# DD2434 - Machine Learning, Advanced Course

Assignment 1A

R. Darous darous@ug.kth.se

A. Jones arjjones@kth.se

December 11, 2023

In this assignment, we will cover an overview of the exponential family of distributions, the dependencies in a DGM, the Coordinate Ascent for Variational Inference, SVI for the LDA model and Black Box Variational Inference.

# Contents

# 1 Exponential family

A number of common distributions can be rewritten as exponential-family distributions with natural parameters, in the following form:

$$p(x \mid \boldsymbol{\theta}) = h(x) \exp\left(\boldsymbol{\eta}(\boldsymbol{\theta}) \cdot \boldsymbol{T}(x) - A(\boldsymbol{\eta})\right)$$

Below, we provide five different distributions from exponential-family. We will show that they correspond to common distributions.

## 1.1 Poisson distribution

We start using the following parameters :

- $\theta = \lambda$
- $\eta(\theta) = \log \theta$
- $h(x) = \frac{1}{x!}$
- $T(x) = x$
- $A(\eta) = e^{\eta}$

Then,

$$\begin{aligned} p(x|\theta) &= \frac{1}{x!} e^{\log \lambda x - e^{\log \lambda}} \\ &= \frac{1}{x!} \lambda^x e^{-\lambda} \end{aligned}$$

We recognize a **Poisson distribution** with parameter $\lambda$.

## 1.2 Gamma distribution

Secondly, we use the following parameters :

- $\boldsymbol{\theta} = [\alpha, \beta]$
- $\boldsymbol{\eta}(\boldsymbol{\theta}) = [\theta_1 - 1, -\theta_2]$
- $h(x) = 1$
- $T(x) = [\log x, x]$
- $A(\boldsymbol{\eta}) = \log \Gamma\left(\eta_1 + 1\right) - \left(\eta_1 + 1\right) \log\left(-\eta_2\right)$

Then,

$$\begin{aligned} p(x|(\alpha, \beta) &= 1 \cdot \exp\left( \begin{bmatrix} \alpha - 1 \\ -\beta \end{bmatrix} \cdot \begin{bmatrix} \log x \\ x \end{bmatrix} - \log \Gamma(\alpha) + \alpha \log(\beta) \right) \\ &= e^{(\alpha-1)\log x - \beta x - \log \Gamma(\alpha) + \alpha \log(\beta)} \\ &= \frac{x^{\alpha-1} c^{-\beta x} \beta^{\alpha}}{\Gamma(\alpha)} \end{aligned}$$

We recognize a **Gamma distribution** with parameters $(\alpha, \beta)$.

## 1.3 Gaussian distribution

Thirdly, we use the following parameters :

- $\boldsymbol{\theta} = \left[\mu, \sigma^2\right]$
- $\boldsymbol{\eta}(\boldsymbol{\theta}) = \left[\frac{\theta_1}{\theta_2}, -\frac{1}{2\theta_2}\right]$
- $h(x) = \frac{1}{\sqrt{2\pi}}$
- $\boldsymbol{T}(x) = \left[x, x^2\right]$
- $A(\boldsymbol{\eta}) = -\frac{\eta_1^2}{4\eta_2} - \frac{1}{2}\log\left(-2\eta_2\right)$

Then,

- $\eta(\theta) = \begin{pmatrix} \mu/\sigma^2 \\ -\frac{1}{2\sigma^2} \end{pmatrix}$
- $A(\eta) = \left(\frac{\mu}{\sigma}\right)^2 + \frac{1}{2}\log\left(\sigma^2\right)$

By injecting in the distribution, we get :

$$\begin{aligned} p(x|(\mu, \sigma^2) &= \frac{1}{\sqrt{2\pi}} \exp\left(\begin{pmatrix} \mu/\sigma^2 \\ -1/(2\sigma^2) \end{pmatrix} \cdot \begin{pmatrix} x \\ x^2 \end{pmatrix} - \left(\frac{\mu}{\sigma}\right)^2 + \frac{1}{2}\log\left(\sigma^2\right)\right) \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{1}{2\sigma^2}\left(2\mu x - x^2 - \mu^2\right)} \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \end{aligned}$$

We recognize a **Gaussian distribution** with parameters $(\mu, \sigma^2)$.

## 1.4 Exponential distribution

Fourthly, we use the following parameters :

- $\theta = \lambda$
- $\eta(\theta) = -\theta$
- $h(x) = 2$
- $T(x) = x$
- $A(\eta) = -\log(-\eta/2)$

Then,

$$\begin{aligned} p(x \mid \theta) &= 2e^{-\lambda x + \log(+1/2)} \\ &= 2\frac{1}{2}e^{-1x} \\ &= \lambda e^{-1x} \end{aligned}$$

We recognize an **Exponential distribution** with parameter $\lambda$.

## 1.5 Bêta distribution

Lastly, we use the following parameters :

- $\boldsymbol{\theta} = [\psi_1, \psi_2]$
- $\boldsymbol{\eta}(\boldsymbol{\theta}) = [\theta_1 - 1, \theta_2 - 1]$
- $h(x) = 1$
- $\boldsymbol{T}(x) = [\log x, \log(1-x)]$
- $A(\boldsymbol{\eta}) = \log\Gamma(\eta_1 + 1) + \log\Gamma(\eta_2 + 1) - \log\Gamma(\eta_1 + \eta_2 + 2)$

Then,

- $\eta(\theta) = (\psi_1 - 1, \psi_2 - 1)$
- $A(\eta) = \log\Gamma(\psi_1) + \log\Gamma(\psi_2) - \log\Gamma(\psi_1 + \psi_2)$

By injecting in the distribution, we get,

$$\begin{aligned} p(x \mid (\psi_1, \psi_2)) =& 1e^{(\psi_1 - 1)\log x + (\psi_2 - 1)\log(1-x)} e^{-\log\Gamma(\psi_1) - \log\Gamma(\psi_2) + \log\Gamma(\psi_1 + \psi_2).} \\ =& \frac{x^{\psi_1 - 1}(1-x)^{\psi_2 - 1}}{\Gamma(\psi_1)\Gamma(\psi_2)/\Gamma(\psi_1 + \psi_2)} \end{aligned}$$

We recognize a **Bêta distribution** with parameters $(\psi_1, \psi_2)$.

## 1.6 Conclusion

This gives an overview of the exponential family. We have seen that the Poisson, Gamma, Gaussian, Exponential and Bêta distributions are part from it.

## 2 Dependencies in a Directed Graphical Model

We now consider the graphical models shown in Figures 1 and 2. We will list some dependencies properties for those two Directed Graphical Models :

### 2.1 First DGM

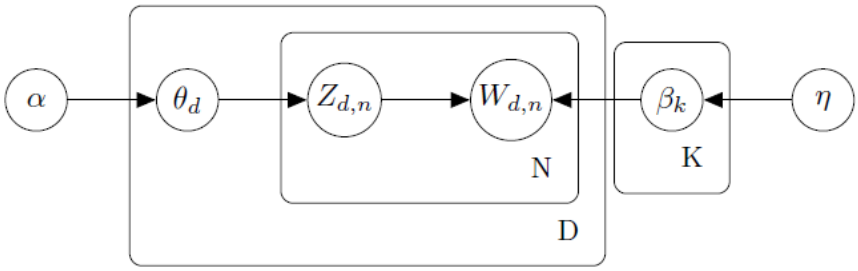



Figure 1: Graphical model of smoothed LDA.

1. $W_{d,n} \perp W_{d,n+1} \mid \theta_d, \beta_{1:K}$ is true,
2. $\theta_d \perp \theta_{d+1} \mid Z_{d,1:N}$ is false,
3. $\theta_d \perp \theta_{d+1} \mid \alpha, Z_{1:D,1:N}$ is true,

### 2.2 Second DGM

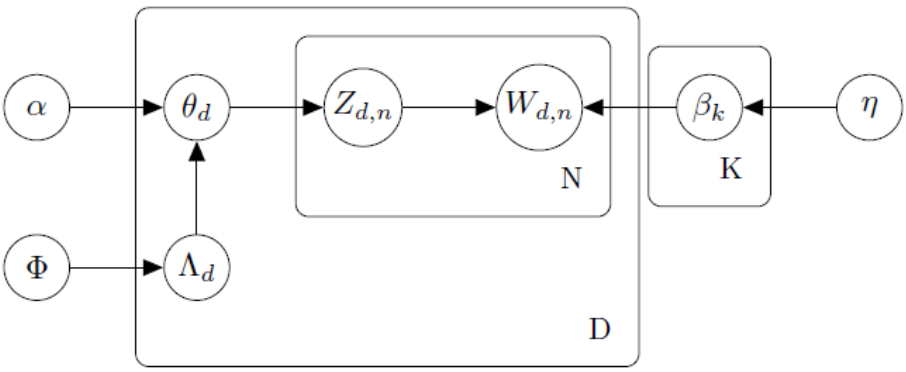



Figure 2: Graphical model of Labeled LDA.

1. $W_{d,n} \perp W_{d,n+1} \mid \Lambda_d, \beta_{1:K}$ is false,
2. $\theta_d \perp \theta_{d+1} \mid Z_{d,1:N}, Z_{d+1,1:N}$ is false,
3. $\Lambda_d \perp \Lambda_{d+1} \mid \Phi, Z_{1:D,1:N}$ is false.

# 3 CAVI

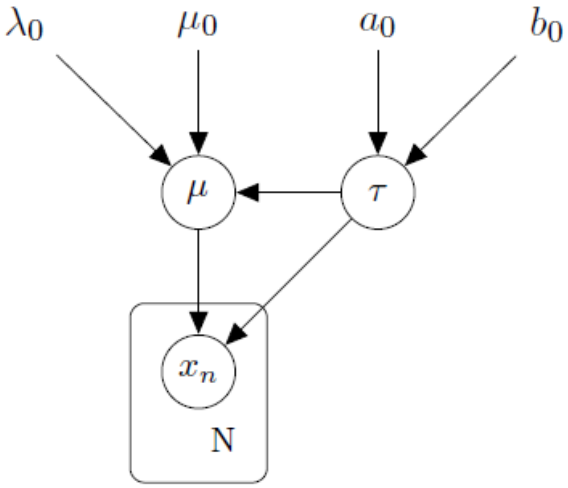



Figure 3: DGM

We now consider the model defined by Equation (10.21)-(10-23) in Bishop, for which DGM is presented in Figure 3. We are here concerned with the VI algorithm for this model covered during the lectures and in the book.

## 3.1 Sampling data

We start by implementing a function that generates data points for the given model. Data points are supposed to be sampled from a Gaussian with mean $\mu$ and precision $\tau$. We will use the parameters :

- $\mu = 1$
- $\tau = 0.5$
- datasets have size $N = 10, 100, 1000$.

The code is in the Annex. Below are plotted the histograms of the datasets :

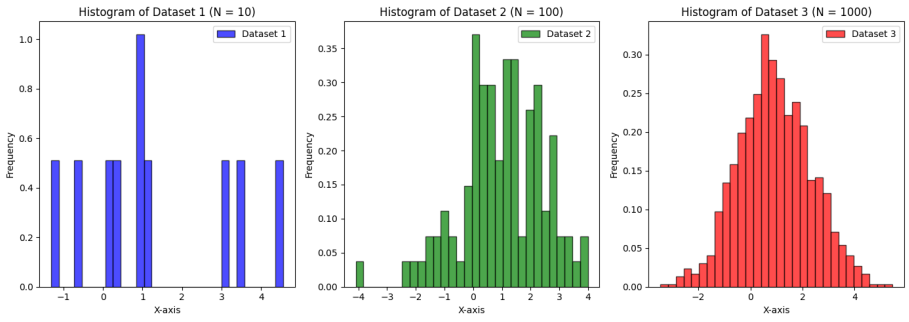



Figure 4: Plot of three samples of data points for the given model.

We can see that the more $N$ increases, the more the data distribution looks close to the underlying Gaussian distribution.

## 3.2 ML estimates of $\mu$ and $\tau$

Now, we want to find the ML estimates of $\mu$ $\tau$, using a MLE. According to the model defined by Equation (10.21)-(10-23) in Bishop, we get the following likelihood :

$$p(D \mid \mu, \tau) = \left(\frac{\tau}{2\pi}\right)^{N/2} e^{-\frac{\tau}{2}\sum_{n=1}^{N}(x_n-\mu)^2}$$

Taking the log, we get :

$$\log p(D \mid \mu, \tau) \stackrel{+}{=} \frac{N}{2}\log(\tau) - \frac{\tau}{2}\sum_{n=1}^{N}(x_n-\mu)^2$$

Now, let's compute the partial derivatives of the log-likelihood. **Note :** We assume $\tau > 0$, as it's a precision.

$$\begin{cases} \frac{\partial \mathcal{L}(\partial|\mu,\tau)}{\partial \mu} = \tau \sum_{n=1}^{N}(x_n-\mu) \\ \frac{\partial \mathcal{L}(\partial|\mu,\tau)}{\partial \tau} = \frac{N}{2\tau} - \frac{1}{2}\sum_{n=1}^{N}(x_n-\mu)^2 \end{cases}$$

To get the MLE, we must set the partial derivatives (i.e the gradient of the log-likelihood) to zero. We finally get that :

$$\begin{cases} \mu_{MLE} = \overline{x} \\ \tau_{MLE} = \frac{N}{\sum_{n=1}^{N}(x_n-\overline{x})^2} \end{cases}$$

where $\overline{x} = \frac{\sum_{n=1}^{N} x_n}{N}$.

On the data we have, we get the following MLE values :

- Dataset 1 : $\mu_{MLE} = 0.73,\ \tau_{MLE} = 0.36,$
- Dataset 2: $\mu_{MLE} = 1.24,\ \tau_{MLE} = 0.49,$
- Dataset 3 : $\mu_{MLE} = 0.99,\ \tau_{MLE} = 0.53.$

It corroborates the fact that the more data points we have, the closer we get to the true distribution of the data, as observed above.

## 3.3 Finding the exact posterior

As the given model remains rather simple, we can show that the posterior distribution has a closed form and takes the form of a Gaussian-gamma distribution.

According to the Exercices of Module 1 of the DD2423 course, we can show that the exact posterior follows a Gaussiam-gamma distribution with parameters :

- $a^* = a_0 + \frac{N-1}{2}$
- $b^* = b + \frac{1}{2}\left(\sum x_n^2 + \lambda_0\mu_0^2 - \frac{(\sum x_n + \mu_0\lambda_0)^2}{N+\lambda_0}\right)$
- $\lambda^* = \lambda_0 + N$
- $\mu^* = \frac{\sum x_n + \mu_0\lambda_0}{N+\lambda_0}$

## 3.4 Implementing the VI algorithm

We first compute the distribution of the joint probability of the model, given the DGM. It will be useful in computing the ELBO and the densities of $\mu$ and $\tau$ while performing variational inference.

$$p(D, \mu, \tau) = p(\tau)p(\mu|\tau)p(D|\mu, \tau)$$

### 3.4.1 Performing variational inference

To approximate the true posterior, we first use the mean field approximation :

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

The VI algorithm consists in **iteratively maximizing the ELBO with respect to each coordinate**.

Maximizing the ELBO with respect to $\mu$ gives, using the general result (10.9) from Bishop, the formula of the joint probability of the model computed above and the Bayes' rule :

$$\begin{aligned}\ln q_\mu^\star(\mu) &= \mathbb{E}_\tau[\ln p(\mathcal{D} \mid \mu, \tau) + \ln p(\mu \mid \tau)] + \text{const} \\ &= -\frac{\mathbb{E}[\tau]}{2}\left\{\lambda_0 (\mu - \mu_0)^2 + \sum_{n=1}^{N} (x_n - \mu)^2\right\} + \text{const.}\end{aligned}$$

We see that $q_\mu(\mu)$ is a Gaussian $\mathcal{N}\left(\mu \mid \mu_N, \lambda_N^{-1}\right)$ with mean and precision given by

$$\begin{aligned}\mu_N &= \frac{\lambda_0\mu_0 + N\bar{x}}{\lambda_0 + N} \\ \lambda_N &= (\lambda_0 + N)\,\mathbb{E}[\tau].\end{aligned}$$

Now, maximizing the ELBO with respect to $\tau$ gives :

$$\begin{aligned}\ln q_\tau^\star(\tau) =& \mathbb{E}_\mu[\ln p(\mathcal{D} \mid \mu, \tau) + \ln p(\mu \mid \tau)] + \ln p(\tau) + \text{ const} \\ =& (a_0 - 1)\ln\tau - b_0\tau + \frac{N+1}{2}\ln\tau \\ & - \frac{\tau}{2}\mathbb{E}_\mu\left[\sum_{n=1}^{N} (x_n - \mu)^2 + \lambda_0 (\mu - \mu_0)^2\right] + \text{ const}\end{aligned}$$

and hence $q_\tau(\tau)$ follows a gamma distribution $\text{Gam}\,(\tau \mid a_N, b_N)$ with parameters

$$\begin{aligned}a_N &= a_0 + \frac{N+1}{2} \\ b_N &= b_0 + \frac{1}{2}\mathbb{E}_\mu\left[\sum_{n=1}^{N} (x_n - \mu)^2 + \lambda_0 (\mu - \mu_0)^2\right] \\ &= b_0 + \frac{1}{2}\left(\sum x_n^2 + \lambda_0\mu_0^2 - 2\mathbb{E}_{q(\mu)}[\mu]\left(\lambda_0\mu_0 + \sum x_n\right) + (N + \lambda_0)\mathbb{E}_{q(\mu)}[\mu^2]\right)\end{aligned}$$

Where :

- $\mathbb{E}_{q(\mu)}[\mu] = \mu_N$
- $\mathbb{E}_{q(\mu)}[\mu^2] = \mu_N^2 + \frac{1}{\lambda_N}$

### 3.4.2 Computing the ELBO

To implement the VI algorithm, we need to be able to compute the ELBO, that we will use as a convergence criterion.

The ELBO is given by :

$$\begin{aligned}\mathcal{L}(q) &= \mathbb{E}_{q(\mu,\tau)}\left[\log \frac{p(\mu,\tau,D)}{q(\mu,\tau)}\right] \\ &= \mathbb{E}_{q(\tau)}[\log p(\tau)] + \mathbb{E}_{q(\mu,\tau)}[\log(p(\mu \mid \tau)] + \mathbb{E}_{q(\mu,\tau)}[\log(p(D \mid \mu,\tau)] \\ &- \mathbb{E}_{q(\mu)}[\log q(\mu)] - \mathbb{E}_{q(\tau)}[\log(q(\tau)]\end{aligned}$$

The last two terms correspond to the entropy of $\mu$ and $\tau$ (with respect to $q(\mu), q(\tau)$ respectively).

Let's denote $A(q) = \mathbb{E}_{q(\tau)}[\log p(\tau)] + \mathbb{E}_{q(\mu,\tau)}[\log(p(\mu \mid \tau)] + \mathbb{E}_{q(\mu,\tau)}[\log(p(D \mid \mu,\tau)]$. We have :

$$\begin{aligned}A(q) &\overset{+}{=} -b_0\mathbb{E}_{q(\tau)}[\tau] + (a_0-1)\mathbb{E}_{q(\tau)}\left[\log\tau\right] + \frac{1}{2}\mathbb{E}_{q(\tau)}\left[\log(\tau)\right] \\ &- \frac{1}{2}\mathbb{E}_{q(\mu,\tau)}\left[\lambda_0\tau\left(\mu-\mu_0\right)^2\right] + \frac{N}{2}\mathbb{E}_{q(\tau)}[\log(\tau)] - \frac{1}{2}\mathbb{E}_{q(\mu,\tau)}\left[\tau\sum_{n=1}^{N}\left(x_n-\mu\right)^2\right] \\ &\overset{+}{=} -b_0\mathbb{E}_{q(\tau)}[\tau] + \left(a_0 - \frac{1}{2} + \frac{N}{2}\right)\mathbb{E}_{q(\tau)}[\log(\tau)] \\ &- \frac{1}{2}\lambda_0\mathbb{E}_{q(\tau)}[\tau]\mathbb{E}_{q(\mu)}\left[\left(\mu-\mu_0\right)^2\right] - \frac{1}{2}\mathbb{E}_{q(\tau)}[\tau]\mathbb{E}_{q(\mu)}\left[\sum_{n=1}^{N}\left(x_n-\mu\right)^2\right] \\ &\overset{+}{=} \left(a_0 - \frac{1}{2} + \frac{N}{2}\right)\mathbb{E}_{q(\tau)}[\log(\tau)] \\ &- \mathbb{E}_{q(\tau)}[\tau]\left(b_0 + \frac{\lambda_0}{2}\left(\mathbb{E}_{q(\mu)}\left[\mu^2\right] - 2\mu_0\mathbb{E}_{q(\mu)}[\mu] + \mu_0^2\right)\right) \\ &- \frac{\mathbb{E}_{q(\tau)}[\tau]}{2}\left(\sum x_n^2 - 2\mathbb{E}_{q(\mu)}\left[\mu\right]\sum x_n + N\mathbb{E}_{q(\mu)}\left[\mu^2\right]\right)\end{aligned}$$

We admit that $\tau \sim \mathrm{Gam}\left(\tau \mid a_N, b_N\right)$ implies $\mathbb{E}_{q(\tau)}\left[\log\tau\right] = \psi(a_N) - \ln(b_N)$, where $\psi$ is the digamma function. Moreover, the results above give :

- $\mathbb{E}_{q(\tau)}[\tau] = \frac{a_N}{b_N}$
- $\mathbb{E}_{q(\mu)}[\mu^2] = \mu_N^2 + \frac{1}{\lambda_N}$

Finally, we have

$$\mathcal{L}(q) = A(q) + \mathbb{H}_{q(\mu)}[\mu] + \mathbb{H}_{q(\tau)}[\tau]$$

and we know how to compute all the terms using python libraries and formulas.

### 3.4.3 Applying the VI algorithm over the sampled datasets

We can now apply the VI algorithm over the sampled datasets.

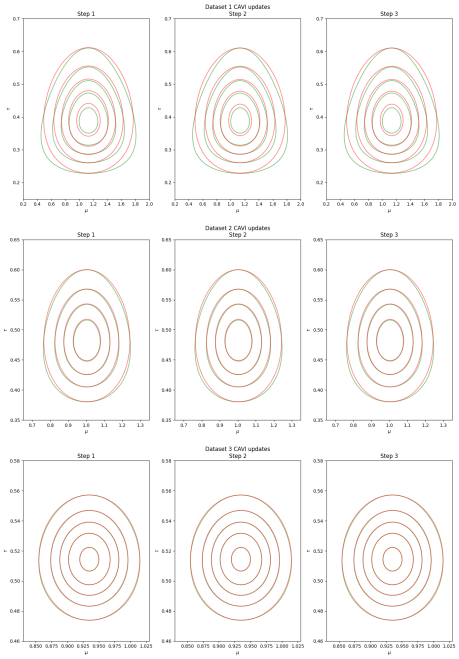



Figure 5: Three first steps of the VI algorithm for each dataset.

We can see that the estimated posterior **gets closer to the posterior as the number of data points increases**. The estimation seems rather accurate.

One may notice that the algorithm seems to converge rather quickly, as we don't see drastic changes between each iteration. To check this, let's take a look at the ELBO values through the iterations :

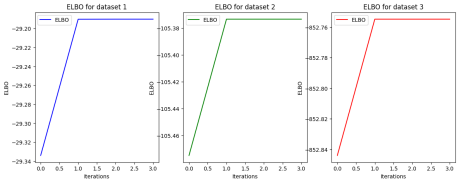
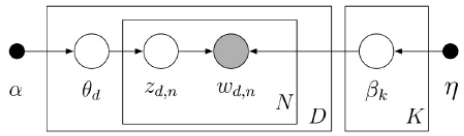



Figure 6: ELBO values given the iteration number for each dataset.

Indeed, after two steps, the ELBO seems to converge, which tells us that the **convergence rate is quick for this model**.

# 4 SVI - LDA




| Var | Type | Conditional | Param | Relevant Expectations |
|---|---|---|---|---|
| $z_{dn}$ | Multinomial | $\log \theta_{dk} + \log \beta_{k,w_{dn}}$ | $\phi_{dn}$ | $\mathbb{E}[Z^k_{dn}] = \phi^k_{dn}$ |
| $\theta_d$ | Dirichlet | $\alpha + \sum_{n=1}^{N} z_{dn}$ | $\gamma_d$ | $\mathbb{E}[\log \theta_{dk}] = \Psi(\gamma_{dk}) - \sum_{j=1}^{K} \Psi(\gamma_{dj})$ |
| $\beta_k$ | Dirichlet | $\eta + \sum_{d=1}^{D} \sum_{n=1}^{N} z^k_{dn} w_{dn}$ | $\lambda_k$ | $\mathbb{E}[\log \beta_{kv}] = \Psi(\lambda_{kv}) - \sum_{y=1}^{V} \Psi(\lambda_{ky})$ |

Figure 7: LDA DGM and conditional distributions (taken from Hoffman et al. 2013)

## 4.1 Local hidden variables

According to the Hoffman paper, local hidden variables $z_{1:N}$ are the latent variables that govern the observations $x_{1:N}$. In other words, the vector of the local hidden variables has the same shape as the vector of the observations, and one can map one local hidden variable to one observation.

The paper also gives that the joint distribution that factorizes into a global term and a product of local terms,

$$p(x, z, \beta \mid \alpha) = p(\beta \mid \alpha) \prod_{n=1}^{N} p\left(x_n, z_n \mid \beta\right).$$

## 4.2 Global and local hidden variables of the LDA model

In the LDA model the local hidden variables are the topic proportions $\theta_d$ and the topic assignments $z_{d,1:N}$. The global hidden variables are the topics $\beta_{1:K}$.

## 4.3 ELBO for the LDA model

Here is given the ELBO for the $LDA$ model as a function of variational parameters, $\phi_{dn}, \gamma_d, \lambda_k$, prior parameters, $\alpha, \eta$ and hyperparameters $D, N, K, W$:

$$\begin{aligned}
\mathcal{L} = &\sum_{d=1}^{D} \{\mathbb{E}_q\left[\log p\left(w_d \mid \theta_d, z_d, \beta\right)\right] + \mathbb{E}_q\left[\log p\left(z_d \mid \theta_d\right)\right] - \mathbb{E}_q\left[\log q\left(z_d \mid \theta_d\right)\right] \\
&+ \mathbb{E}_q\left[\log p\left(\theta_d \mid \alpha\right)\right] - \mathbb{E}_q\left[\log q\left(\theta_d\right)\right]\} \\
&+ \left(\mathbb{E}_q[\log p(\beta \mid \eta)] - \mathbb{E}_q[\log q(\beta)]\right) \\
= &\sum_{d=1}^{D} \{\mathbb{E}_q\left[\log p\left(w_d \mid \theta_d, z_d, \beta\right)\right] + \mathbb{E}_q\left[\log p\left(z_d \mid \theta_d\right)\right] - \mathbb{E}_q\left[\log q\left(z_d \mid \theta_d\right)\right] \\
&+ \mathbb{E}_q\left[\log p\left(\theta_d \mid \alpha\right)\right] + \mathbb{H}_q[\theta]\} \\
&+ \mathbb{E}_q[\log p(\beta \mid \eta)] + \mathbb{H}_q[\beta]
\end{aligned}$$

Where :

$$\mathbb{E}_q\left[\log p\left(w_d \mid \theta_d, z_d, \beta\right)\right] = \sum_{i=1}^{N}\sum_{k=1}^{K} \phi^k_{dw^i_d} \mathbb{E}_q\left[\log \beta_{kw^i_d}\right]$$

$$\mathbb{E}_q\left[\log p\left(z_d \mid \theta_d\right)\right] = \sum_{n=1}^{N}\sum_{k=1}^{K} \phi^k_{dw^i_d} \mathbb{E}_q\left[\log \theta_{dk}\right],$$

$$\mathbb{E}_q\left[\log q\left(z_d\right)\right] = \sum_{n=1}^{N}\sum_{k=1}^{K} \phi^k_{dw^i_d} \log \phi^k_{dw^i_d}$$

$$\mathbb{E}_q\left[\log p\left(\theta_d \mid \alpha\right)\right] = \log\Gamma(K\alpha) - K\log\Gamma(\alpha) + (\alpha - 1)\sum_{k=1}^{K}\log\theta_{dk}$$

$$\mathbb{E}_q[\log p(\beta \mid \eta)] = K[\log\Gamma(W\eta) - W\log\Gamma(\eta)] + \sum_{k=1}^{K}\sum_{w=1}^{W}(\eta - 1)\mathbb{E}_q\left[\log\beta_{kw}\right]$$

This formula was taken from **https://joshnguyen.net/posts/vb-lda**. Formula were slightly changed to incorporate the entropy in the final result and expressions that were $N$-dependant, without introducing new notations.

## 4.4 SVI implementation

In the Annex is given the code that gives the Three examples of datasets are studied :

- A tiny dataset (D = 50, N = 50, K = 2, W = 5)
- A small dataset (D = 1000, N = 50, K = 3, W = 10)
- A medium dataset (D = 10 000, N = 500, K = 5, W = 10)

### 4.4.1 Tiny dataset

Here is plotted the ELBO evolution of both CAVI and SVI algorithms. We can notice that despite local randomness inherent to the stochastic gradient descent algorithms, the ELBO of the SVI algorithm tends to increase across iterations. However, it increases slower that the one of the CAVI and does not reach as high values.

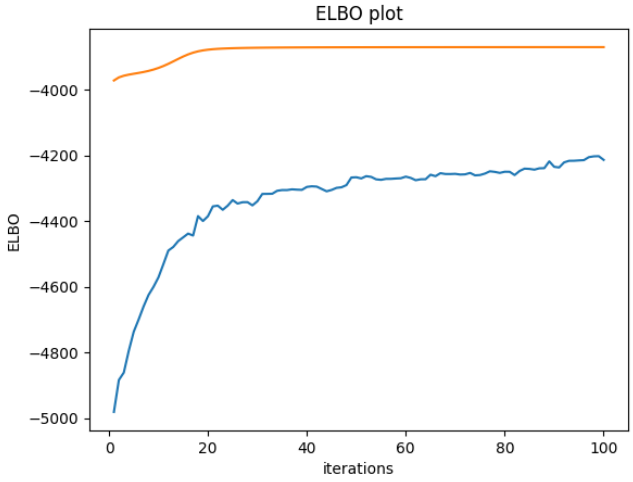



Figure 8: ELBO evolution across iterations. In orange, this is the CAVI algorithm ELBO. In blue, the SVI algorithm ELBO.

### 4.4.2 Small dataset

Here, the amount of data increased gives a smoother ELBO increase for the SVI algorithm. In addition, we can observer that it gets closer to the ELBO of the CAVI algorithm.

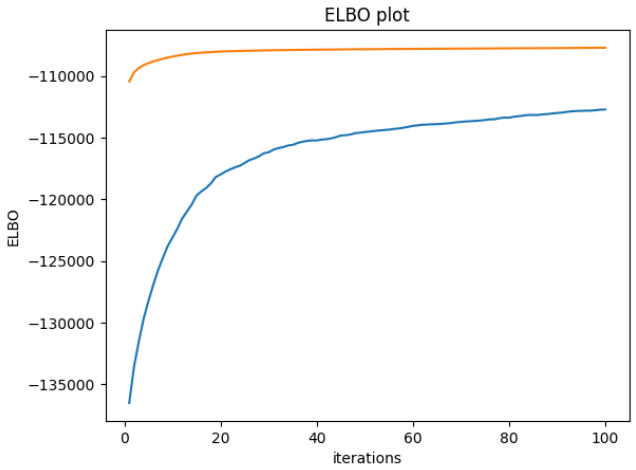



Figure 9: ELBO evolution across iterations. In orange, this is the CAVI algorithm ELBO. In blue, the SVI algorithm ELBO.

A very powerful asset of the SVI algorithm is its good results combined to a short running time. Let's check this using a bigger dataset.

### 4.4.3 Medium dataset

For the last dataset, only one iteration is performed to compare the run time. We get the following results :

- Time SVI: 2.9462718963623047 s,
- Time CAVI: 100.22320795059204 s.

Each iteration is approximately **30** times faster using SVI implementation, which makes it a lot more time efficient.

# 5 BBVI

In BBVI without Rao-Blackwellization and control variates, the gradient is estimated using Monte-Carlo sampling, the score function of q and the joint of p.

## 5.1 Gradient estimate of a simple model

For the simple model, $X \mid \theta, \sigma^2 \sim \mathcal{N}\left(\theta, \sigma^2\right), \theta \sim \text{Gamma}(\alpha, \beta)$ and $\sigma^2$ fixed, we will derive the gradient estimate w.r.t. $\nu$ using one sample $z \sim q(\theta), q(\theta) = \log \text{Normal}\left(\nu, \epsilon^2\right)$.

We only have one sample, so $\nabla_\nu \mathcal{L}$ is given by :

$$\nabla_\nu \mathcal{L} = \nabla_\nu \log q(\theta \mid \nu)(\log p(X, \theta) - \log q(\theta \mid \nu))$$

First, we have that

$$\begin{aligned}
\log q(\theta \mid \nu) &= \log\left(\frac{1}{\theta\epsilon\sqrt{2\pi}} e^{-\frac{(\log(\theta)-\nu)^2}{2\epsilon^2}}\right) \\
&= -\frac{1}{2}\log(2\pi) - \log(\epsilon) - \log(\theta) - \frac{(\log(\theta)-\nu)^2}{2\epsilon^2}
\end{aligned}$$

Hence, we have that :

$$\nabla_\nu \log q(\theta \mid \nu) = \frac{\log(\theta) - \nu}{\epsilon^2}$$

Then,

$$\begin{aligned}
\log p(X, \theta) - \log q(\theta \mid \nu) &= \log p(X \mid \theta) + \log p(\theta) - \log q(\theta \mid \nu) \\
&= \log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{(X-\theta)^2}{2\sigma^2} \\
&+ \log\left(\frac{\beta}{\Gamma(\alpha)}\right) - \beta\theta + (\alpha - 1)\log(\beta\theta) \\
&- \log\left(\frac{1}{\theta\epsilon\sqrt{2\pi}}\right) + \frac{(\log(\theta)-\nu)^2}{2\epsilon^2} \\
&= \log\left(\frac{\beta^\alpha \theta\epsilon}{\sigma\Gamma(\alpha)}\right) + (\alpha-1)\log(\theta) - \beta\theta + \frac{(\log(\theta)-\nu)^2}{2\epsilon^2} - \frac{(X-\theta)^2}{2\sigma^2}
\end{aligned}$$

Putting the two computations together gives us the **final result** :

$$\nabla_\nu \mathcal{L} = \frac{\log(\theta) - \nu}{\epsilon^2}\left(\log\left(\frac{\beta^\alpha \theta\epsilon}{\sigma\Gamma(\alpha)}\right) + (\alpha-1)\log(\theta) - \beta\theta + \frac{(\log(\theta)-\nu)^2}{2\epsilon^2} - \frac{(X-\theta)^2}{2\sigma^2}\right)$$

## 5.2 Role of Control variates in the BBVI paper

In the BBVI paper are used **Control variates** Control variates are used to reduce the variance of the estimator of the gradient in the BBVI algorithm.

# 6 Annex

## 6.1 Code for Section 3: CAVI

December 11, 2023

# 1 *Assignment 1.3 - CAVI*

Consider the model defined by Equation (10.21)-(10-23) in Bishop, for which DGM is presented below:

### 1.0.1 *Question 1.3.12:*

Implement a function that generates data points for the given model.

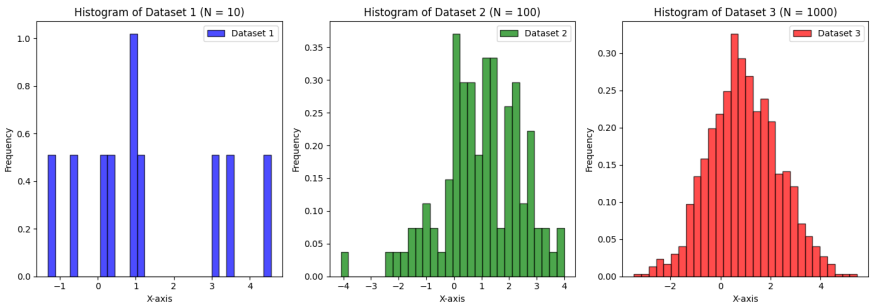
[37]:
```
import numpy as np
import numpy.random as np_rand
import matplotlib.pyplot as plt
import scipy.special as sp_spec
from scipy.stats import norm as norm
from scipy.stats import gamma as gamma
np.random.seed(345)
```

[38]:
```
# Generate datapoints that have a normal distribution with mean mu and precision␣
↪tau
def generate_data(mu, tau, N):
  data = np_rand.normal(mu, 1/np.sqrt(tau), N)
  return data
```

Set $\mu = 1$, $\tau = 0.5$ and generate datasets with size N=10,100,1000. Plot the histogram for each of 3 datasets you generated.

[39]:
```
# generate data
mu, tau = 1, 0.5
dataset_1 = generate_data(mu, tau, 10) # N = 10
dataset_2 = generate_data(mu, tau, 100) # N = 100
dataset_3 = generate_data(mu, tau, 1000) # N = 1000


fig, axs = plt.subplots(1, 3, figsize=(14, 5))

# Plot histogram for dataset 1
```

```
axs[0].hist(dataset_1, bins=30, density=True, alpha=0.7, color='blue',␣
 ↪edgecolor='black', label='Dataset 1')
axs[0].set_title('Histogram of Dataset 1 (N = 10)')
axs[0].set_xlabel('X-axis')
axs[0].set_ylabel('Frequency')
axs[0].legend()

# Plot histogram for dataset 2
axs[1].hist(dataset_2, bins=30, density=True, alpha=0.7, color='green',␣
 ↪edgecolor='black', label='Dataset 2')
axs[1].set_title('Histogram of Dataset 2 (N = 100)')
axs[1].set_xlabel('X-axis')
axs[1].set_ylabel('Frequency')
axs[1].legend()

# Plot histogram for dataset 3
axs[2].hist(dataset_3, bins=30, density=True, alpha=0.7, color='red',␣
 ↪edgecolor='black', label='Dataset 3')
axs[2].set_title('Histogram of Dataset 3 (N = 1000)')
axs[2].set_xlabel('X-axis')
axs[2].set_ylabel('Frequency')
axs[2].legend()

plt.tight_layout()
#plt.savefig('../Images/histograms.png')
plt.show()
```




### 1.0.2 *Question 1.3.13:*

Find ML estimates of the variables μ and τ

[40]:
```
def ML_est(data):
  N = data.shape[0]
```

```
  mu_ml = np.mean(data)
  tau_ml = N/np.sum((data - mu_ml)**2)
  return mu_ml, tau_ml

# Printing ML mean and precision estimates for each dataset
print('Mean and precision for dataset 1: ', ML_est(dataset_1))
print('Mean and precision for dataset 2: ', ML_est(dataset_2))
print('Mean and precision for dataset 3: ', ML_est(dataset_3))
```

```
Mean and precision for dataset 1:  (1.266028105946296, 0.3144166580382695)
Mean and precision for dataset 2:  (1.004280422073872, 0.48794889952614123)
Mean and precision for dataset 3:  (0.9343529824279556, 0.5157473861094543)
```

The more the number of datapoints increases, the closer we get to the real values of parameters $\mu$ and $\tau$.

### 1.0.3 *Question 1.3.14:*

You will implement the VI algorithm for the variational distribution in Equation (10.24) in Bishop. Start with introducing the prior parameters:

[41]:
```
# prior parameters
mu_0 = 1
lambda_0 = 10
a_0 = 10
b_0 = 20
```

Continue with a helper function that computes ELBO:

[42]:
```
def compute_elbo (D, a_0, b_0, mu_0, lambda_0, a_N , b_N, mu_N, lambda_N):
  # given the prior and posterior parameters together with the data,
  # compute ELBO here (up to a constant)

  # computing useful expectation value (given the distribution q(mu, tau))
  E_tau = a_N / b_N
  E_mu2 = 1/lambda_N + mu_N**2
  E_log_tau = sp_spec.digamma(a_N) - np.log(b_N)

  # computing entropies
  entropy_q = norm.entropy(loc=mu_N, scale=1/np.sqrt(lambda_N))
  entropy_tau = gamma.entropy(a_N, scale=1/b_N)

  # computing the data terms
  data_sum = np.sum(D)
  data_square_sum = np.sum(D**2)
  N = D.shape[0]

  # computing the ELBO
  elbo = - E_tau * (b_0 + lambda_0 * ( E_mu2 - 2 * mu_0 * mu_N + mu_0**2) / 2) \
```

```
    -(1/2) * E_tau * (data_square_sum - 2 * data_sum * mu_N + N * E_mu2) \
    + (a_0 - 1/2 + N/2)*E_log_tau \
    + entropy_q + entropy_tau
  """
  elbo = - E_tau * (b_0 + lambda_0 * ( E_mu2 - 2 * mu_0 * mu_N + mu_0**2)) \
    -(1/2) * E_tau * (data_square_sum - 2 * data_sum * mu_N + N * E_mu2) \
    + (a_0 - 1/2 + N/2)*E_log_tau \
    + entropy_q + entropy_tau
  """


  return elbo
```

Now, implement the CAVI algorithm:

[43]:
```
def CAVI(D, a_0, b_0, mu_0, lambda_0, n_iter):
  N = len(D)
  E_tau = 1 #we use the ML estimate of tau as its first expectation guess

  elbo_list = []
  a_N_list = []
  b_N_list = []
  mu_N_list = []
  lambda_N_list = []


  # CAVI iterations ...
    # save ELBO for each iteration, plot them afterwards to show convergence

  for i in range(n_iter):
    # updating the parameters of mu
    mu_N = (lambda_0 * mu_0 + N * D.mean()) / (lambda_0 + N)
    lambda_N = (lambda_0 + N) * E_tau

    # updating the parameters of tau

    ### computing useful expectations and data terms
    E_mu = mu_N
    E_mu2 = 1/lambda_N + mu_N**2
    data_sum = np.sum(D)
    data_square_sum = np.sum(D**2)

    ### updating the parameters
    a_N = a_0 + (N+1)/2
    b_N = b_0 + 0.5 * (data_square_sum + lambda_0*mu_0**2 - 2*(data_sum +␣
↪lambda_0*mu_0)*E_mu + (N+lambda_0)*E_mu2)
    E_tau = a_N / b_N
```

```
    # saving the parameters
    a_N_list.append(a_N)
    b_N_list.append(b_N)
    mu_N_list.append(mu_N)
    lambda_N_list.append(lambda_N)

    i += 1 #iteration counter
    elbo_list.append(compute_elbo(D, a_0, b_0, mu_0, lambda_0, a_N, b_N, mu_N,␣
↪lambda_N))

  return a_N_list, b_N_list, mu_N_list, lambda_N_list, elbo_list
```

#### 1.0.4 *Question 1.3.15:*

What is the exact posterior? First derive it in closed form, and then implement a function that computes it for the given parameters:

[44]:
```
def compute_exact_posterior(D, a_0, b_0, mu_0, lambda_0):
  N = len(D)

  # computing parameters of the posterior Normal-Gamma distribution
  a_star = a_0 + (N-1)/2
  b_star = b_0 + 0.5 * (np.sum(D**2) + lambda_0*mu_0**2 - (np.sum(D) +␣
↪lambda_0*mu_0)**2/(N+lambda_0))
  lambda_star = lambda_0 + N
  mu_star = (lambda_0 * mu_0 + np.sum(D)) / (lambda_0 + N)

  return a_star, b_star, mu_star, lambda_star
```

#### 1.0.5 *Question 1.3.16:*

Run the VI algorithm on the datasets. Compare the inferred variational distribution with the exact posterior and the ML estimate. Visualize the results and discuss your findings.

[45]:
```
# number of iterations
n_iter = 4

# Comparing the exact posterior with the CAVI approximation
a_star_1, b_star_1, mu_star_1, lambda_star_1 =␣
↪compute_exact_posterior(dataset_1, a_0, b_0, mu_0, lambda_0)
a_0_list_1, b_0_list_1, mu_0_list_1, lambda_0_list_1, elbo_list_1 =␣
↪CAVI(dataset_1, a_0, b_0, mu_0, lambda_0, n_iter)

a_star_2, b_star_2, mu_star_2, lambda_star_2 =␣
↪compute_exact_posterior(dataset_2, a_0, b_0, mu_0, lambda_0)
```

```
a_0_list_2, b_0_list_2, mu_0_list_2, lambda_0_list_2, elbo_list_2 =␣
↪CAVI(dataset_2, a_0, b_0, mu_0, lambda_0, n_iter)

a_star_3, b_star_3, mu_star_3, lambda_star_3 =␣
↪compute_exact_posterior(dataset_3, a_0, b_0, mu_0, lambda_0)
a_0_list_3, b_0_list_3, mu_0_list_3, lambda_0_list_3, elbo_list_3 =␣
↪CAVI(dataset_3, a_0, b_0, mu_0, lambda_0, n_iter)

# Printing the parameters of the exact posterior and the CAVI approximation
print('Exact posterior parameters for dataset 1: ', round(a_star_1, 2),␣
↪round(b_star_1, 2), round(mu_star_1, 2), round(lambda_star_1, 2))
print('CAVI parameters for dataset 1: ', round(a_0_list_1[-1], 2),␣
↪round(b_0_list_1[-1], 2), round(mu_0_list_1[-1], 2),␣
↪round(lambda_0_list_1[-1], 2), '\n\n')

print('Exact posterior parameters for dataset 2: ', round(a_star_2, 2),␣
↪round(b_star_2, 2), round(mu_star_2, 2), round(lambda_star_2, 2))
print('CAVI parameters for dataset 2: ', round(a_0_list_2[-1], 2),␣
↪round(b_0_list_2[-1], 2), round(mu_0_list_2[-1], 2),␣
↪round(lambda_0_list_2[-1], 2), '\n\n')

print('Exact posterior parameters for dataset 3: ', round(a_star_3, 2),␣
↪round(b_star_3, 2), round(mu_star_3, 2), round(lambda_star_3, 2))
print('CAVI parameters for dataset 3: ', round(a_0_list_3[-1], 2),␣
↪round(b_0_list_3[-1], 2), round(mu_0_list_3[-1], 2),␣
↪round(lambda_0_list_3[-1], 2), '\n\n')
```

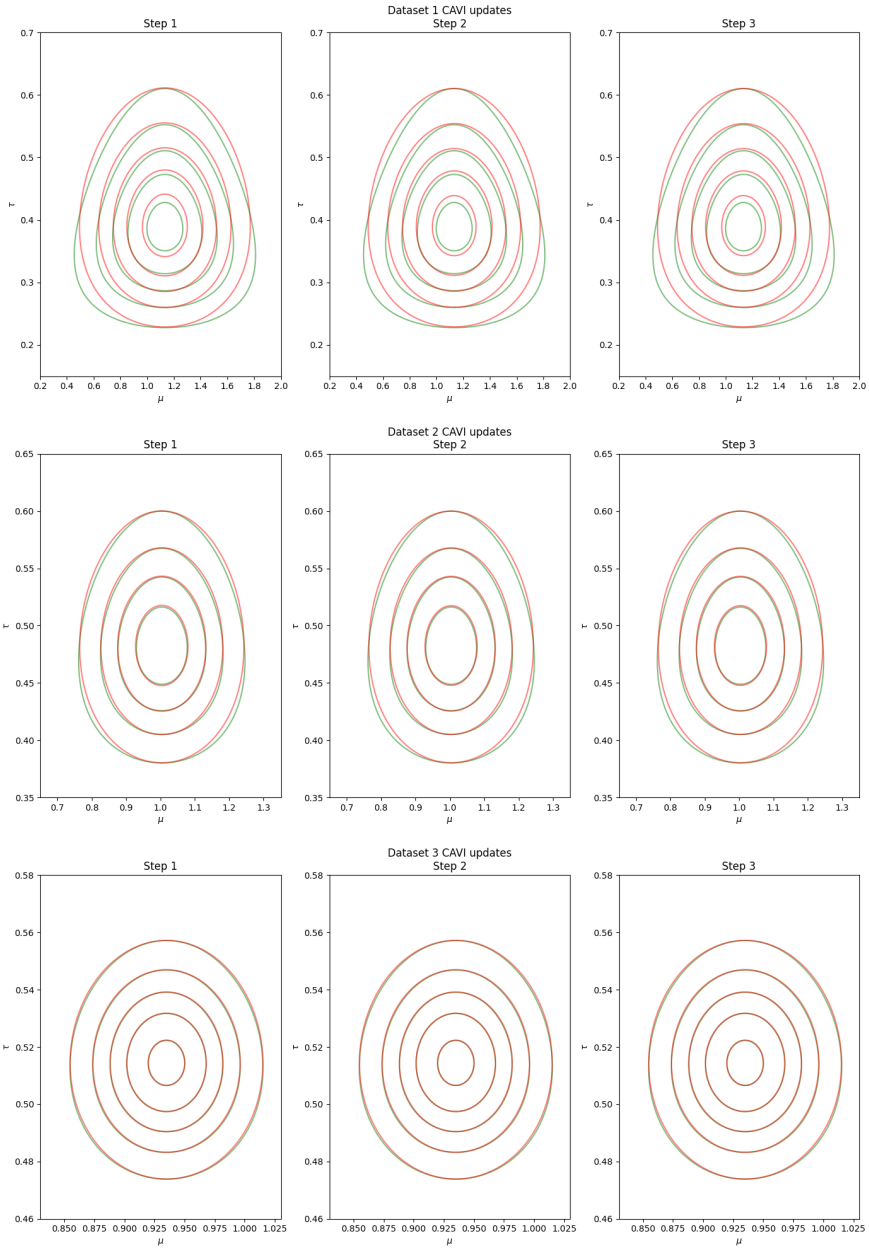
```
Exact posterior parameters for dataset 1:  14.5 36.08 1.13 20
CAVI parameters for dataset 1:  15.5 37.28 1.13 8.32


Exact posterior parameters for dataset 2:  59.5 122.47 1.0 110
CAVI parameters for dataset 2:  60.5 123.49 1.0 53.89


Exact posterior parameters for dataset 3:  509.5 989.49 0.94 1010
CAVI parameters for dataset 3:  510.5 990.46 0.94 520.57
```

[46]:
```
def plot_contour_plots(dataset, ax, X, Y) :
    a_N_list, b_N_list, mu_N_list, lambda_N_list, elbo_list = CAVI(dataset, a_0,␣
↪b_0, mu_0, lambda_0, n_iter)
    a_star, b_star, mu_star, lambda_star = compute_exact_posterior(dataset, a_0,␣
↪b_0, mu_0, lambda_0)

    # Contour plot of the true posterior
```

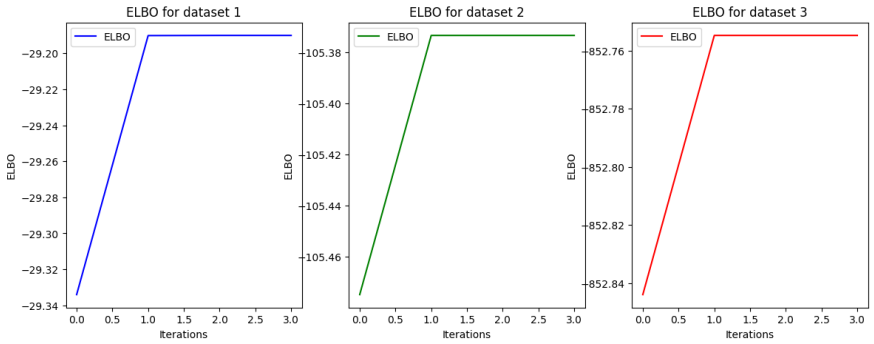
```
    true_posterior = norm.pdf(X, loc=mu_star, scale=np.sqrt(1 / (lambda_star *␣
↪Y))) \
        * gamma.pdf(Y, a_star, scale=1 / b_star)

    ax[0].contour(X, Y, true_posterior, 5, colors='green', alpha=0.5)
    ax[1].contour(X, Y, true_posterior, 5, colors='green', alpha=0.5)
    ax[2].contour(X, Y, true_posterior, 5, colors='green', alpha=0.5)


    for i in range(0, 3) :
        # Contour plot of the variational posterior
        variational_posterior = norm.pdf(X, loc=mu_N_list[-i-1], scale=np.sqrt(1␣
↪/ (lambda_N_list[-i-1]))) \
                                * gamma.pdf(Y, a_N_list[-i-1], scale=1 /␣
↪b_N_list[-i-1])

        ax[-i-1].contour(X, Y, variational_posterior, 5, colors='red', alpha=0.5)

        ax[-i-1].set_xlabel(r'$\mu$')
        ax[-i-1].set_ylabel(r'$\tau$')
```

[47]:
```
# Plotting the contour plots of the true posterior and the approximated posterior
# for the three first steps of CAVI,
# for each dataset?

fix, axs = plt.subplots(3, 3, figsize=(14, 20))

# Create a grid of x and y values for each dataset
mu_1 = np.linspace(0.2, 2, 1000)
tau_1 = np.linspace(0.15, 0.7, 1000)
MU_1, TAU_1 = np.meshgrid(mu_1, tau_1)

mu_2 = np.linspace(0.65, 1.35, 1000)
tau_2 = np.linspace(0.35, 0.65, 1000)
MU_2, TAU_2 = np.meshgrid(mu_2, tau_2)

mu_3 = np.linspace(0.83, 1.03, 1000)
tau_3 = np.linspace(0.46, 0.58, 1000)
MU_3, TAU_3 = np.meshgrid(mu_3, tau_3)

# Plotting the contour plots for each dataset
plot_contour_plots(dataset_1, axs[0], MU_1, TAU_1)
plot_contour_plots(dataset_2, axs[1], MU_2, TAU_2)
plot_contour_plots(dataset_3, axs[2], MU_3, TAU_3)

axs[0,0].set_title('\nStep 1')
axs[0,1].set_title('Dataset 1 CAVI updates\nStep 2')
```

```
axs[0,2].set_title('\nStep 3')

axs[1,0].set_title('\nStep 1')
axs[1,1].set_title('\nDataset 2 CAVI updates\nStep 2')
axs[1,2].set_title('\nStep 3')

axs[2,0].set_title('\nStep 1')
axs[2,1].set_title('\nDataset 3 CAVI updates\nStep 2')
axs[2,2].set_title('\nStep 3')

plt.tight_layout()
plt.show()
```


Dataset 1 CAVI updates
Step 1
Step 2
Step 3
Dataset 2 CAVI updates
Step 1
Step 2
Step 3
Dataset 3 CAVI updates
Step 1
Step 2
Step 3
μ
τ

[48]:
```
# Plotting the ELBO for each dataset
fig, axs = plt.subplots(1, 3, figsize=(14, 5))

axs[0].plot(elbo_list_1, color='blue', label='ELBO')
axs[0].set_title('ELBO for dataset 1')
axs[0].set_xlabel('Iterations')
axs[0].set_ylabel('ELBO')
axs[0].legend()

axs[1].plot(elbo_list_2, color='green', label='ELBO')
axs[1].set_title('ELBO for dataset 2')
axs[1].set_xlabel('Iterations')
axs[1].set_ylabel('ELBO')
axs[1].legend()

axs[2].plot(elbo_list_3, color='red', label='ELBO')
axs[2].set_title('ELBO for dataset 3')
axs[2].set_xlabel('Iterations')
axs[2].set_ylabel('ELBO')
axs[2].legend()

plt.show()
```



### 6.2 Code for Section 4: SVI-LDA

December 11, 2023

[1]:
```
import time

import numpy
import matplotlib.pyplot as plt
import numpy as np
import scipy.special as sp_spec
import scipy.stats as sp_stats
```

## 0.1 Assignment 1A. Problem 1.4.19 SVI.

### 0.1.1 Generate data

The cell below generates data for the LDA model. Note, for simplicity, we are using N_d = N for all d.

[2]:
```
def generate_data(D, N, K, W, eta, alpha):
    # sample K topics
    beta = sp_stats.dirichlet(eta).rvs(size=K)  # size K x W

    theta = np.zeros((D, K))   # size D x K

    w = np.zeros((D, N, W))
    z = np.zeros((D, N), dtype=int)
    for d in range(D):
        # sample document topic distribution
        theta_d = sp_stats.dirichlet(alpha).rvs(size=1)
        theta[d] = theta_d
        for n in range(N):
            # sample word to topic assignment
            z_nd = sp_stats.multinomial(n=1, p=theta[d, :]).rvs(size=1).
↪argmax(axis=1)[0]

            # sample word
            w_nd = sp_stats.multinomial(n=1, p=beta[z_nd, :]).rvs(1)

            z[d, n] = z_nd
            w[d, n] = w_nd
```

```
    return w, z, theta, beta

D_sim = 500
N_sim = 50
K_sim = 2
W_sim = 5

eta_sim = np.ones(W_sim)
eta_sim[3] = 0.0001  # Expect word 3 to not appear in data
eta_sim[1] = 3.  # Expect word 1 to be most common in data
alpha_sim = np.ones(K_sim) * 1.0
w0, z0, theta0, beta0 = generate_data(D_sim, N_sim, K_sim, W_sim, eta_sim,␣
↪alpha_sim)
w_cat = w0.argmax(axis=-1)  # remove one hot encoding
unique_z, counts_z = numpy.unique(z0[0, :], return_counts=True)
unique_w, counts_w = numpy.unique(w_cat[0, :], return_counts=True)

# Sanity checks for data generation
print(f"Average z of each document should be close to theta of document. \n␣
↪Theta of doc 0: {theta0[0]} \n Mean z of doc 0: {counts_z/N_sim}")
print(f"Beta of topic 0: {beta0[0]}")
print(f"Beta of topic 1: {beta0[1]}")
print(f"Word to topic assignment, z, of document 0: {z0[0, 0:10]}")
print(f"Observed words, w, of document 0: {w_cat[0, 0:10]}")
print(f"Unique words and count of document 0: {[f'{u}: {c}' for u, c in␣
↪zip(unique_w, counts_w)]}")
```

```
Average z of each document should be close to theta of document.
 Theta of doc 0: [0.83470541 0.16529459]
 Mean z of doc 0: [0.8 0.2]
Beta of topic 0: [0.03137604 0.74838297 0.01369107 0.         0.20654992]
Beta of topic 1: [0.22154783 0.53229331 0.11406411 0.         0.13209475]
Word to topic assignment, z, of document 0: [0 0 1 0 1 0 0 0 0 1]
Observed words, w, of document 0: [1 1 1 4 1 4 4 1 1 1]
Unique words and count of document 0: ['0: 2', '1: 37', '4: 11']
```

[3]:
```
import torch
import torch.distributions as t_dist

def generate_data_torch(D, N, K, W, eta, alpha):
    """
    Torch implementation for generating data using the LDA model. Needed for␣
↪sampling larger datasets.
    """
    # sample K topics
    beta_dist = t_dist.Dirichlet(torch.from_numpy(eta))
    beta = beta_dist.sample([K])  # size K x W
```

```
    # sample document topic distribution
    theta_dist = t_dist.Dirichlet(torch.from_numpy(alpha))
    theta = theta_dist.sample([D])

    # sample word to topic assignment
    z_dist = t_dist.OneHotCategorical(probs=theta)
    z = z_dist.sample([N]).reshape(D, N, K)

    # sample word from selected topics
    beta_select = torch.einsum("kw, dnk -> dnw", beta, z)
    w_dist = t_dist.OneHotCategorical(probs=beta_select)
    w = w_dist.sample([1])

    w = w.reshape(D, N, W)

    return w.numpy(), z.numpy(), theta.numpy(), beta.numpy()
```

### 0.1.2 Helper functions

[4]:
```
def log_multivariate_beta_function(a, axis=None):
    return np.sum(sp_spec.gammaln(a)) - sp_spec.gammaln(np.sum(a, axis=axis))
```

### 0.1.3 CAVI Implementation, ELBO and initialization

[5]:
```
def initialize_q(w, D, N, K, W):
    """
    Random initialization.
    """
    phi_init = np.random.random(size=(D, N, K))
    phi_init = phi_init / np.sum(phi_init, axis=-1, keepdims=True)
    gamma_init = np.random.randint(1, 10, size=(D, K))
    lmbda_init = np.random.randint(1, 10, size=(K, W))
    return phi_init, gamma_init, lmbda_init

def update_q_Z(w, gamma, lmbda):
    D, N, W = w.shape
    K, W = lmbda.shape
    E_log_theta = sp_spec.digamma(gamma) - sp_spec.digamma(np.sum(gamma, axis=1,␣
↪keepdims=True))  # size D x K
    E_log_beta = sp_spec.digamma(lmbda) - sp_spec.digamma(np.sum(lmbda, axis=1,␣
↪keepdims=True))   # size K x W
    log_rho = np.zeros((D, N, K))
    w_label = w.argmax(axis=-1)
    for d in range(D):
        for n in range(N):
            E_log_beta_wdn = E_log_beta[:, int(w_label[d, n])]
```

```
            E_log_theta_d = E_log_theta[d]
            log_rho_n = E_log_theta_d + E_log_beta_wdn
            log_rho[d, n, :] = log_rho_n

    phi = np.exp(log_rho - sp_spec.logsumexp(log_rho, axis=-1, keepdims=True))
    return phi

def update_q_theta(phi, alpha):
    E_Z = phi
    D, N, K = phi.shape
    gamma = np.zeros((D, K))
    for d in range(D):
        E_Z_d = E_Z[d]
        gamma[d] = alpha + np.sum(E_Z_d, axis=0)  # sum over N
    return gamma

def update_q_beta(w, phi, eta):
    E_Z = phi
    D, N, W = w.shape
    K = phi.shape[-1]
    lmbda = np.zeros((K, W))
    for k in range(K):
        lmbda[k, :] = eta
        for d in range(D):
            for n in range(N):
                lmbda[k, :] += E_Z[d,n,k] * w[d,n]  # Sum over d and n
    return lmbda

def calculate_elbo(w, phi, gamma, lmbda, eta, alpha):
    D, N, K = phi.shape
    W = eta.shape[0]
    E_log_theta = sp_spec.digamma(gamma) - sp_spec.digamma(np.sum(gamma, axis=1,␣
↪keepdims=True))  # size D x K
    E_log_beta = sp_spec.digamma(lmbda) - sp_spec.digamma(np.sum(lmbda, axis=1,␣
↪keepdims=True))  # size K x W
    E_Z = phi  # size D, N, K
    log_Beta_alpha = log_multivariate_beta_function(alpha)
    log_Beta_eta = log_multivariate_beta_function(eta)
    log_Beta_gamma = np.array([log_multivariate_beta_function(gamma[d, :]) for d␣
↪in range(D)])
    dg_gamma = sp_spec.digamma(gamma)
    log_Beta_lmbda = np.array([log_multivariate_beta_function(lmbda[k, :]) for k␣
↪in range(K)])
    dg_lmbda = sp_spec.digamma(lmbda)

    neg_CE_likelihood = np.einsum("dnk, kw, dnw", E_Z, E_log_beta, w)
    neg_CE_Z = np.einsum("dnk, dk -> ", E_Z, E_log_theta)
```

```
    neg_CE_theta = -D * log_Beta_alpha + np.einsum("k, dk ->", alpha - 1,␣
↪E_log_theta)
    neg_CE_beta = -K * log_Beta_eta + np.einsum("w, kw ->", eta - 1, E_log_beta)
    H_Z = -np.einsum("dnk, dnk ->", E_Z, np.log(E_Z))
    gamma_0 = np.sum(gamma, axis=1)
    dg_gamma0 = sp_spec.digamma(gamma_0)
    H_theta = np.sum(log_Beta_gamma + (gamma_0 - K) * dg_gamma0 - np.einsum("dk,␣
↪dk -> d", gamma - 1, dg_gamma))
    lmbda_0 = np.sum(lmbda, axis=1)
    dg_lmbda0 = sp_spec.digamma(lmbda_0)
    H_beta = np.sum(log_Beta_lmbda + (lmbda_0 - W) * dg_lmbda0 - np.einsum("kw,␣
↪kw -> k", lmbda - 1, dg_lmbda))
    return neg_CE_likelihood + neg_CE_Z + neg_CE_theta + neg_CE_beta + H_Z +␣
↪H_theta + H_beta

def CAVI_algorithm(w, K, n_iter, eta, alpha):
  D, N, W = w.shape
  phi, gamma, lmbda = initialize_q(w, D, N, K, W)

  # Store output per iteration
  elbo = np.zeros(n_iter)
  phi_out = np.zeros((n_iter, D, N, K))
  gamma_out = np.zeros((n_iter, D, K))
  lmbda_out = np.zeros((n_iter, K, W))

  for i in range(0, n_iter):

    ###### CAVI updates #######

    # q(Z) update
    phi = update_q_Z(w, gamma, lmbda)

    # q(theta) update
    gamma = update_q_theta(phi, alpha)

    # q(beta) update
    lmbda = update_q_beta(w, phi, eta)

    # ELBO
    elbo[i] = calculate_elbo(w, phi, gamma, lmbda, eta, alpha)

    # outputs
    phi_out[i] = phi
    gamma_out[i] = gamma
    lmbda_out[i] = lmbda

  return phi_out, gamma_out, lmbda_out, elbo
```

```
n_iter0 = 100
K0 = K_sim
W0 = W_sim
eta_prior0 = np.ones(W0)
alpha_prior0 = np.ones(K0)
phi_out0, gamma_out0, lmbda_out0, elbo0 = CAVI_algorithm(w0, K0, n_iter0,␣
↪eta_prior0, alpha_prior0)
final_phi0 = phi_out0[-1]
final_gamma0 = gamma_out0[-1]
final_lmbda0 = lmbda_out0[-1]
```

[6]:
```
precision = 3
print(f"----- Recall label switching - compare E[theta] and true theta and check␣
↪for label switching -----")
print(f"Final E[theta] of doc 0 CAVI:  {np.round(final_gamma0[0] / np.
↪sum(final_gamma0[0], axis=0, keepdims=True), precision)}")
print(f"True theta of doc 0:          {np.round(theta0[0], precision)}")

print(f"----- Recall label switching - e.g. E[beta_0] could be fit to true␣
↪theta_1. -----")
print(f"Final E[beta] k=0: {np.round(final_lmbda0[0, :] / np.sum(final_lmbda0[0,␣
↪:], axis=-1, keepdims=True), precision)}")
print(f"Final E[beta] k=1: {np.round(final_lmbda0[1, :] / np.sum(final_lmbda0[1,␣
↪:], axis=-1, keepdims=True), precision)}")
print(f"True beta k=0: {np.round(beta0[0, :], precision)}")
print(f"True beta k=1: {np.round(beta0[1, :], precision)}")
```

```
----- Recall label switching - compare E[theta] and true theta and check for
label switching -----
Final E[theta] of doc 0 CAVI:  [0.889 0.111]
True theta of doc 0:          [0.835 0.165]
----- Recall label switching - e.g. E[beta_0] could be fit to true theta_1.
-----
Final E[beta] k=0: [0.001 0.777 0.    0.    0.222]
Final E[beta] k=1: [0.308 0.448 0.155 0.    0.089]
True beta k=0: [0.031 0.748 0.014 0.    0.207]
True beta k=1: [0.222 0.532 0.114 0.    0.132]
```

### 0.1.4 SVI Implementation

Using the CAVI updates as a template, finish the code below.

[7]:
```
def update_q_Z_svi(batch, w, gamma, lmbda):
    """
    TODO: rewrite to SVI update
    """
```

```
    w = w[batch]
    gamma = gamma[batch]

    D, N, W = w.shape
    K, W = lmbda.shape
    E_log_theta = sp_spec.digamma(gamma) - sp_spec.digamma(np.sum(gamma, axis=1,␣
 ↪keepdims=True))  # size D x K
    E_log_beta = sp_spec.digamma(lmbda) - sp_spec.digamma(np.sum(lmbda, axis=1,␣
 ↪keepdims=True))   # size K x W
    log_rho = np.zeros((D, N, K))
    w_label = w.argmax(axis=-1)
    for d in range(D):
        for n in range(N):
            E_log_beta_wdn = E_log_beta[:, int(w_label[d, n])]
            E_log_theta_d = E_log_theta[d]
            log_rho_n = E_log_theta_d + E_log_beta_wdn
            log_rho[d, n, :] = log_rho_n

    phi = np.exp(log_rho - sp_spec.logsumexp(log_rho, axis=-1, keepdims=True))
    return phi

def update_q_theta_svi(batch, phi, alpha):
    """
    TODO: rewrite to SVI update
    """

    phi = phi[batch]

    E_Z = phi
    D, N, K = phi.shape
    gamma = np.zeros((D, K))
    for d in range(D):
        E_Z_d = E_Z[d]
        gamma[d] = alpha + np.sum(E_Z_d, axis=0)  # sum over N
    return gamma

def update_q_beta_svi(batch, w, phi, eta):
    """
    TODO: rewrite to SVI update
    """

    D = w.shape[0]

    w = w[batch]
    phi = phi[batch]

    E_Z = phi
```

```
    S, N, W = w.shape
    K = phi.shape[-1]
    lmbda = np.zeros((K, W))
    for k in range(K):
        lmbda[k, :] = eta
        for d in range(S):
            for n in range(N):
                lmbda[k, :] += (D/S) * E_Z[d,n,k] * w[d,n]  # Sum over d and n

    return lmbda


def SVI_algorithm(w, K, S, n_iter, eta, alpha):
  """
  Add SVI Specific code here.
  """
  D, N, W = w.shape
  phi, gamma, lmbda = initialize_q(w, D, N, K, W)

  # Store output per iteration
  elbo = np.zeros(n_iter)
  phi_out = np.zeros((n_iter, D, N, K))
  gamma_out = np.zeros((n_iter, D, K))
  lmbda_out = np.zeros((n_iter, K, W))

  for i in range(0, n_iter):
    # Sample batch and set step size, rho.
    batch = np.random.choice(D, S, replace=False)
    rho = 1 / (i + 1)  # step size that respects Robbins-Monro conditions

    ###### SVI updates #######

    # q(Z) update
    phi_batch = update_q_Z_svi(batch, w, gamma, lmbda)

    # q(theta) update
    gamma_batch = update_q_theta_svi(batch, phi, alpha)

    # update phi and gamma
    for d in range(batch.shape[0]) :
        phi[batch[d]] = phi_batch[d]
        gamma[batch[d]] = gamma_batch[d]


    # q(beta) update
    lmbda_batch = update_q_beta_svi(batch, w, phi, eta)
```

```
    lmbda = rho * lmbda_batch + (1 - rho) * lmbda_out[i-1]

    # ELBO
    elbo[i] = calculate_elbo(w, phi, gamma, lmbda, eta, alpha)

    # outputs
    phi_out[i] = phi
    gamma_out[i] = gamma
    lmbda_out[i] = lmbda

  return phi_out, gamma_out, lmbda_out, elbo
```

### 0.1.5 CASE 1

Tiny dataset

[8]:
```
np.random.seed(0)

# Data simulation parameters
D1 = 50
N1 = 50
K1 = 2
W1 = 5
eta_sim1 = np.ones(W1)
alpha_sim1 = np.ones(K1)

w1, z1, theta1, beta1 = generate_data(D1, N1, K1, W1, eta_sim1, alpha_sim1)


# Inference parameters
n_iter_cavi1 = 100
n_iter_svi1 = 100
eta_prior1 = np.ones(W1) * 1.
alpha_prior1 = np.ones(K1) * 1.
S1 = 5 # batch size

start_cavi1 = time.time()
phi_out1_cavi, gamma_out1_cavi, lmbda_out1_cavi, elbo1_cavi = CAVI_algorithm(w1,␣
 ↪K1, n_iter_cavi1, eta_prior1, alpha_prior1)
end_cavi1 = time.time()

start_svi1 = time.time()
phi_out1_svi, gamma_out1_svi, lmbda_out1_svi, elbo1_svi = SVI_algorithm(w1, K1,␣
 ↪S1, n_iter_svi1, eta_prior1, alpha_prior1)
end_svi1 = time.time()

final_phi1_cavi = phi_out1_cavi[-1]
```

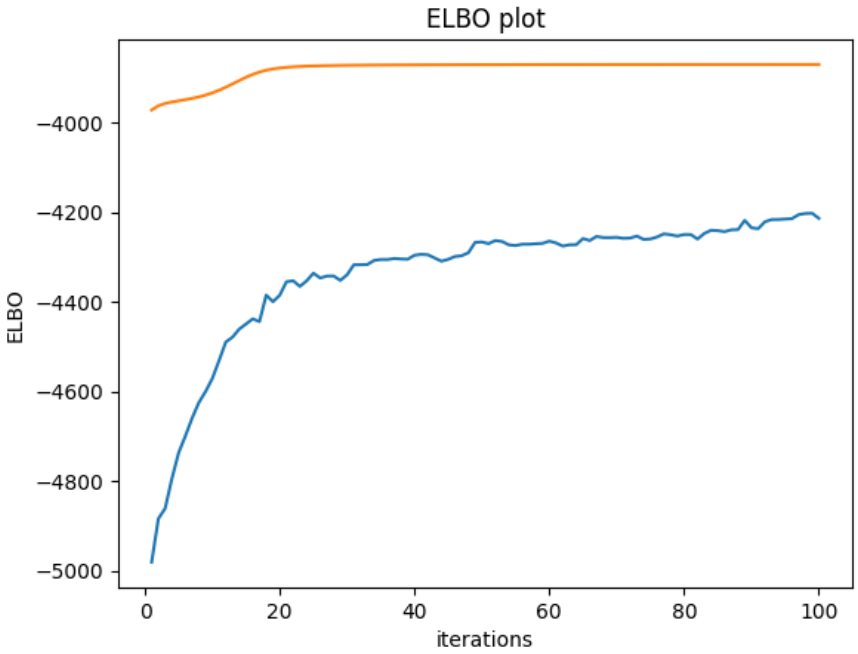
```
final_gamma1_cavi = gamma_out1_cavi[-1]
final_lmbda1_cavi = lmbda_out1_cavi[-1]
final_phi1_svi = phi_out1_svi[-1]
final_gamma1_svi = gamma_out1_svi[-1]
final_lmbda1_svi = lmbda_out1_svi[-1]
```

**Evaluation** Do not expect perfect results in terms expectations being identical to the "true" theta and beta. Do not expect the ELBO plot of your SVI alg to be the same as the CAVI alg. However, it should increase and be in the same ball park as that of the CAVI alg.

[9]:
```
np.set_printoptions(formatter={'float': lambda x: "{0:0.3f}".format(x)})
print(f"----- Recall label switching - compare E[theta] and true theta and check␣
 ↪for label switching -----")
print(f"E[theta] of doc 0 SVI:  {final_gamma1_svi[0] / np.
 ↪sum(final_gamma1_svi[0], axis=0, keepdims=True)}")
print(f"E[theta] of doc 0 CAVI: {final_gamma1_cavi[0] / np.
 ↪sum(final_gamma1_cavi[0], axis=0, keepdims=True)}")
print(f"True theta of doc 0:    {theta1[0]}")

print(f"----- Recall label switching - e.g. E[beta_0] could be fit to true␣
 ↪theta_1. -----")
print(f"E[beta] SVI k=0:    {final_lmbda1_svi[0, :] / np.sum(final_lmbda1_svi[0,␣
 ↪:], axis=-1, keepdims=True)}")
print(f"E[beta] SVI k=1:    {final_lmbda1_svi[1, :] / np.sum(final_lmbda1_svi[1,␣
 ↪:], axis=-1, keepdims=True)}")
print(f"E[beta] CAVI k=0:   {final_lmbda1_cavi[0, :] / np.
 ↪sum(final_lmbda1_cavi[0, :], axis=-1, keepdims=True)}")
print(f"E[beta] CAVI k=1:   {final_lmbda1_cavi[1, :] / np.
 ↪sum(final_lmbda1_cavi[1, :], axis=-1, keepdims=True)}")
print(f"True beta k=0:      {beta1[0, :]}")
print(f"True beta k=1:      {beta1[1, :]}")
```

```
----- Recall label switching - compare E[theta] and true theta and check for
label switching -----
E[theta] of doc 0 SVI:  [0.588 0.412]
E[theta] of doc 0 CAVI: [0.475 0.525]
True theta of doc 0:    [0.676 0.324]
----- Recall label switching - e.g. E[beta_0] could be fit to true theta_1.
-----
E[beta] SVI k=0:    [0.199 0.121 0.293 0.333 0.054]
E[beta] SVI k=1:    [0.126 0.222 0.197 0.300 0.156]
E[beta] CAVI k=0:   [0.276 0.347 0.129 0.095 0.154]
E[beta] CAVI k=1:   [0.075 0.011 0.351 0.503 0.059]
True beta k=0:      [0.185 0.291 0.214 0.183 0.128]
True beta k=1:      [0.136 0.075 0.291 0.434 0.063]
```

[10]:
```
plt.plot(list(range(1, n_iter_cavi1 + 1)), elbo1_svi[np.arange(0, n_iter_svi1,␣
↪int(n_iter_svi1 / n_iter_cavi1))])
plt.plot(list(range(1, n_iter_cavi1 + 1)), elbo1_cavi)
plt.title("ELBO plot")
plt.xlabel("iterations")
plt.ylabel("ELBO")
plt.show()
```




### 0.1.6 CASE 2

Small dataset

[11]:
```
np.random.seed(0)

# Data simulation parameters
D2 = 1000
N2 = 50
K2 = 3
W2 = 10
eta_sim2 = np.ones(W2)
```

```
alpha_sim2 = np.ones(K2)

w2, z2, theta2, beta2 = generate_data(D2, N2, K2, W2, eta_sim2, alpha_sim2)

# Inference parameters
n_iter_cavi2 = 100
n_iter_svi2 = 100
eta_prior2 = np.ones(W2) * 1.
alpha_prior2 = np.ones(K2) * 1.
S2 = 100 # batch size

start_cavi2 = time.time()
phi_out2_cavi, gamma_out2_cavi, lmbda_out2_cavi, elbo2_cavi = CAVI_algorithm(w2,␣
 ↪K2, n_iter_cavi2, eta_prior2, alpha_prior2)
end_cavi2 = time.time()

start_svi2 = time.time()
phi_out2_svi, gamma_out2_svi, lmbda_out2_svi, elbo2_svi = SVI_algorithm(w2, K2,␣
 ↪S2, n_iter_svi2, eta_prior2, alpha_prior2)
end_svi2 = time.time()

final_phi2_cavi = phi_out2_cavi[-1]
final_gamma2_cavi = gamma_out2_cavi[-1]
final_lmbda2_cavi = lmbda_out2_cavi[-1]
final_phi2_svi = phi_out2_svi[-1]
final_gamma2_svi = gamma_out2_svi[-1]
final_lmbda2_svi = lmbda_out2_svi[-1]
```

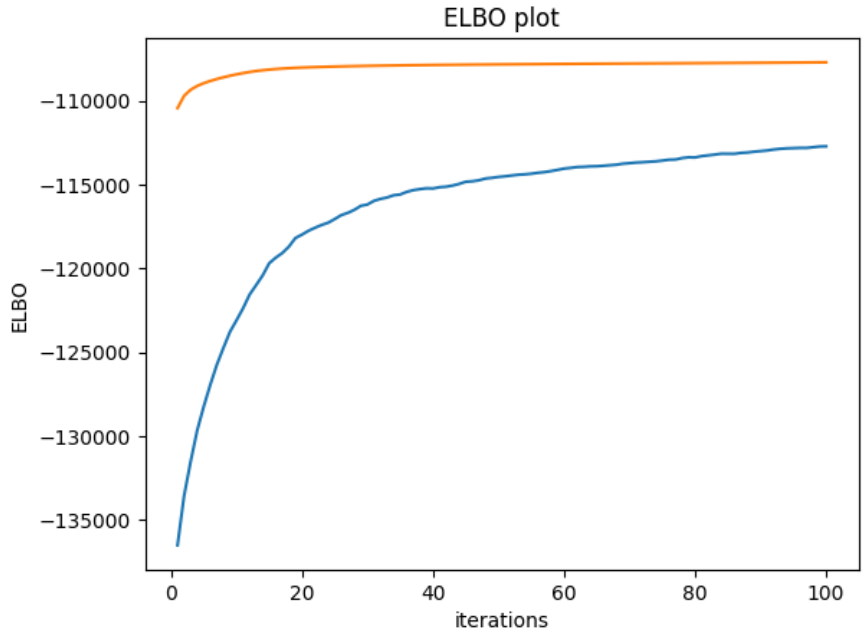
**Evaluation** Do not expect perfect results in terms expectations being identical to the "true" theta and beta. Do not expect the ELBO plot of your SVI alg to be the same as the CAVI alg. However, it should increase and be in the same ball park as that of the CAVI alg.

[12]:
```
np.set_printoptions(formatter={'float': lambda x: "{0:0.3f}".format(x)})
print(f"----- Recall label switching - compare E[theta] and true theta and check␣
 ↪for label switching -----")
print(f"E[theta] of doc 0 SVI:      {final_gamma2_svi[0] / np.
 ↪sum(final_gamma2_svi[0], axis=0, keepdims=True)}")
print(f"E[theta] of doc 0 CAVI:     {final_gamma2_cavi[0] / np.
 ↪sum(final_gamma2_cavi[0], axis=0, keepdims=True)}")
print(f"True theta of doc 0:        {theta2[0]}")

print(f"----- Recall label switching - e.g. E[beta_0] could be fit to true␣
 ↪theta_1. -----")
print(f"E[beta] k=0:    {final_lmbda2_svi[0, :] / np.sum(final_lmbda2_svi[0, :],␣
 ↪axis=-1, keepdims=True)}")
```

```
print(f"E[beta] k=1:    {final_lmbda2_svi[1, :] / np.sum(final_lmbda2_svi[1, :],␣
 ↪axis=-1, keepdims=True)}")
print(f"E[beta] k=2:    {final_lmbda2_svi[2, :] / np.sum(final_lmbda2_svi[2, :],␣
 ↪axis=-1, keepdims=True)}")
print(f"True beta k=0:  {beta2[0, :]}")
print(f"True beta k=1:  {beta2[1, :]}")
print(f"True beta k=2:  {beta2[2, :]}")

print(f"Time SVI: {end_svi2 - start_svi2}")
print(f"Time CAVI: {end_cavi2 - start_cavi2}")
```

```
----- Recall label switching - compare E[theta] and true theta and check for
label switching -----
E[theta] of doc 0 SVI:      [0.451 0.176 0.373]
E[theta] of doc 0 CAVI:     [0.238 0.338 0.424]
True theta of doc 0:        [0.128 0.619 0.253]
----- Recall label switching - e.g. E[beta_0] could be fit to true theta_1.
-----
E[beta] k=0:    [0.103 0.094 0.061 0.247 0.034 0.015 0.027 0.048 0.280 0.091]
E[beta] k=1:    [0.155 0.144 0.056 0.125 0.021 0.065 0.006 0.280 0.082 0.066]
E[beta] k=2:    [0.245 0.057 0.085 0.045 0.008 0.093 0.030 0.220 0.110 0.106]
True beta k=0:  [0.067 0.105 0.077 0.066 0.046 0.087 0.048 0.186 0.277 0.040]
True beta k=1:  [0.139 0.067 0.074 0.230 0.007 0.008 0.002 0.158 0.134 0.181]
True beta k=2:  [0.295 0.123 0.047 0.116 0.010 0.078 0.012 0.222 0.057 0.041]
Time SVI: 9.657719135284424
Time CAVI: 66.80321002006531
```

[13]:
```
plt.plot(list(range(1, n_iter_cavi2 + 1)), elbo2_svi[np.arange(0, n_iter_svi2,␣
 ↪int(n_iter_svi2 / n_iter_cavi2))])
plt.plot(list(range(1, n_iter_cavi2 + 1)), elbo2_cavi)
plt.title("ELBO plot")
plt.xlabel("iterations")
plt.ylabel("ELBO")
plt.show()
```




[14]:
```
# Add your own code for evaluation here (will not be graded)
```

### 0.1.7 CASE 3

Medium small dataset, one iteration for time analysis.

[15]:
```
np.random.seed(0)

# Data simulation parameters
D3 = 10**4
N3 = 500
K3 = 5
W3 = 10
eta_sim3 = np.ones(W3)
alpha_sim3 = np.ones(K3)

w3, z3, theta3, beta3 = generate_data_torch(D3, N3, K3, W3, eta_sim3, alpha_sim3)

# Inference parameters
n_iter3 = 1
eta_prior3 = np.ones(W3) * 1.
```

```
alpha_prior3 = np.ones(K3) * 1.
S3 = 100 # batch size

start_cavi3 = time.time()
phi_out3_cavi, gamma_out3_cavi, lmbda_out3_cavi, elbo3_cavi = CAVI_algorithm(w3,␣
↪K3, n_iter3, eta_prior3, alpha_prior3)
end_cavi3 = time.time()

start_svi3 = time.time()
phi_out3_svi, gamma_out3_svi, lmbda_out3_svi, elbo3_svi = SVI_algorithm(w3, K3,␣
↪S3, n_iter3, eta_prior3, alpha_prior3)
end_svi3 = time.time()

final_phi3_cavi = phi_out3_cavi[-1]
final_gamma3_cavi = gamma_out3_cavi[-1]
final_lmbda3_cavi = lmbda_out3_cavi[-1]
final_phi3_svi = phi_out3_svi[-1]
final_gamma3_svi = gamma_out3_svi[-1]
final_lmbda3_svi = lmbda_out3_svi[-1]
```

[16]:
```
print(f"Examine per iteration run time.")
print(f"Time SVI: {end_svi3 - start_svi3}")
print(f"Time CAVI: {end_cavi3 - start_cavi3}")
```

```
Examine per iteration run time.
Time SVI: 2.311847686767578
Time CAVI: 97.87206411361694
```

[17]:
```
# Add your own code for evaluation here (will not be graded)
```