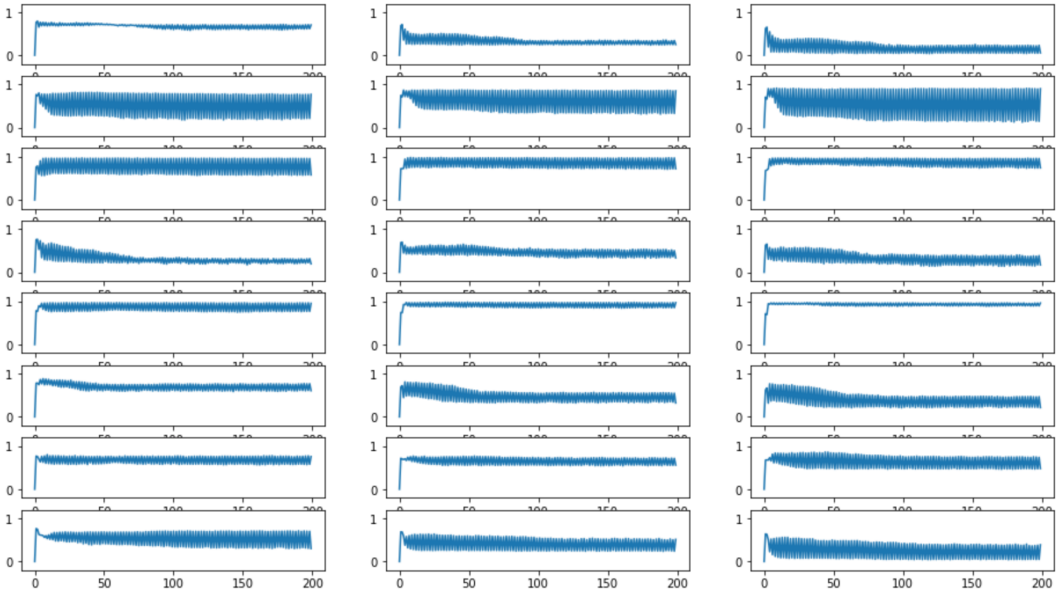

Fig. 7: Trace plots of mean vectors.

6 Python implementation

The algorithm and the applications have been implemented using Python3. The code is also available in the Appendix of this document.

References

1. Bishop, C.M.: Pattern recognition and machine learning. springer (2006)
2. Blei, D.M., Kucukelbir, A., McAuliffe, J.D.: Variational inference: A review for statisticians. Journal of the American statistical Association **112**(518), 859–877 (2017)
3. Casella, G., George, E.I.: Explaining the gibbs sampler. The American Statistician **46**(3), 167–174 (1992)
4. Handcock, M.S., Raftery, A.E., Tantrum, J.M.: Model-based clustering for social networks. Journal of the Royal Statistical Society: Series A (Statistics in Society) **170**(2), 301–354 (2007)
5. Hoff, P.D.: A first course in Bayesian statistical methods, vol. 580. Springer (2009)
6. Johansen, A.M., Evers, L., Whiteley, N.: Monte carlo methods. Lecture notes **200** (2010)
7. Park, J.H.: Color image segmentation using a model-based clustering and a mfa-em algorithm. In: Scandinavian Conference on Image Analysis. pp. 934–941. Springer (2003)
8. Yang, M.H., Ahuja, N.: Gaussian mixture model for human skin color and its applications in image and video databases. In: Storage and retrieval for image and video databases VII. vol. 3656, pp. 458–466. International Society for Optics and Photonics (1998)

Appendix

```
# -*- coding: utf-8 -*-
#
import pandas as pd
import numpy as np
import random
import scipy.stats
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt

#
k = 3 # clusters
# d = 2 # dimensions
prob = 1/k # uniform prior on cluster assignments
n = 150 # data points
iterations = 1000
lambdas = 1
sigma = 0.5 # note: sigma^2
```

```
18
19
20  # Generative process (synthetic data)
21  d = 2
22  z_true = np.zeros(shape = (n))
23  z_true = np.random.choice(range(k), p = np.repeat(prob, k), size =
    ↪  n)
24
25  mu_true = np.zeros(shape = (k,d))
26
27  mu_true[0] = np.array((0,0))
28  mu_true[1] = np.array((1,2))
29  mu_true[2] = np.array((3,-4))
30
31  x = np.zeros(shape = (n,d))
32
33  for i in range(n):
34      cluster_assignment = int(z_true[i])
35      x[i] = scipy.stats.multivariate_normal.rvs(mean =
        ↪  mu_true[cluster_assignment], cov = sigma * np.identity(d))
36
37
38
39  # Matrix declaration
40  z = np.zeros(shape = (n, iterations))
41  mu = np.zeros(shape = (k, d, iterations))
42
43  n_k = np.zeros(shape = (k, iterations))
44  k_bar = np.zeros(shape = (k,d, iterations))
45
46
47  # Graphical representation of synthetic data:
48  from pylab import *
49  import matplotlib.pyplot as plt
50
51  x1 = x[:,0]
52  x2 = x[:,1]
53
54  fig = plt.figure(figsize = (16,9))
55  ax = fig.add_subplot(111)
56
57  scatter(x1, x2, s=100 ,marker='o')
58
59  [ plot( [dot_x,dot_x] ,[0,dot_y], '.', linewidth = 3 ) for
    ↪  dot_x,dot_y in zip(x1,x2) ]
```

```
[ plot( [0,dot_x] ,[dot_y,dot_y], '.', linewidth = 3 ) for dot_x,dot_y in zip(x1,x2) ]

left,right = ax.get_xlim()
low,high = ax.get_ylim()
arrow( left, 0, right -left, 0, length_includes_head = True, head_width = 0.08 )
arrow( 0, low, 0, high-low, length_includes_head = True, head_width = 0.08 )

show()

# Initialization:
mu[:,:,0] = np.zeros(shape = (k,d)) # mixture locations
z[:,0] = np.random.choice(a = np.array((range(k))), p = np.repeat(prob, k), size = n)

# Gibbs sampler:
for it in range(iterations-1):

    for i in range(n):
        #cluster_assignment = int(z[i, it])
        prob_temp = np.zeros(shape = (3,1))
        for kk in range(k):
            prob_temp[kk] = prob * scipy.stats.multivariate_normal.pdf(x = x[i], mean = mu[kk,:,it], cov = sigma * np.identity(d))
        # print(f"Probabilities: {prob_temp}")
        prob_temp = prob_temp / prob_temp.sum()
        # print(f"Probabilities, normalized: {prob_temp}")
        z[i,it+1] = np.random.choice(a = np.array((0,1,2)), p = prob_temp[:,0])

    for kk in range(k):

        n_k[kk,it] = np.unique(z[:,it], return_counts=True)[1][list(np.unique(z[:,0], return_counts=True)[0]).index(kk)]
        for i in range(n):
            if z[i,it] == kk:
                k_bar[kk,:,it] += x[i]
        k_bar[kk,:,it] = k_bar[kk,:,it] / n_k[kk,it]

        mean_k = (n_k[kk,it] / sigma) / ((n_k[kk,it]/sigma) + 1/lambdas) * k_bar[kk,:,it]
```

```
95          cov_k = 1 / (n_k[kk,it]/sigma + 1/lambdas)
96
97          mu[kk,:,it+1] = scipy.stats.multivariate_normal.rvs(mean =
            ↪  mean_k, cov = cov_k)
98
99      print(f"Iteration {it} complete.")
100
101 # Burn-in and trimming of the relevant chains
102 burn_in = int(iterations/3)
103
104 z_trimmed = z[:,burn_in:]
105 mu_trimmed = mu[:,:,burn_in:]
106
107 # Bayesian estimates: mean of the approximate posteriors
108 z_est = z_trimmed.mean(axis = 1)
109 mu_est = mu_trimmed.mean(axis = 2)
110
111 # Sampling from the approximate posterior
112 x_sampl_1 = scipy.stats.multivariate_normal.rvs(mean =
    ↪  mu_est[0,:], cov = sigma * np.identity(2), size = 50)
113 x_sampl_2 = scipy.stats.multivariate_normal.rvs(mean =
    ↪  mu_est[1,:], cov = sigma * np.identity(2), size = 50)
114 x_sampl_3 = scipy.stats.multivariate_normal.rvs(mean =
    ↪  mu_est[2,:], cov = sigma * np.identity(2), size = 50)
115
116 # Graphical representation:
117 from pylab import *
118 import matplotlib.pyplot as plt
119
120 x11 = x_sampl_1[:,0]
121 x12 = x_sampl_2[:,0]
122 x13 = x_sampl_3[:,0]
123
124 x21 = x_sampl_1[:,1]
125 x22 = x_sampl_2[:,1]
126 x23 = x_sampl_3[:,1]
127
128 fig = plt.figure(figsize = (16,9))
129 ax = fig.add_subplot(111)
130
131 scatter(x11, x21, s=100 ,marker='o')
132 scatter(x12, x22, s=100 ,marker='o')
133 scatter(x13, x23, s=100 ,marker='o')
134
135 [ plot( [dot_x,dot_x] ,[0,dot_y], '.', linewidth = 3 ) for
    ↪  dot_x,dot_y in zip(x11,x21) ]
```

```
[ plot( [0,dot_x] ,[dot_y,dot_y], '.', linewidth = 3 ) for dot_x,dot_y in zip(x11,x21) ]

[ plot( [dot_x,dot_x] ,[0,dot_y], '.', linewidth = 3 ) for dot_x,dot_y in zip(x12,x22) ]
[ plot( [0,dot_x] ,[dot_y,dot_y], '.', linewidth = 3 ) for dot_x,dot_y in zip(x12,x22) ]

[ plot( [dot_x,dot_x] ,[0,dot_y], '.', linewidth = 3 ) for dot_x,dot_y in zip(x13,x23) ]
[ plot( [0,dot_x] ,[dot_y,dot_y], '.', linewidth = 3 ) for dot_x,dot_y in zip(x13,x23) ]

left,right = ax.get_xlim()
low,high = ax.get_ylim()
arrow( left, 0, right -left, 0, length_includes_head = True, head_width = 0.08 )
arrow( 0, low, 0, high-low, length_includes_head = True, head_width = 0.08 )

show()

# Trace plots of the means of three clusters
fig, ax = plt.subplots(figsize = (16,9), nrows = k, ncols = d)
for ax2 in range(ax.shape[1]):
    ax[ax1,ax2].plot(mu[0,ax2,:])

# Effective Sample Size
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import acf
def eff_sample_size(chain):
    '''
    Compute the Effective Sample Size for the MCMC provided
    '''
    dimen = len(chain)
    somma = np.sum(acf(chain))
    den = 1 + 2*somma
    ess = dimen/den

    return ess

# Showing ESS results for cluster means
for i in range(k):
    print(f"ESS (means cluster{i}): ",eff_sample_size(mu[i,0,:]), eff_sample_size(mu[i,0,:]))
```

```

# Kolmogorov-Smirnov nonparametric test
from scipy import stats
test_results = np.zeros(50)
effective_ss = []

mu_thinned = mu[:,:,::2] # thinning


for i in range(k):
    for j in range(d):
        burnin = mu_thinned[i,j,:int(len(mu_thinned[i,j,:])/3)]
        sample1 =
        ↪  mu_thinned[i,j,int(len(mu_thinned[i,j,:])/3):2*int(len(mu_thinned[i,j,:])/3)]
        sample2 = mu_thinned[i,j,2*int(len(mu_thinned[i,j,:])/3):]
    print(f"Test completed for cluster {i}")
    print(stats.ks_2samp(sample1, sample2))


z_thinned = z[:,::2]
for i in range(n):
    burnin = z_thinned[i,:int(len(z_thinned)/3)]
    sample1 =
    ↪  z_thinned[i,int(len(z_thinned)/3):2*int(len(z_thinned)/3)]
    sample2 = z_thinned[i,2*int(len(z_thinned)/3):]
print(f"Test completed for data point {i}")
print(stats.ks_2samp(sample1, sample2))

# ------- Application 2: image segmentation
↪  ----------------------------
import pandas as pd
import numpy as np
import random
import scipy.stats
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
import math


#
it = 0
k = 8 # clusters
# d = 2 # dimensions
prob = 1/k # uniform prior on cluster assignments
# n = 150 # data points
```

```
iterations = 200
lambdas = 0.003
sigma = 0.003 # note: sigma^2


#Image data
from matplotlib.image import imread
# img = imread('/Users/leonardo/Downloads/xp.jpg')
img = imread('home.jpg')


# img = imread('home.png')
img_size = img.shape
x = img.reshape(img_size[0] * img_size[1], img_size[2])
x = x.astype(float)

d = x.shape[1]
n = x.shape[0]

# Normalizing color intensities between 0 and 1
for dim in range(x.shape[1]):
    for i in range(x.shape[0]):
        x[i, dim] /= 255


#Matrix declaration
z = np.zeros(shape = (n, iterations))
mu = np.zeros(shape = (k, d, iterations))

n_k = np.zeros(shape = (k, iterations))
k_bar = np.zeros(shape = (k,d, iterations))


#Gibbs sampler:
# mu[:,:,0] = x.mean(axis = 0) # mixture locations
mu[:,:,0] = np.zeros(shape = (k,d))  # mixture locations
z[:,0] = np.random.choice(a = np.array((range(k))), p =
↪  np.repeat(prob, k), size = n)

for it in range(iterations-1):

    for i in range(n):
        # cluster_assignment = int(z[i, it])
        prob_temp = np.zeros(shape = (k,1))
        for kk in range(k):
```

```
259             prob_temp[kk] = prob *
                ↪  scipy.stats.multivariate_normal.pdf(x = x[i], mean
                ↪  = mu[kk,:,it], cov = sigma * np.identity(d))
260         # print(f"Probabilities: {prob_temp}")
261         prob_temp /= prob_temp.sum()
262         # print(f"Probabilities, normalized: {prob_temp}")
263         z[i,it+1] = np.random.choice(a = np.array(range(k)), p =
            ↪  prob_temp[:,0])
264
265     for kk in range(k):
266         if kk in list(np.unique(z[:,it], return_counts=True)[0]):
267             n_k[kk,it] = np.unique(z[:,it],
                ↪  return_counts=True)[1][list(np.unique(z[:,it],
                ↪  return_counts=True)[0]).index(kk)]
268         for i in range(n):
269             if z[i,it] == kk:
270                 k_bar[kk,:,it] += x[i]
271         k_bar[kk,:,it] = k_bar[kk,:,it] / n_k[kk,it]
272
273         mean_k = (n_k[kk,it] / sigma) / ((n_k[kk,it]/sigma) +
            ↪  1/lambdas) * k_bar[kk,:,it]
274         cov_k = 1 / (n_k[kk,it]/sigma + 1/lambdas)
275
276         mu[kk,:,it+1] = scipy.stats.multivariate_normal.rvs(mean =
            ↪  mean_k, cov = cov_k)
277
278     print(f"Iteration {it} complete.")
279
280 #Burn-in and trimming
281 burn_in = int(iterations/3)
282 mu_trimmed = mu[:,:,burn_in:]
283 mu_est = mu_trimmed[:,:,:].mean(axis=2)
284
285 x_compr = np.zeros(shape = (1024,d))
286 for i in range(x.shape[0]):
287     x_compr[i] = mu[int(z[i,it-1]),:,it] * 255
288 x_compr = x_compr.astype(int)
289
290 x_reshaped = x.reshape(img_size[0], img_size[1], img_size[2])
291 x_compr = x_compr.reshape(img_size[0], img_size[1], img_size[2])
292
293 #Drawing the final results
294 fig, ax = plt.subplots(1, 2, figsize = (12, 8))
295 ax[0].imshow(img)
296 ax[0].set_title('Original Image')
```

```
ax[1].imshow(x_compr)
ax[1].set_title(f'Segmented Image with {k} colors')
for ax in fig.axes:
    ax.axis('off')
plt.tight_layout();

#Plot of the means of three clusters
fig, ax = plt.subplots(figsize = (16,9), nrows = k, ncols = d)
for ax1 in range(ax.shape[0]):
    for ax2 in range(ax.shape[1]):
        ax[ax1,ax2].plot(mu[ax1,ax2,:])
        ax[ax1,ax2].set_ylim((-0.2,1.2))

#Original pixels, clsuter assignment and assigned cluster means
show_its = it-1
for i in range(int(x.shape[0]/20)):
    print(x[i], z[i,show_its], mu_est[int(z[i,show_its]),:])
```