# Variational Inference for Hierarchical Poisson Factorization

## Computational Statistics Project 1

Leonardo Ruggieri

## 1 Introduction

Hierarchical Poisson Factorization (HPF) [2] is a probabilistic model for a recommender system. It is based on the Poisson matrix factorization, that gives the name to the model. This factorization involves the observations on the data matrix $y$ of users and items, where each data point is assumed to be a draw from an independent Poisson distribution.

## 2 The model

The HPF is based on the data matrix $y$, of dimensions $N$ (number of users) and $M$ (number of items). Each user is characterized by a vector of $K$ latent components, which represent the latent user preferences. Analogously, each item is characterized by a vector of $K$ latent components, which represent the latent item attributes. Thus, each data point is the inner product of the latent component for the user $u$ and the item $i$.

On top of that, Gamma hyperpriors[1] are specified for the rate parameters $\theta$ and $\beta$. $\xi_u$ is the rate parameter of $\theta_u$ and captures the prior on the user activity. $\eta_i$ is the rate parameter of $\beta_i$ and represents the item popularity.

The generative process is then the following:

1. For each user $u$:
   (a) Sample an activity value $\xi_u \sim Gamma(a', \frac{a'}{b'})$
   (b) For each component $k = 1, ..., K$, sample a preference $\theta_{u,k} \sim Gamma(a, \xi_u)$
2. For each item $i$:
   (a) Sample a popularity value $\eta_i \sim Gamma(c', \frac{c'}{d'})$
   (b) For each component $k = 1, ..., K$, sample an attribute $\beta_{i,k} \sim Gamma(c, \eta_i)$
3. For each $(u, i)$, sample an observation (rating) $y_{u,i} \sim Poisson(\theta_u^T \beta_i)$

Thanks to the Poisson matrix factorization, this model scales very efficiently in the context of sparse data. Indeed, since Poisson distribution puts positive mass on zero, the likelihood of the model will only depends on observation points that are different from zero:

$$p(y) = \prod_{u,i} \frac{(\theta_u^T \beta_i)^{y_{u,i}} \cdot e^{-\theta_u^T \beta_i}}{y_{u,i}!} = \prod_{u,i:\, y_{u,i}>0} \frac{(\theta_u^T \beta_i)^{y_{u,i}}}{y_{u,i}!} \cdot \prod_{u,i} e^{-\theta_u^T \beta_i}$$

As a consequence, inference will be conducted only on a fraction of the data points, speeding up the runtime of the algorithm.

[1] All the Gamma distributions are parameterized with a shape and a rate parameter.

## 3 Mean-field Variational Inference

We derive the variational algorithm, which is a method for approximating probability densities through optimization. It aims at approximating the untractable posterior distribution of the model. We aim at minimizing the Kullback-Leibler divergence between the *true* posterior and a variational distribution, which amounts to finding the tightest lower bound for the log-evidence $p(y)$, called evidence lower bound (ELBO), by means of a coordinate ascent algorithm.

We introduce additional latent variables, $k$ of them for each pair of user and item, such that $z_{u,i;k} \sim Poisson(\theta_{uk}\beta_{ik})$, a technical device that simplifies the process of finding the solution of the optimization problem. In this way, $y_{u,i} = \sum_k z_{u,i;k}$. Hence, the posterior of interest is $p(\beta, \theta, \eta, \xi, z|y)$ and the variational will be $q(\beta, \theta, \eta, \xi, z)$. We impose the *mean-field* assumption, that is, the variational distribution can be factorized, so that each variable is independent and governed by its own distribution.[1]

$$q(\beta, \theta, \xi, \eta, z) = \prod_{i,k} q(\beta_{ik}|\lambda_{ik}) \prod_{u,k} q(\theta_{uk}|\gamma_{uk}) \prod_{u} q(\xi_u|k_u) \prod_{u,k} q(\eta_i|\tau_i) \prod_{u,k} q(z_{ui}|\phi_{ui})$$

The first step is to derive the complete conditionals of $\theta_{uk}$, $\beta_{ik}$, $\xi_u$, $\eta_i$ and $z_{ui}$. By Gamma-Poisson conjugacy, the first four complete conditionals are:

1. From $\theta_{uk}$ and $z_{uk}|\theta_{uk}$ we get: $\theta_{uk} \sim \text{Gamma}(a + \sum_i z_{uik}, \xi_u + \sum_i \beta_{ik})$.
2. From $\beta_{ik}$ and $z_{ik}|\beta_{ik}$ we get: $\beta_{ik} \sim \text{Gamma}(c + \sum_i z_{uik}, \eta_i + \sum_u \theta_{uk})$
3. From $\xi_u$ and $\theta_u|\xi_u$ we obtain: $\xi_u|\theta_u \sim \text{Gamma}(a' + Ka, a'/b' + \sum_k \theta_{uk})$
4. From $\eta_i$ and $\beta_|\eta_i$ we obtain: $\eta_i|\beta_i \sim \text{Gamma}(c' + Kc, c'/d' + \sum_k \beta_{ik})$

Since $z_{ui}$ is a K vector of Poisson that sum to $y_{ui}$, the complete conditional for the vector $z_{ui}$ is a Multinomial with normalized probabilities:

$$z_{ui}|\beta, \theta, y \sim \text{Mult}\left(y_{ui}, \frac{\theta_u \beta_i}{\sum_k \theta_u k \beta_i k}\right)$$

It can be shown[3] that if all the complete conditionals of the model are in the exponential family, then the update for each variational component is optimal if it is in the same family as the complete conditional: hence, we impose this condition. In this case, the update rule only amounts to update each variational parameter with the expectation of the corresponding natural conditional parameter, given all the other parameters and the observations. For instance, the variational parameter of $\theta_{uk}$ will be updated as follows[2]:

$$\gamma_{uk}^{shape} = E^q\left[a + \sum_i z_{uik}\right] = a + \sum_i (y_{ui}\phi_{uik})$$

$$\gamma_{uk}^{rate} = E^q\left[\xi_u + \sum_i \beta_{ik}\right] = k_u^{shape}/k_u^{rate} + \sum_i (\lambda_i^{shape}/\lambda_i^{rate})$$

[2] For the sake of brevity, updates for $\lambda$, $k$, and $\tau$ are omitted, but are analogous.

The update for $\phi$ is $\phi_{u,i} \propto \exp\{E_q[log\ \theta_u + log\ \beta_i]\}$. Hence, we can write the update rule for $\phi_{u,i}$ as[3]:

$$\phi_{uik} \propto \Psi(\gamma_{uk}^{shape}) - \log(\gamma_{uk}^{rate}) + \Psi(\lambda_{ik}^{shape}) - \log(\lambda_{ik}^{rate}) \tag{1}$$

We now have all the update rules we need to implement the algorithm.

### 3.1 The algorithm

**Algorithm 1:** Variational inference for Poisson Factorization

**Initialize** $\gamma_u, \kappa_u^r, \lambda_i, \tau_i^r$ to the prior with a small random offset.
Set $\kappa_u^s = Ka + a'$ and $\tau_i^s = Kc + c'$.
**while** *log-likelihood not converged* **do**
  **for** $u$ *and* $i$ *such that* $y_{ui} > 0$ **do**
    **for** $k = 1, ..., K$ **do**
      $\phi_{uik} \propto \Psi(\gamma_{uk}^{shp}) - log(\gamma_{uk}^{rte}) + \Psi(\lambda_{ik}^{shp}) - log(\lambda_{ik}^{rte})$
    **end**
  **end**
  **for** $u = 1, ..., U$ **do**
    $\gamma_{uk}^{shp} = a + \sum_i y_{ui}\phi_{uik}$
    $\gamma_{uk}^{rte} = \frac{\kappa_u^{shp}}{\kappa_u^{rte}} + \sum_i \frac{\lambda_{ik}^{shp}}{\lambda_{ik}^{r}}$
    $\kappa_u^{rte} = a'/b' + \sum_k \frac{\gamma_{uk}^{rte}}{\gamma_{uk}^{shp}}$
  **end**
  **for** $i = 1, ..., I$ **do**
    $\lambda_{i,k}^{shp} = c + \sum_i y_{ui}\phi_{uik}$
    $\lambda_{i,k}^{rte} = \frac{\tau_i^{shp}}{\tau_i^{rte}} + \sum_u \frac{\gamma_{u,k}^{rte}}{\gamma_{u,k}^{shp}}$
    $\tau_i^{rte} = c'/d' + \sum_k \frac{\lambda_{ik}^{shp}}{\lambda_{ik}^{rte}}$
  **end**
**end**

### 3.2 Possible improvements

Although the algorithm implemented scales very well with massive dataset thanks to the sparse matrix estimation, there are other solutions other than the mean-field variational algorithm that may speed up the training process even more. For example, one could implement a stochastic variational inference algorithm [3]. The main advantages would be the use of subsets of data at each iterations and more efficient updates.

[3] Recall that the expectation of the log-$\Gamma(\alpha, \beta)$ random variable is $\Psi(\alpha) - \log\beta$, where $\Psi$ is the digamma function.

# 4 Implementation in Python

The algorithm has been implemented using Python3 and it is available in the hpf_vi.py file. The code is also available in the Appendix of this document. A notebook is also provided with the algorithm applied on simulated data.

## 4.1 Initialization of the variational parameters

The model implemented initializes the user parameters $\gamma_u$ and the item parameters $\lambda_i$ to the prior, plus a small random offset generated with a Uniform$(0, 1)$.

In this way, $\gamma_{uk}^{shp} = a$ and $\gamma_{uk}^{rte} = a/b'$, so that the expectation of the variational $\theta$ is $b'$. Similarly, $\lambda_{ik}^{shp} = c$ and $\lambda_{ik}^{rte} = c/d'$, so that the expectation of the variational $\beta$ is $d'$. For the same reason, we set the variational parameters of $\xi_u$ and $\eta_i$, which are $k_u^{rte}$ and $\tau_i^{rte}$, to the prior, plus the same small random offset.

In addition, as suggested in the original paper, low shape parameters for the Gamma priors on user preferences $\theta_u$ and item attributes $\beta_i$ are set, favouring a sparse representation that better fits the data in most of the cases.

# References

1. 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)
2. Gopalan, P., Hofman, J.M., Blei, D.M.: Scalable recommendation with hierarchical poisson factorization. In: UAI. pp. 326–335 (2015)
3. Hoffman, M.D., Blei, D.M., Wang, C., Paisley, J.: Stochastic variational inference. The Journal of Machine Learning Research **14**(1), 1303–1347 (2013)

# A Appendix – Python Implementation

```
import numpy as np
from scipy.special import digamma
import time
from sklearn.metrics import mean_squared_error

class hpf_vi():
    def __init__(self, a = 0.3, c = 0.3, a1 = 0.3, b1 = 1, c1 = 0.3, d1 = 1, K = 10):
        '''
        Initialization of the parameter matrices used in the CAVI algorithm.
        The user can modify the hyperparameters and the dimension of latent attributes and preferences K.

        Parameters:
        ----------
            - a : float
              shape parameter for the Gamma(a, activity_u) prior
              for user preferences.
            - c : float
              shape parameter for the Gamma(c, popularity_i) prior
              for item attributes.
            - a1, b1 : floats
              parameters of the Gamma(a1, a1/b1) prior for user activity.
            - c_1, d_1 : floats
              parameters of the Gamma(c1, c1/d1) prior for item popularity.
            - K : int
              dimensionality of latent attributes and preferences.
        '''
        self.a, self.c, self.a1, self.b1, self.c1, self.d1, self.K = a, c, a1, b1, c1, d1, K

    def fit(self, train, iterations, tol=0.5, valid=None):
        '''
        Fit the Hierarchical Poisson Factorization model via Coordinate Ascent Variational Algorithm (CAVI)
        and generates the corresponding observation matrix based on the variational parameters.
```

```
        The algorithm stops either when the log-likelihood difference is less than the tolerance or after the number of iterations specified.

        Parameters:
        ----------
            - train : numpy.array
              UxI array with U = users and I = items.
            - iterations: int
              number of desired training epochs.
            - tol: float
              tolerance stopping criterion.
        '''
        self.train = train
        self.valid = valid

        # Dataset dimensions
        self.U, self.I = train.shape

        self.initialize()
        self.mse_train = np.zeros(iterations)
        self.ll = np.zeros(iterations)

        self.its = 0

        # Building user preferences and item attributes
        self.theta = self.gamma_shp/self.gamma_rte
        self.beta = self.lambda_shp/self.lambda_rte
        old_ll = self.log_likelihood(self.train, self.beta, self.theta)
        stop = False

        self.mse_valid = np.zeros(iterations)

        def MSE(pred, values):
            prediction, values = pred[values.nonzero()].flatten(), values[values.nonzero()].flatten()
            return mean_squared_error(prediction, values)

        import time
        from sklearn.metrics import mean_squared_error
        tic = time.clock()

        while not stop and self.its < iterations:
            for u, i in zip(train.nonzero()[0], train.nonzero()[1]):
```

```
74                  self.phi[u,i] =
                    ↪  [np.exp(digamma(self.gamma_shp[u,k]) -
                    ↪  np.log(self.gamma_rte[u,k])\
75                  + digamma(self.lambda_shp[i,k]) -
                    ↪  np.log(self.lambda_rte[i,k])) for k in
                    ↪  range(self.K)]
76                  self.phi[u,i] = self.phi[u,i] /
                    ↪  np.sum(self.phi[u,i])
77
78              for u in range(self.U):
79                  self.gamma_shp[u] = [self.a +
                    ↪  np.sum(train[u]*self.phi[u,:,k]) for k in
                    ↪  range(self.K)]
80                  self.gamma_rte[u] = [self.k_shp/self.k_rte[u] +
                    ↪  np.sum(self.lambda_shp[:,k]/self.lambda_rte[:,k])
                    ↪  for k in range(self.K)]
81                  self.k_rte[u] = self.a1/self.b1 +
                    ↪  np.sum(self.gamma_shp[u]/self.gamma_rte[u])
82
83              for i in range(self.I):
84                  self.lambda_shp[i] = [self.c +
                    ↪  np.sum(train[:,i]*self.phi[:,i,k]) for k in
                    ↪  range(self.K)]
85                  self.lambda_rte[i] = [self.tau_shp/self.tau_rte[i]
                    ↪  +
                    ↪  np.sum(self.gamma_shp[:,k]/self.gamma_rte[:,k])
                    ↪  for k in range(self.K)]
86                  self.tau_rte[i] = self.c1/self.d1 +
                    ↪  np.sum(self.lambda_shp[i]/self.lambda_rte[i])
87
88              # Building user preferences and item attributes
89              self.theta = self.gamma_shp/self.gamma_rte
90              self.beta = self.lambda_shp/self.lambda_rte
91
92              # Evaluating the Loglikelihood
93              self.ll[self.its] = self.log_likelihood(self.train,
                ↪  self.beta, self.theta)
94
95              # Generating observations y
96              self.predicted = np.dot(self.theta, self.beta.T)
97
98              # In-sample MSE
99              self.mse_train[self.its] = MSE(self.predicted,
                ↪  self.train)
100
```

```
            # Out-of-sample MSE
            if valid is not None:
                self.mse_valid[self.its] = MSE(self.predicted,
                ↪  self.valid)

            if abs(self.ll[self.its] - old_ll) > tol:
                old_ll = self.ll[self.its]
                self.its += 1
                print(f"Iteration {self.its} completed.
                ↪  Log-likelihood: {self.ll[self.its-1]}")
            else:
                stop = True



        else:
            toc1 = time.clock()
            time1 = np.round(toc1 - tic,3)
            if self.its < iterations:
                print(f"Converged in {time1} seconds after
                ↪  {self.its} iterations. Log-likelihood:
                ↪  {self.ll[self.its-1]}.")
            else:
                print(f"Stopped after {time1} seconds, {self.its}
                ↪  iterations. Log-likelihood:
                ↪  {self.ll[self.its-1]}.")



    def recommend(self, test, t = 0.3):
        '''
        Using the fitted algorithm to make recommendations to
↪   users of our dataset.

        Parameters:
        ----------
            - t : float
              Delta-threshold for activating recommendations
        '''
        for u in range(self.U):
            recomm = []
            for i in range(self.I):
                if self.predicted[u,i] > t:
                    recomm.append(i)
            if [i > 0 for i in recomm]:
```

```
                print(f"User {u} may also like these items:
                ↪  {recomm}")


    def initialize(self):
        self.gamma_shp = np.random.uniform(0,1, size = (self.U,
        ↪  self.K)) + self.a
        self.gamma_rte = np.repeat(self.a/self.b1,self.K) +
        ↪  np.random.uniform(0,1, size = (self.U, self.K))
        self.k_rte = self.a1/self.b1 + np.random.uniform(0,1,
        ↪  self.U)
        self.k_shp = self.a1 + self.K*self.a

        self.lambda_shp = np.random.uniform(0,1, size = (self.I,
        ↪  self.K)) + self.c
        self.lambda_rte = np.repeat(self.c/self.d1,self.K) +
        ↪  np.random.uniform(0,1, size = (self.I, self.K))
        self.tau_rte = self.c1/self.d1 + np.random.uniform(0,1,
        ↪  self.I)
        self.tau_shp = self.c1 + self.K*self.c
        # Note that the parameters tau_shp and k_shp are not
        ↪  updated in the algorithm, so they are declared here.

        self.phi = np.zeros(shape=[self.U, self.I, self.K])

    def log_likelihood(self, train, beta, theta):
        '''
        Evaluating the log-likelihood
        '''
        self.train = train
        self.beta = beta
        self.theta = theta

        self.sumlog = 0
        self.prod = 1
        count_array = 0
        for u, i in zip(self.train.nonzero()[0],
        ↪  self.train.nonzero()[1]):
            self.dot = float(np.dot(theta[u],beta[i].T))
            self.dot_y = float(self.dot**train[u,i])
            self.dot_y_fact =
            ↪  float(self.dot/np.math.factorial(train[u,i]))
            self.logdot_y_fact = np.log(self.dot_y_fact)
            self.sumlog += self.logdot_y_fact
            count_array += 1
```

```
174
175
176         for u,i in zip(range(train.shape[0]),
            ↪  range(train.shape[1])):
177             self.exp = float(np.exp(-np.dot(theta[u],beta[i].T)))
178             self.prod = float(self.prod * self.exp)
179         self.logprod = np.log(self.prod)
180
181         return self.sumlog+self.logprod
```