Variational Methods and Expectation Propagation

Ceriani Paolo, Silva Luca

03/02/2020

1 Introduction

Bayesian Variational methods are a vast class of techniques for approximating integrals that are intractable, arising usually in Bayesian inference. Their are primarly used to provide an analytical approximation of the posterior distribution, in a easier family, or also to derive a lower bound for the marginal likelihood of the observed data. Given a fully Bayesian model, in which all the parameters involved are given prior distributions, it could be really interesting the study of quantities involving the posterior distribution of the parameters or the marginal distribution of the observed variables. Let's now formalise the setting: we'll denote by $\mathbf{Z}$ the set of all latent variables and similarly by $\mathbf{X}$ the set of observed variables. In the case of a N dimensional sample of iid data, $\mathbf{X} = \{x_1, ..., x_n\}$ and $\mathbf{Z} = \{z_1, ..., z_n\}$.

Our model will therefore specify the distribution of $p(\mathbf{X}, \mathbf{Z})$ through the definition of $p(\mathbf{X}|\mathbf{Z}) * p(\mathbf{Z})$. As said, we will focus primarly on the approximation of the posterior distribution $p(\mathbf{Z}|\mathbf{X})$ through mathematical approximation. We underline that this is only one of the possible approaches to the task: other common ones relies on stochastic approximations, such as MCMC, that could in theory generate exact results, given infinite computation resources, but in turn can require extensive use of resources.

2 Approximation of the posterior

It can be easily shown that we can decompose the logarithm of the marginal as:

$$ln(p(\mathbf{X})) = L(q) + KL(q||p)$$

where we define:

$$L(q) = \int q(\mathbf{Z}) ln \left(\frac{p(\mathbf{X}, \mathbf{Z})}{q(\mathbf{Z})} \right) d\mathbf{Z}$$

$$KL(q||p) = -\int q(\mathbf{Z}) ln \left(\frac{p(\mathbf{Z}|\mathbf{X})}{q(\mathbf{Z})} \right) d\mathbf{Z}$$

It can be noted that the way to maximise the log-likelihood of the data seen, is maximise the lower bound $L(q)$ by optimization with respect to the distribution $q(\mathbf{Z})$, which in turn is equivalent to minimizing the KL divergence: indeed the likelihood of the data doesn't depend on the function q and therefore the maximum of $L(q)$ can only occur whenever KL is minimized. Of course the minimum of KL will be whenever $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X})$, but for our purpose we shall suppose that working with the posterior is infeasible. What we will do in variational methods will be constraining the function q to belong to some simpler family of function, and then find the best q among that set.

2.1 Mean Field Approximation

Suppose we partition the elements of $\mathbf{Z}$ into disjoint groups of variables that we will denote as $\mathbf{Z}_i i = 1...m$. We also assume that the function q must factorise in:

$$q(\mathbf{Z}) = \prod_{i=1}^{m} q_i(\mathbf{Z}_i)$$

. Mind that the "partition" of Z is done at our own choice, but we are not assuming anything else over the form of the q_i's.

Let's now find among this family what are the q_i's for which the lower bound is bigger.

$$L(q) = \int \prod_{i=1}^{m} q_i \{ln(p(\mathbf{X}, \mathbf{Z})) - \sum_i ln(q_i)\} d\mathbf{Z}$$

$$= \int q_j \{ \int \prod_{i \neq j}^{m} q_i \, ln(p(\mathbf{X}, \mathbf{Z})) d\mathbf{Z}_i \} d\mathbf{Z}_j - \int q_j \, ln(q_i) \} d\mathbf{Z}_j + c$$

$$= \int q_j \ \ln \tilde{p}(\mathbf{X}, \mathbf{Z_j}) \, d\mathbf{Z}_j - \int q_j \, ln(q_i) d\mathbf{Z}_j + c$$

where

$$\ln \tilde{p}(\mathbf{X}, \mathbf{Z_j}) = \mathbb{E}_{i \neq j}[\ln(p(\mathbf{X}, \mathbf{Z})] + c_1$$

(c_1 is added so that the left member of the equality sums to one, and therefore is a distribution)

Noting that the last expression we obtained is nothing more than the negative Kullback Leibler Divergence between $q_j(\mathbf{Z}_j)$ and $\tilde{p}(\mathbf{X}, \mathbf{Z}_j)$, if we now suppose to keep $q_{i \neq j}$ fixed, and we maxize $L(q)$, then we simply have to minime the KL just found, so whenever $\ln(q_j(\mathbf{Z}_j)) = \ln(\tilde{p}(\mathbf{X}, \mathbf{Z_j}))$.

2.2 Approximation of a normal

In the following we illustrate an example that we also implemented in Python. The idea is to consider a simple problem for which the solution for the posterior exists in close form so to compare our results with the true ones. Let's now consider:

$$x_1, ...x_n | \mu, \tau \ \sim \ N(\mu, (\tau)^{-1})$$

$$\mu | \tau \ \sim \ N(\mu_0, (k_0 \tau)^{-} 1)$$

$$\tau \ \sim Gamma(a_0, b_0)$$

As we already Know, the distributions we chose as prior are conjugate for the Normal distribution, and therefore will lead to an easily computable posterior. In the case the posterior will be:

$$p(\mu,\tau|x_1,...,x_n) = p(\mu|\tau,x_1,...,x_n)\, p(\tau|x_1,...x_n)$$

$$\sim\ N(\frac{n}{n+k_0}\frac{\sum x_i}{n}+\frac{k_0}{n+k_0}\mu_0,(n\tau+k_0\tau)^{-1})\ Gamma(a_0+\frac{n}{2},b_0+\frac{1}{2}\sum(x_i-\overline{x})^2+\frac{k_0 n}{2(k_0+n)}(\overline{x}-\mu_0)^2)$$

If we now apply the theory of mean field methods, we seek for a factorized q of the form:

$$q(\mu,\tau) = q_\mu(\mu) * q_\tau(\tau)$$

such that

$$q_\mu(\mu) = \frac{exp(\mathbb{E}_{q_\tau}[\ln\, p(x_1,...,x_n,\mu,\tau)]}{\int exp(\mathbb{E}_{q_\tau}[\ln\, p(x_1,...,x_n,\mu,\tau)]\, d\mu}$$

and

$$q_\tau(\tau) = \frac{exp(\mathbb{E}_{q_\mu}[\ln\, p(x_1,...,x_n,\mu,\tau)]}{\int exp(\mathbb{E}_{q_\mu}[\ln\, p(x_1,...,x_n,\mu,\tau)]\, d\tau}$$

Now in our particular case things simplifies a bit, indeed:

$$\ln(p(x_1,...,x_n,\mu,\tau)) = \ln(p(x_1,...,x_n|\mu,\tau)) + \ln(p(\mu|\tau)) + \ln(\tau)$$

and substituting in the formula:

$$\ln(q_\mu(\mu)) = \mathbb{E}_{q_\tau}[\ln(p(x_1,...,x_n|\mu,\tau)) + \ln(p(\mu|\tau)) + \ln(\tau)]$$

$$= E_{q_\tau}[\ln(p(x_1,...,x_n|\mu,\tau)) + \ln(p(\mu|\tau))] + c$$

$$= \mathbb{E}_{q_\tau}[\frac{n}{2}\ln(\tau) - \frac{\tau}{2}\sum_{i=1}^{n}(x_i-\mu)^2 + \frac{1}{2}\ln(k_0\tau) - \frac{k_0\tau}{2}(\mu-\mu_0)^2] + c$$

$$= -\frac{\mathbb{E}_{q_\tau}[\tau]}{2}[\sum_{i=1}^{n}(x_i-\mu)^2 + k_0(\mu-\mu_0)^2] + c$$

and by the form of the logarithm of $q(\mu)_\mu$ one can reckon that this is the kernel of a normal distribution $\sim N(\mu_n,(\tau_n)^{-1})$ where:

$$\mu_n = \frac{k_0\mu_0 + \sum_{i=1}^{n} x_i}{k_0+n}$$

$$\tau_n = (k_0+n) * \mathbb{E}_\tau\tau$$

And equally for τ we have that:

$$\ln(q_\tau(\tau)) = \mathbb{E}_{q_\mu}[\ln(p(x_1, ..., x_n|\mu, \tau)) + \ln(p(\mu|\tau)) + \ln(\tau)]$$

$$\ln(q_\tau(\tau)) = \mathbb{E}_{q_\mu}[\frac{n}{2}\ln(\tau) - \frac{\tau}{2}\sum_{i=1}^{n}(x_i-\mu)^2 + \frac{1}{2}\ln(k_0\tau) - \frac{k_0\tau}{2}(\mu-\mu_0)^2 + (a_0-1)\ln(\tau) - b_0\tau] + c$$

$$= (a_0 - 1)\ln\tau - b_0\tau + \frac{1}{2}\ln\tau + \frac{n}{2}\ln\tau - \frac{\tau}{2}\mathbb{E}_{q_\mu}[\sum_{i=1}^{n}(x_i - \mu)^2 + k_0(\mu - \mu_0)^2] + c$$

Again, we can recognise the kernel of a gamma variable: Gamma(a_n, b_n), with:

$$a_n = a_0 + \frac{n+1}{2}$$

$$b_n = b_0 + \frac{1}{2}\mathbb{E}_{q_\mu}[\sum_{i=1}^{n}(x_i - \mu)^2 + k_0(\mu - \mu_0)^2]$$

$$= \ ... \ = b_0 + \frac{k_0}{2}(\mathbb{E}_{q_\mu}[\mu^2] + \mu_0^2 - 2\mathbb{E}_{q_\mu}[\mu]\mu_0) + \frac{1}{2}\sum_{i=1}^{n}(x_i^2 + \mathbb{E}_{q_\mu}[\mu^2] - 2 * \mathbb{E}_{q_\mu}[\mu]x_i)$$

It's interesting to underline that we didn't make any assumption on the form of the factors for q, but they arise from problem being considered.

Notice also that, being q_μ a normal $\sim N(\mu_n, (\tau_n)^{-1})$, it's moment are:

$$\mathbb{E}_{q_\mu}[\mu] = \mu_n$$

$$\mathbb{E}_{q_\mu}[(\mu - \mu_0)^2] = \frac{1}{\tau_n}$$

$$\mathbb{E}_{q_\mu}[\mu^2] = var(\mu) + (\mathbb{E}_{q_\mu}[\mu])^2 = \frac{1}{\tau_n} + \mu_n^2$$

Furthermore being q_τ a gamma $\sim Gamma(a_n, b_n)$:

$$\mathbb{E}_{q_\tau}[\tau] = \frac{a_n}{b_n}$$

The problem with this solution is that it's not explicit: indeed the distribution of q_μ depends on the expected value of q_τ and the latter is function of the first and second moments of the former. So the idea for the algorithm we implemented was:

1. Compute the constants $\mathbb{E}_{q_\mu}[\mu], \mathbb{E}_{q_\mu}[\mu^2], \mathbb{E}_{q_\tau}[\tau], a_n, \mu_n$
2. initialize τ_n to some arbitrary value
3. repeat the following until convergence:

(a) use current value of τ_n and the constants determined before to compute b_n

(b) use current value of b_n to compute τ_n

!!!! Convergence of this algorithm is guaranteed because the bound is convex with respect to each of the factor $q_i(\theta_i)\, i = 1, 2$

2.3 Our code

Following what written before, we implemented the code in python for the problem described. Below the script:

```
# we'll suppose data coming from N(5,1)
N = 1000
x_real = np.random.normal(5,1, N)

mu_0 = 3
a_0 = 2
b_0 = 3
k_0= 2

#initialize tau_n to some arbitrary value
tau_n = 1
control= True
n_iter = 0
```

After the inizialization of the parameters and hyperparameters, we'll compute the true posterior using the formulas above:

```
######################## TRUE POSTERIOR ########################

mean= np.mean(x_real)
tau_true_sample = np.random.gamma(a_0+N/2, np.power(b_0 + 1/2 *
    np.sum(np.power(x_real-mean,2)) + k_0*N/(2*(k_0+N))*(mean-mu_0)**2,-1), size=
    10000)
mu_true = np.sum(x_real)/(N+k_0)+ k_0 /(k_0+N)*mu_0
mu_t_sample=[]
for i in range(10000):
     tau_true = tau_true_sample[i]*(N+k_0)
    mu_t_sample.append(np.random.normal(mu_true, np.power(tau_true,-1)))
```

We now begin the iterative part of the algorithm, where we try to approximate the values of b_n and τ_n. Notice that for a_n and μ_n there is no need of approximation, being them completely known.

```
########################## APPROXIMATED #################
```

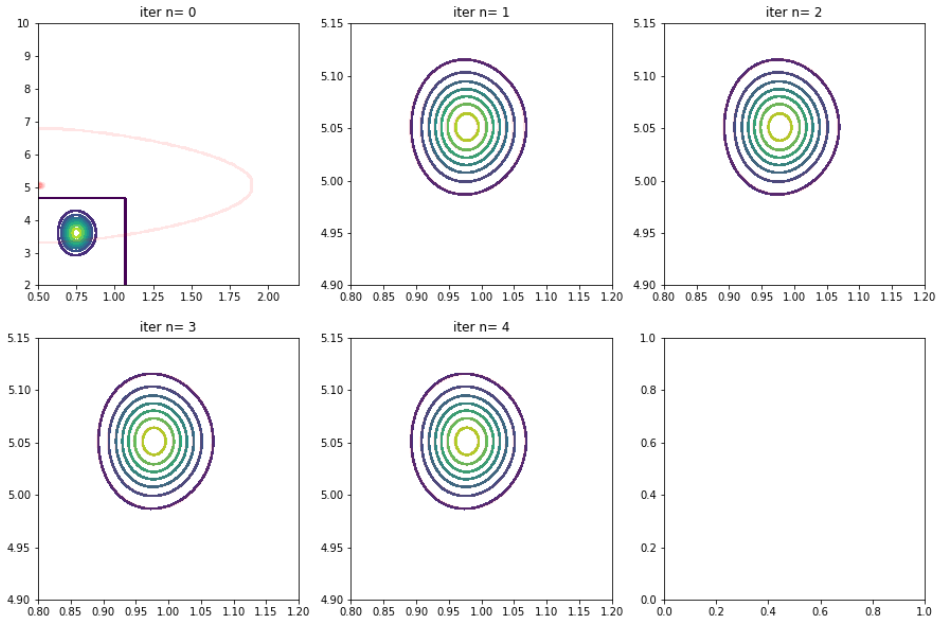
```
mu_n = (k_0*mu_0 + np.sum(x_real))/(k_0+N) # known
a_n = a_0 + (N+1)/2 #known

while(control):
    summation = np.sum(np.power(x_real,2))+N*(1/tau_n+
        mu_n**2)-2*mu_n*np.sum(x_real)
    b = b_0 + k_0/2 *(mu_n**2+1/tau_n+mu_0**2-2*mu_n*mu_0) + 1/2*summation

    tau = (k_0+N)*a_n / b
    if abs(tau_n - tau)<1e-5:
        if abs(b-b_n)<1e-5:
            control = False
    b_n = b
    tau_n = tau
    n_iter +=1
```

2.3.1 Notes on the code

There is nothing particularly interesting in the code itself: indeed, once the calculations are done, we only had to implement the exact passages described above. We could have initialized the parameters in a smarter way, for example estimating them from the original data ecc, but instead we deliberately choose the other way in order to check for convergence.

2.3.2 Comments on the results

After running the code, we obtained the results showed in the figure below. In red the contour plot of the approximated distribution, the other of the real posterior. The algorithm converges only in 5 iteration, and already from the beginning results are fantastic: this mainly depends on the form of the true posterior, indeed the algorithm tries to approximate the posterior as independent factors, but in this particular case there is also low correlation between $\mu|x_1, ..., x_n$ and $\tau|x_1, ..., x_n$ in the true posterior, therefore the algorithm works perfectly.

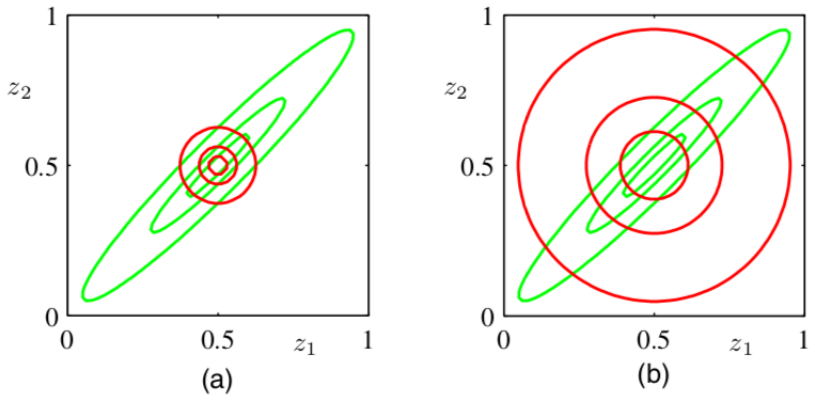


On the y axis μ (that being the true posterior, should somehow converge to $\mu = 5$, the mean value from where data were sampled), and on the x axis the value of τ (the precision, that again should approach 1).

Notice that this won't be the case when correlation is higher.

	mu	tau
mu	1	-0.0130225
tau	-0.0130225	1

We report also the contour plot of an attempt to approximate a highly correlated gaussian with 2 independent factors (on the left of the image the mean field approximation found minimizing $KL(q||p)$, on the left the result minimizing $KL(p||q)$ with constrained optimization)



Notice that the results differs from each other because of the asymmetry of the KL itself: whenever one wants to minimze $KL(q||p) = \int q(z) \ln\left(\frac{q(z)}{p(z)}\right)$, there is a large contribution to the value from region when p is small, unless q is smaller. Thus minimizing this form of the kl, will lead to distributions that avoids region where p is small. Conversely, $KL(p||q)$ will favour distributions that are non zero where p is non zero.

3 Variational Logistic Regression

We will now consider an application of mean field and variational bound to the logistic regression. The setting from now on will be the following: y ,the response variable coded in $\{-1, +1\}$, is assumed to be dependent on the D-dimensional input $\underline{x}$. The log likelihood ratio $\ln(\frac{P(y=1|\mathbf{X})}{P(y=-1|\mathbf{X})})$ is assumed to be linear in the input, such that we can retrieve the possibility of success as:

$$P(y = 1|x, w) = \frac{1}{1 + e^{(-w^t x)}} = \sigma(w^t x)$$

And equally:

$$P(y = -1|x, w) = 1 - P(y = 1|x, w) = \frac{1}{1 + e^{(w^t x)}} = \sigma(-w^t x)$$

Therefore

$$P(y|x, w) = \sigma(y w^t x)$$

Assuming some prior on the distribution of w, the aim is to find the form of the posterior $P(w|D)$ given some data $D = \{x_1, ..., x_n\}$. The problem is that the sigmoid data likelihood does not admit an easy conjugate prior. Therefore, approximations need to be applied to find an analytic expression for the posterior.

Being the approximation we will use quadratic in w in the exponential, it's therefore natural to use a gaussian prior for w and a Gamma prior for the precision, so to obtain a conjugate model. The hierarchical model is as follows :

$$P(w|x,\alpha) \sim N(\mathbf{0}, \alpha^{-1}\mathbf{I})$$

$$P(\alpha) = Gamma(a_0, b_0)$$

The idea as before, is to maximise a lower bound on the marginal data log-likelihood (that won't depend by definition on α and w):

$$\ln P(Y|X) = \ln \int \int P(Y|X,w)P(w|\alpha)P(\alpha)dw\, d\alpha$$

The lower bound will be

$$L(Q) = \int \int Q(w,\alpha)\, ln\left(\frac{P(Y,X,w,\alpha)}{Q(w,\alpha)}\right)$$

where the distribution $Q(w,\alpha)$ is approximating the posterior, indeed is found by minimising the KL between $P(w,\alpha|y,X)$ and $Q(w,\alpha)$ itself. We'll assume $Q(w,\alpha) = Q_w(w)\, Q_\alpha(\alpha)$ and apply what seen before.

Remember that this approximation will lead to analytic posterior expression if the model structure is conjugate exponential. But as we have seen, the sigmoid data likelihood doesn't admit a conjugate exponential prior, and therefore, using the lower bound found by Jaakkola and Jordan (2000) we have

$$\sigma(z) \geq \sigma(\xi) exp((z-\xi)/2 - \lambda(\xi)(z^2 - \xi^2)), \;\; \lambda(\xi) = \frac{1}{2\xi}(\sigma(\xi) - \frac{1}{2})$$

a tight lower bound on the sigmoid with one additional parameter ξ per datum.

After calculations you can obtain:

$$Q_w^*(w) \sim N(w_n, V_n)$$

$$Q_\alpha^*(\alpha) \sim Gamma(a_n, b_n)$$

where

$$a_n = a_0 + \frac{D}{2}$$

$$b_n = b_0 + \frac{1}{2} * \mathbb{E}_{q_w}[w^t w]$$

$$V_n^{-1} = \mathbb{E}_{q_\alpha}[\alpha] I + 2\sum_n \lambda(\xi_n) x_n^t x_n$$

$$w_n = V_n \sum \frac{y_n}{2} x_n$$

.

where x_n is the $(1 * D)$-vector of covariates for the n-th observation, ξ_n the variational parameter for each observation, V_n a $(D * D)$ matrix, w the weight $(D * 1)$ vector .

And also, for each observation:

$$\xi_n^{new} = \sqrt{x_n^t(V_n + w_n * w_n^t)x_n}$$

We write below the implementation of the code, we tested it on common dataset obtaining results comparable with the implementation done by packages in Python. One of the attentions required was in the definition of the function lambda, for small values:

```
def lambd(a):
    a = -abs(a)
    return 0.25 * exprel(a) / (np.exp(a) + 1) #to avoid problems with small a

def sigma(x):
    return 1/(1+np.e**(-x))


# the only quanity we can already implement
a_n = a_0 + D/2

b_n = 1 #for beginning
max_iter = 50
stop = False
n_iter = 0
V_n = np.identity(D)
b_n = 10
w_n = np.zeros((1,D))
while(not stop):
    A = np.zeros((D,D))
    summ = 0
    for i in range(n):
        A += np.transpose(X_t[i:i+1, ::]).dot(X_t[i:i+1, ::])*lambd(eps[i])
        summ += y_t[i]/2*X_t[i:i+1, ::]

    V = np.linalg.inv(a_n/b_n * np.identity(D)+ 2* A )
    w = V.dot(np.transpose(summ))
    b = b_0 + 0.5* (np.transpose(w).dot(w) + np.trace(V))
    n_iter +=1

    print(np.linalg.norm(V-V_n, 'fro'),np.linalg.norm(w-w_n), np.abs(b-b_n) )

    if(np.linalg.norm(V-V_n, 'fro')< 1e-3 and np.linalg.norm(w-w_n)< 1e-3 and
        np.abs(b-b_n)< 1e-3):
```

```
    stop= True
if (n_iter>max_iter):
    stop = True
V_n = V
w_n = w
b_n = b

H = V_n+w.dot(np.transpose(w))

for i in range(n):
    eps[i] = np.sqrt(X_t[i:i+1, ::].dot(H).dot(np.transpose(X_t[i:i+1, ::])))
```

4 Expectation Propagation

As with other Variational Bayes methods we have discussed so far, also *Expectation Propagation* tries to minimize the Kullback-Leibler divergence but this time of the reverse form. For many probabilistic models we have that:

$$p(D, \theta) = \prod (f_{\mathrm{i}}(\theta))$$

Like in the case of iid marginals given a value of the parameter generated from the prior which we include as f_0.The posterior hence then is simply

$$p(\theta|D) = \frac{1}{p(D)} \prod f_{\mathrm{i}}(\theta)$$

Expectation propagation is base on an approximation to the posterior distribution with is also given by a product of factors

$$q(\theta) = \frac{1}{Z} \prod \tilde{f}_{\mathrm{i}}(\theta)$$

Were each $\tilde{f}_{\mathrm{i}}$ is an approximation of f_{i} which we constrain by assuming it comes from the exponential family. Ideally we would like to define $\tilde{f}_{\mathrm{i}}$ by minimizing

$$KL(p||q) = KL\left(\frac{1}{p(D)} \prod f_{\mathrm{i}}(\theta) \,||\, \frac{1}{Z} \prod \tilde{f}_{\mathrm{i}}(\theta)\right)$$

In general this minimization will be intractable because it involves averaging with respect to the true distribution. EP starts by initializing the factors $\tilde{f}(\theta)$, and then cycles trough the factors refining them one at a time in the context of all the remaining factors, similarly to the VB method we have seen before.

5 Clutter Problem Algorithm

In the clutter problem we are trying to infer the mean of a group of Gaussian Distributed observations that are embedded in background clutter. When using the Expectation Propagation Algorithm in this case one has to be careful to some key steps of the implementation. In order to have some initial value for the mean and the covariance of the approximation, we want to initialize all of the $\tilde{f}_n(\theta)$ for n>1 such that $q(\theta) = p(\theta)$. To do so we have to se for every n :

$$\begin{cases} s_n = 1 \\ m_n = 0 \\ v_n = \infty \end{cases} \tag{1}$$

This initialization can create some numerical problems since it might cause to have situation in which we'd have to calculate a division for 0 or $0 \cdot \infty$, which would result the program to return a Nan and hence fail the whole optimization procedure from there on. These problems can first occur in function calculating cavities, whose code is the following:

```
// calculating cavities
def calculating_cavities(m_n, v_n, m, v):
    #cavity_v = v*v_n/(v_n-v)
    aux = 1/v - 1/v_n
    cavity_v = 1/aux
    cavity_m = m + cavity_v*(m-m_n)/v_n
    return cavity_m, cavity_v
```

Here we have to be careful with two aspects:

1)If aux is equal to zero then clearly we can't calculate cavity_v, hence in this case of this happening we just have to skip this observation for the moment and move to the next one, this will be accounted for simply by a control previous to calling the function.

2)If v_n is equal to ∞ then one has to be careful at how it calculates cavity_v. Infact if one executes some algebra then we easily get that $cavity\_n = \frac{v \cdot v_n}{(v_n - v)}$, and now if v_n where to be equal to ∞ then we'd encounter another Nan. Here there is no need to skip, indeed we actually initialize all of the v_n to ∞, we just have to split it as we did in the code so that we don't create a Nan.

Similar problems can arise when we are calculating the function new marginal, whose code is the following:

```
// new marginal
def new_marginal(update_m, update_v, cavity_m, cavity_v, z):
    aux=1/update_v-1/cavity_v
    if aux!=0:
        new_v=1/aux
    else:
```

```
        ##this remains the same
        new_v=np.inf
    if new_v==np.inf and (1/cavity_v == 0 or update_m==cavity_m):
        new_m=cavity_m
    else:
        new_m=cavity_m+(new_v+cavity_v)/(cavity_v)*(update_m-cavity_m)

    denominator=((2*np.pi*new_v)**(1/2)*normal_pdf(new_m, cavity_m,
        new_v+cavity_v));
    if denominator>0:
        new_s=z/denominator
    else:
        new_s=np.inf
    return new_v, new_m, new_s
```

As one can see here in the case where $aux = 0$, in order to dodge a division for zero we manually set $v_{\text{new}} = \infty$, while if in the calculation of m_{new} we encounter a $0 \cdot \infty$ we simply se $m_{\text{new}} = cavity_{\text{n}}$

After having taken care of these technical details, we can test our algorithm for a given set of parameters of the clutter problem.

```
// setting parameters
w=0.5;
clutter_mean=0;
clutter_var=1;
prior_mean=0;
prior_var=2.3;
var=1;
```

```
// generating the data
n_samples=50
x=np.zeros(n_samples)
#mean=np.random.normal(loc=prior_mean, scale=prior_var)
mean=2
print("the value of the true mean is: ",mean)
for i in range(n_samples):
    coin=np.random.random()
    if coin<w:
        x[i]=np.random.normal(loc=clutter_mean, scale=clutter_var)
    else:
        x[i]=np.random.normal(loc=mean, scale=var)
```

```
// code
thresh=1e-3
```

```
conv=False
iteration=0
while not conv and iteration<=100:
    iteration+=1
    m_old=np.copy(m_n)
    v_old=np.copy(v_n)
    s_old=np.copy(s_n)
    for i in range(len(x)):
        if v_n[i]!=v_posterior:
            cavity_m, cavity_v=calculating_cavities(m_n[i], v_n[i], m_posterior,
                v_posterior)
        else:
            #print("skipping i: ", i)
            continue

        z=normalizer(w, x[i], cavity_m, cavity_v, clutter_var)
        update_m, update_v = update(w, x[i], cavity_m, cavity_v, m_n[i], v_n[i],
            clutter_var)
        new_v, new_m, new_s = new_marginal(update_m, update_v, cavity_m,
            cavity_v, z)

        v_n[i]=new_v
        m_n[i]=new_m
        s_n[i]=new_s

        m_posterior, v_posterior = update_m , update_v
    conv=convergence(m_n, v_n, s_n, m_old, v_old, s_old, thresh)
```

5.1 Plots

Here we present some of the graphs of the previous code, we'll present only some of the iterations for conciseness.

As we can see for this setting of parameters, our algorithm estimates a posterior that converges very well to the true value of the parameter, that we had manually set to 2 and the approximation given by Expectation Propagation is very close to the one given by approximation trough numerical integration(see Fig1). Also we can see that the convergence is very quick (see Fig2)

5.2 Convergence Diagnostics

When creating the function that evaluates the convergence we have to be very careful at how we define it, in fact since we initialize every v_n to ∞ then we could encounter other indecision forms of the form $\infty - \infty$ if for instance we don't update a variance at a given step, also it is possible that some of the s_n get set to ∞ and hence we incur in the same

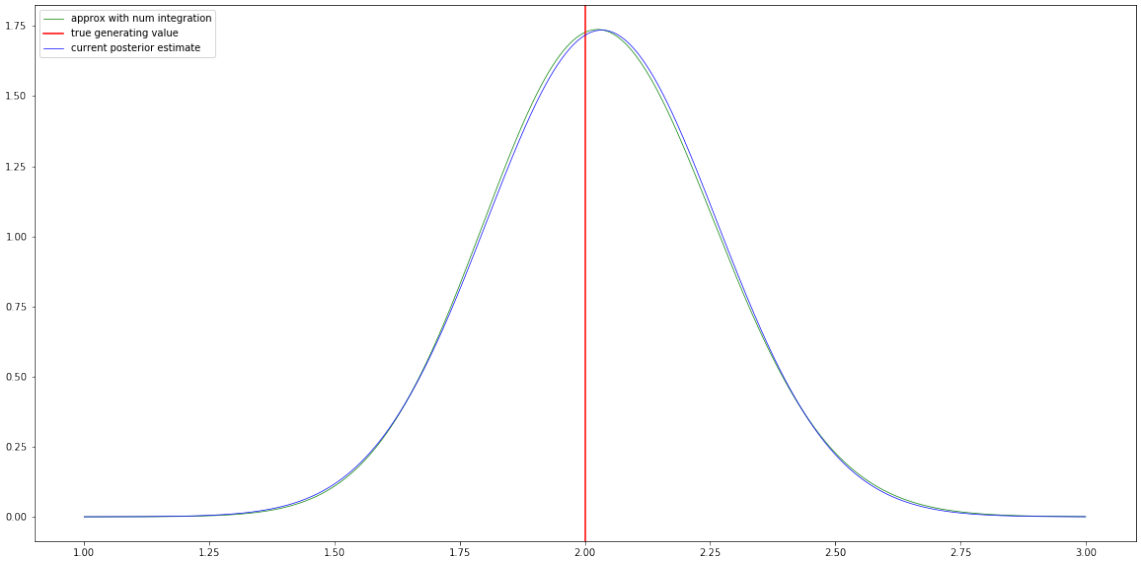
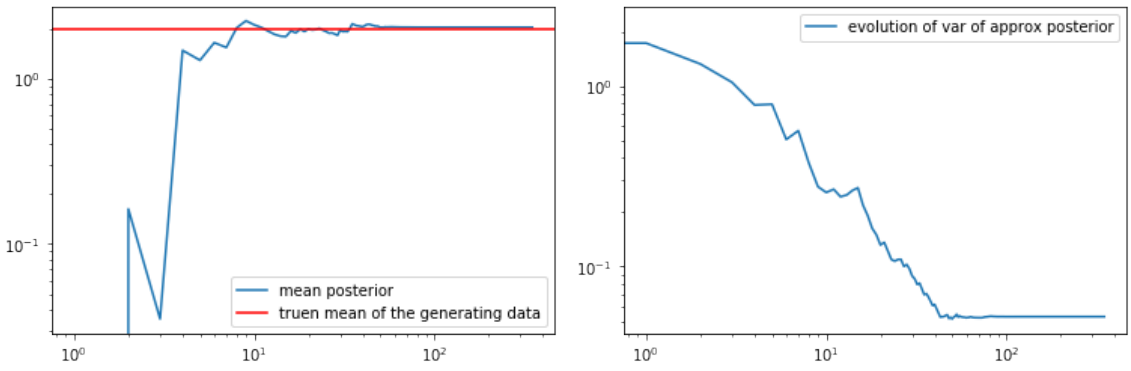


Figure 1: Final Approximation



(a)

Figure 2: Evolution of the Mean and Variance of Expectation Propagation

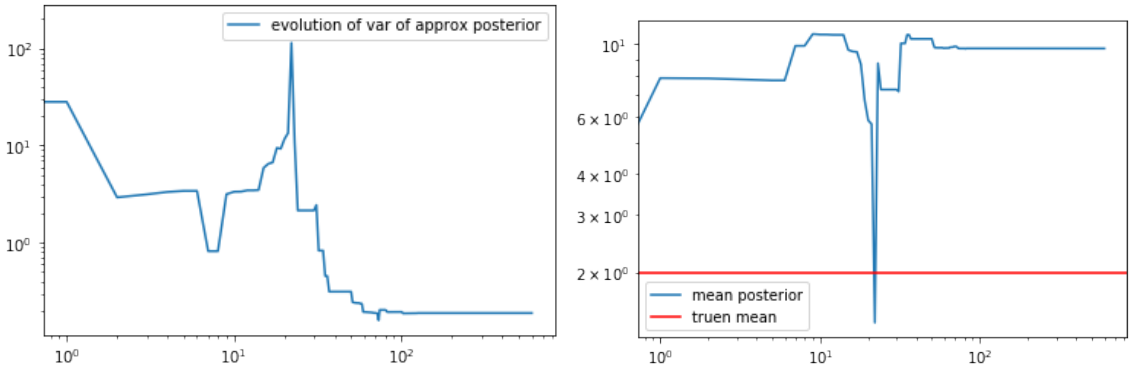
problem. To by pass this problem we simply manually set to zero that form of indecision if the program encounters it during the optimization, the code hence is the following:

```
//convergence function
def convergence(m_old, v_old, s_old, m_new, v_new, s_new, thresh):
    val_old=np.concatenate([m_old, v_old, s_old])
    val_new=np.concatenate([m_new, v_new, s_new])
    val=0
    for i in range(len(val_old)):
        if val_old[i]!=np.inf or val_new[i]!=np.inf:
            ##this way i have no form of indecision
            aux=np.abs(val_old[i]-val_new[i])
            if aux>val:
                val=aux
    if val<thresh:
        return (True, val)
    else:
        return (False, val)
```

As we know expectation propagation from *Minka et al 2001* is not guaranteed to converge, and when it does it converges to a fixed point of a given energy function. The problem is that it might happen that this energy function has multiple fixed point, and hence we converge to the wrong one. One could decide to find another optimization technique (still in the framework of Expectation Propagation), but generally we know that if EP doesn't converge to the true value than this just means that the chosen family of approximations is not good for the posterior we are dealing with. This is clear from the following example, we can see that we have failure of convergence (see Figure 3).

```
// setting parameters
w=0.5;
clutter_mean=0;
clutter_var=10;
prior_mean=0;
prior_var=200;
var=1;
```

Now if we plot the last approximation given by EP and we confront it with the one given by numerical approximation we clearly see that our posterior is far from gaussian centered in the true mean, and hence our approximation will never be able to be good in this case. The more the posterior looks like a normal centered in the true value, the better our approximation with EP will be, in fact we know by asympotic results that as the size of the sample increases the posterior tends to a normal centered in the parameter's true value with variance that goes to 0.



(a)

Figure 3: Evolution of the Mean and Variance of Expectation Propagation

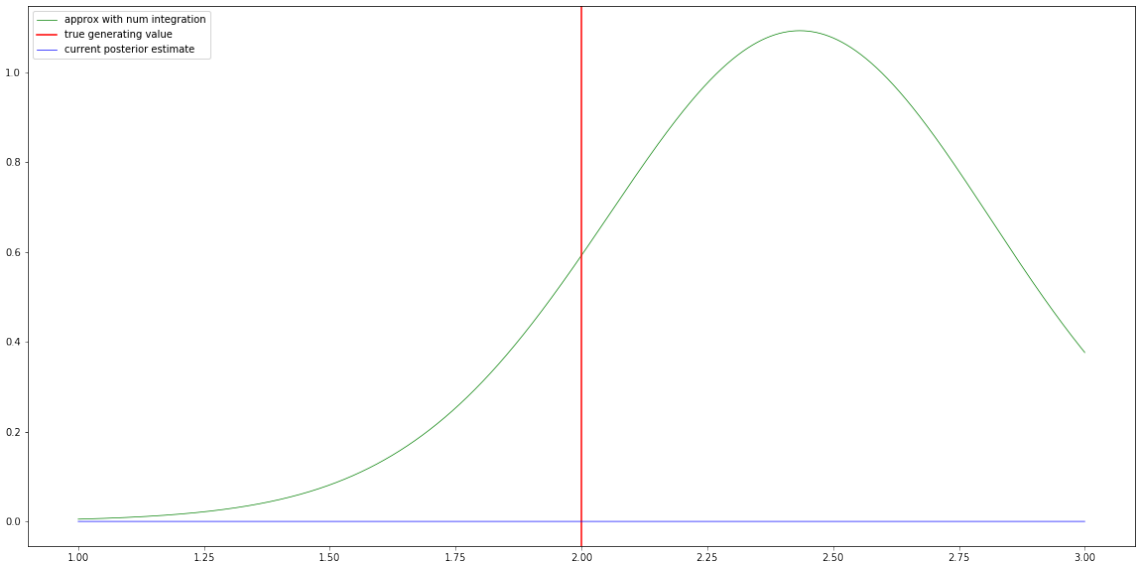


Figure 4: Final Approximation

5.3 Sampling from the posterior

In order to verify results coming from our algorithm, we decided to sample from the posterior of the data and plot the estimated pdf. We completed the task in two ways: using Gibb's Sampler and numerical approximation. Let's write the model in the hierarchical form, where we have added the hidden variables $z_1, ..., z_n$, one for each observation, giving information on which cluster the observation belongs:

$$X|z_1, ..., z_n, \mu \sim N(\mu, 1)$$

$$\mu \sim N(\mu_0, 1)$$

$$z_i \sim B(0.5)$$

5.3.1 Gibb's sampler

In order to apply Gibb's sampler, we have to compute the partial marginals:

$$P(x_1, ..., x_n, z_1, ..., z_n, \mu) = P(x_1, ..., x_n|z_1, ..., z_n, \mu)\, P(z_1, ..., z_n|\mu)\, P(\mu)$$

$$= \prod_i P(x_i|z_1, ..., z_n, \mu)\, P(z_1, ..., z_n)\, P(\mu)$$

And after basic computation one obtains, recalling that $P(x_i|z_1, ..., z_n, \mu) = P(x_i|z_i, \mu)$:

$$P(z_j|x_j, \mu) = \begin{cases} c\, N(x_j - \mu_j, 1)\, z = 1 \\ c\, N(0, 1)\, z = 0 \end{cases}$$

where c is the normalizing constant of the form:

$$c = \frac{1}{e^{-0.5(x_j - \mu_j)^2} + e^{-0.5(x_j)^2}}$$

And for μ, remembering that once conditioning on z, μ will depend only the x belonging to the cluster of interest (with z=1):

$$P(\mu|x_1, ..., x_n, z_1, ..., z_n) = P(\mu|D) \qquad D = \{x_i\, i = 1, ..., n\, |z_i = 1\}$$

and given the conjugate prior we chose, we obtain:

$$P(\mu|x_1, ..., x_n, z_1, ..., z_n) \sim N(\overline{\mu_n}, \overline{\sigma_n})$$

where they are equal at:

$$\overline{\mu_n} = \frac{n_1}{n_1 + k_0} \frac{\sum I_{z_i=1} x_i}{n_1} + \frac{k_0}{n_1 + k_0} \mu_0$$

$$\overline{\sigma_n} = (n_1\tau + k_0\tau)^{-1}$$

$$n_1 = \sum_i z_i$$

that is the number of x coming from the cluster of interest.

Sampling from these distribution it's easy, therefore we can implement the Gibbs sampler as follow:

1. initialize values of $\mu_n, z_1, ..., z_n$ (for the last we decided to use random inizialization)
2. for t going from 1 to s-1:
   (a) sample z_j^{t+1} from $p(z_j|\mu^t, x_1, ..., x_n)$ for $i = 1, ..., n$
   (b) sample μ^{t+1} from $p(\mu|z_1^{t+1}, ..., z_n^{t+1}, x_1, ..., x_n)$

The algorithm is as follows:

```
# setting parameters
n_iter = 1e4
mu_n = [1.6]*(int(n_iter)+1)
z_history = []
sigma_0 = 2
z_n = [round(i) for i in list(np.random.random(size = size_sample))]
z_history = np.zeros(( int(n_iter), size_sample))

# now update until convergence
for i in range(int(n_iter)-1):
    for j in range(size_sample):
        h = np.exp(-0.5*(x[j]-mu_n[i])**2)
        if np.random.rand() < np.power((h+np.exp(-0.5*np.power(x[j],2))),-1)*h:
            z_n[j] = 1
        else:
            z_n[j] = 0

    z_history[i]= z_n
    mu_n[i] = np.random.normal(sigma_0/(sigma_0 + 1/np.sum(z_n))*
        pd.Series(x)[pd.Series(z_n)==1].mean(), (np.sum(z_n)+1/sigma_0)**(-1))
```