EECS 6892 Bayesian models for machine learning Building a radio recommendation engine with Mixed Membership Matrix Factorization

John Min

May 5, 2014

1 Introduction

In building a radio recommendation engine, there are two approaches to generating content recommendations. The first is content-based; the other is to analyze the dyadic data in a collaborative filtering (matrix completion) setting. The dyads, i.e. the ordered pairs of object, in this context are the users and songs. In the content-based context, the high-level idea is to map each song into the same feature space, for example, by computing Mel-frequency cepstral coefficients and representing each song as a k dimensional vector. In the collaborative filtering environment, we observe N users, M items (songs), and data set $\mathcal{D} = \{(u_i, v_j, r_i j)\}$ for $i = 1, \ldots, N$ and $j = 1, \ldots, M$.

Two classes of dyadic data analysis algorithms are latent factor models such as (Bayesian) Probabilistic Matrix Factorization and mixed membership models such as Latent Dirichlet Allocation (LDA).

Matrix factorization represents each user u and each item v as vectors of latent features, $u_i, v_j \in \mathcal{R}^d$, respectively, where d is relatively small. A *static rating* is generated for a user-item pair by adding Gaussian noise to the inner product of the latent factor vectors, $r_{ij} = u_i \cdot v_j$.[1]

2 Mixed Membership Matrix Factorization

The Mixed Membership Matrix Factorization (M^3F) framework integrates discrete mixed membership modeling and continuous latent factor modeling for probabilistic dyadic data prediction. The following is the generative process

- K^U: the number of user topics
- K^M: the number of item topics
- $\Lambda^U \sim \text{Wishart}(W_0, \nu_0), \Lambda^M \sim \text{Wishart}(W_0, \nu_0)$
- $\mu^U \sim N(\mu_0, (\lambda_0 \Lambda^U)^{-1}), \mu^M \sim N(\mu_0, (\lambda_0 \Lambda^U)^{-1})$
- For each user $i \in \{1, \ldots, N\}$:
  - $u_i \sim N(\mu^U, (\lambda^U)^{-1})$
  - $\theta_i^U \sim \text{Dir}(\alpha / K^U)$
- For each item $j \in \{1, \ldots, M\}$:
  - $v_i \sim N(\mu^V, (\lambda^V)^{-1})$
  - $\theta_i^V \sim \text{Dir}(\alpha / K^V)$
- For each rating $r_i j$:
  - $z_{ij}^U \sim \text{Multi}(1, \theta_i^U), z_{ij}^V \sim \text{Multi}(1, \theta_j^V)$
  - $r_{ij} \sim N(\beta_{ij}^{kl} + u_i \cdot v_j, \sigma^2)$

2.1 Gibbs sampler

- $\Lambda^U \sim \text{Wishart}\left(\left(\mathbf{W_0}^{-1} + \sum_{i=1}^N (\mathbf{u_i} - \bar{\mathbf{u}})(\mathbf{u_i} - \bar{\mathbf{u}})^T + \frac{\lambda_0}{\lambda_0 + N} (\mu_0 - \bar{u})(\mu_0 - \bar{u})^\top \right)^{-1}, \nu_0 + N \right)$
- $\Lambda^M \sim \text{Wishart}\left(\left(\mathbf{W_0}^{-1} + \sum_{j=1}^M (\mathbf{v_j} - \bar{\mathbf{v}})(\mathbf{v_j} - \bar{\mathbf{v}})^T + \frac{\lambda_0}{\lambda_0 + M} (\mu_0 - \bar{v})(\mu_0 - \bar{v})^\top \right)^{-1}, \nu_0 + M \right)$

[1] This rating is called a static rating because the latent factor rating mechanism does not model the context in which a rating is given and does not model to exhibit different moods in dyadic interactions. This can be particularly useful in the case of Netflix, which allows multiple users to share a single account.

- $\mu^U \sim N\Big(\frac{\lambda_0\mu_0+\sum_{i=1}^N u_i}{\lambda_0+N}, ((\lambda_0+N)\Lambda^U)^{-1}\Big)$
- $\mu^V \sim N\Big(\frac{\lambda_0\mu_0+\sum_{j=1}^M v_j}{\lambda_0+M}, ((\lambda_0+M)\Lambda^V)^{-1}\Big)$
- For each user, $\{u_i\}_{i=1}^N$ and $k \in \{1,\dots,K^U\}$,

$$c_i^u|rest \sim N\left(\frac{\frac{c_0}{\sigma_0^2}+\sum_{j\in V_u}\frac{1}{\sigma^2}z_{ijk}^V(r_{ij}-\chi_0-d_j^{z_{ij}^U}-u_i\cdot v_j)}{\frac{1}{\sigma_0^2}+\sum_{j\in V_u}\frac{1}{\sigma^2}z_{ijk}^V}, \frac{1}{\frac{1}{\sigma_0^2}+\sum_{j\in V_u}\frac{1}{\sigma^2}z_{ijk}^V}\right)$$

- For each item, $\{v_j\}_{j=1}^M$ and $k \in \{1,\dots,K^V\}$,

$$d_j^i|rest \sim N\left(\frac{\frac{d_0}{\sigma_0^2}+\sum_{u:j\in V_u}\frac{1}{\sigma^2}z_{ijk}^U(r_{ij}-\chi_0-c_i^{z_{ij}^V}-u_i\cdot v_j)}{\frac{1}{\sigma_0^2}+\sum_{u:j\in V_u}\frac{1}{\sigma^2}z_{ijk}^U}, \frac{1}{\frac{1}{\sigma_0^2}+\sum_{j\in V_u}\frac{1}{\sigma^2}z_{ijk}^V}\right)$$

- For each user, $\{u_i\}_{i=1}^N$ and $k \in \{1,\dots,K^V\}$,

$$u_i|rest \sim N\left((\Lambda_i^{U^*})^{-1}(\Lambda^U\mu^U+\sum_{j\in V_u}\frac{1}{\sigma^2}v_j(r_{ij}-\chi_0-c_i^{z_{ij}^U}-d_j^{z_{ij}^U}), (\Lambda_i^{U^*})^{-1}\right)$$

  where $\Lambda_i^{U^*} = (\Lambda_U+\sum_{j\in V_u}\frac{1}{\sigma_2}v_jv_j^\top)$

- For each item, $\{v_j\}_{j=1}^M$ and $k \in \{1,\dots,K^U\}$,

$$v_j|rest \sim N\left((\Lambda_j^{V^*})^{-1}(\Lambda^V\mu^V+\sum_{u:j\in V_u}\frac{1}{\sigma^2}u_i(r_{ij}-\chi_0-c_i^{z_{ij}^V}-d_j^{z_{ij}^V}), (\Lambda_j^{V^*})^{-1}\right)$$

  where $\Lambda_j^{V^*} = (\Lambda_V+\sum_{u:j\in V_u}\frac{1}{\sigma_2}u_iu_i^\top)$

- For each user, $\{u_i\}_{i=1}^N$,

$$\theta_i^U|rest \sim Dir(\alpha/K^U+\sum_{j\in V_u}z_{ij}^U)$$

- For each item, $\{v_j\}_{j=1}^M$,

$$\theta_i^U|rest \sim Dir(\alpha/K^V+\sum_{u:j\in V_u}z_{ij}^V)$$

- For each user, $\{u_i\}_{i=1}^N$, and $j \in V_u$,

$$z_{ij}^U|rest \sim \text{Multi}(1,\theta_{ij}^{U^*}) \text{ where } \theta_{ij}^{U^*} \propto \theta_{ij}^U \exp\left(-\frac{(r_{ij}-\chi_0-c_i^{z_{ij}^M}-d_j^i-u_i\cdot b_j)^2}{2\sigma^2}\right)$$

- For each item, $\{v_j\}_{j=1}^M$,

$$z_{ij}^V|rest \sim \text{Multi}(1,\theta_{ij}^{V^*}) \text{ where } {\theta_{ijk}^U}^* \propto \theta_{ij}^U \exp\left(-\frac{(r_{ij}-\chi_0-c_i^k-d_j^{z_{ij}^U}-u_i\cdot b_j)^2}{2\sigma^2}\right)$$

2.2 Topic-Indexed Bias

The Topic-Indexed Bias (TIB) model assumes that the contextual bias for a particular user and item pairing decomposes into a latent user bias and latent item bias. The user bias is determined by the interaction-specific topic selected by the item. Similarly, the item bias is influenced by the user's selected topic. The contextual bias for a particular user-iterm iteraction is found by summing the two latent biases and a fixed global bias: $\beta_{ij}^{k_u k_v} = \chi_0 + c_u^{k_u} + d_j^{k_v}$.

3 Data

While I had originally planned to use playcounts of a (user, song) pair, upon examining the data set, I discovered that about 85% of streams opened on the site come from the Weekly Top 15, a curated playlist that reveals 15 different songs each week. This demonstrates that Saavn is a music streaming site that provides not only content but also and perhaps more importantly, curated playlists. Users are not exhibiting their music preferences via the site; rather, the site is molding the user's tastes. Thus, playcount seemed to be a bad proxy for preference – the data set was nearly singular. Thus, instead of using total playcounts, I extracted counts of add-to-queue events which involve a user adding a song to the current play queue that is being listened to or to a saved playlist that can be accessed at a later time.

While the full data set boasts over 12 million users and close to 1 million songs across a multitude of languages including Hindi, Telegu, English, and others, I choose to subsample from the Hindi language subset of the full data which accounts for 103,921 users and 17,346 songs. Unfortunately, because the add-to-queue data pipeline is a relatively new and unsued feature, there are only 459,396 observations. Out of the existing observations where the data density is .00025, over half the observations of counts are just 1 and while the counts range from 1 to 315, the mean of the add-to-queue counts is just 1.33 and the variance is 2.34. Other events need to be added to improve counts to better exhibit user preference for songs.

4 Experiment

With W_0 as the identity matrix, ν_0 equal to the number of static matrix factors, D, μ_0 the D-dimensional zero vector, $\chi_0 = 1.33$, the mean rating in the data set, $(c_0, d_0, \alpha) = (0, 0, 1000)$, $(\lambda_0, \sigma_0^2, \sigma^2) = (10, 5, 2.5)$. With no burn-in period, the Gibbs sampler is run for 100 iterations on D = 10 and D = 20, with the D = 20 performing slightly better than that of the D = 10 model. RMSE_{10} is 5.61 whereas the RMSE_{20} is 3.80. These results are currently misleading as virtually all of the RMSE comes from just a few observations where the add-to-queue count is quite large and more work needs to be done in terms of trying out parameter initializations.

5 Future Work

Mackey, Weiss, and Jordan [2010] point out that if $K^U and K^M$ are both set to zero, the M^3F framework is equivalent to Bayesian Probabilistic Matrix Factorization (Salakhutdinov & Mnih, 2008). While I wrote code that performs Probabilistic Matrix Factorization *Salakhutdinov & Mnih, 2006) as in , I ran out of time to implement one with hyperparameters as to verify that the M3F code is behaving as it should. There are many next steps after this initial implementation of this model beyond the parameterization. Better data that more completely represents user preference will improve this model's ability to capture the individual dynamics with respect to the user and item. More thought needs to go into the topic-indexed bias model as there is more information about user behavior than just the counts such as location of the user, medium of usage such as web versus mobile, etc. Finally, this matrix completion algorithm tool should be compared and additionally, integrated with a content-based approach to clustering similar songs.

6 References

Mackey, L., Weiss, D., and Jordan, M.I. Mixed Membership Matrix Factorization. ICML 2010.

Salakhutdinov, R. and Mnih, A. Bayesian Probabilistic Matrix Factorization using Markov Chain Monte Carlo. 2008.

Salakhutdinov, R. and Mnih, A. Probabilistic Matrix Factorization. NIPS. 2008.

7 Code

```
from __future__ import division
import numpy as np
from numpy import exp
from numpy import trace
from numpy import sqrt
from numpy.linalg import det
from numpy.linalg import pinv
from numpy.linalg import cholesky
import scipy.stats as stats
import scipy

class Wishart:
    def __init__(self, V, n):
        self.V = V
        shape = V.shape
        assert shape[0]==shape[1]
        self.p = shape[0]
        self.n = float(n)
        assert type(n) is int, 'n is not an integer'
        assert type(self.p) is int, 'p is not an integer'
        assert n > self.p-1, 'n <= p-1'
        assert not False in (V>0), 'V is not > 0'

    def pdf(self, X):
        X_shape = X.shape
        assert X_shape[0] == X_shape[1], 'X is not a square matrix'
        assert X_shape[0] == self.p
        Z_denom = 2**(-self.n*self.p/2.0)*det(self.V)**(-self.n/2.0)*exp(scipy.special.multigammaln(s
        return Z_denom*det(X)**((self.n-self.p-1)/2.0)*exp(-0.5*trace(pinv(self.V).dot(X)))

    def sample(self):
        chol = cholesky(self.V)
        if self.n <= 81 + self.p:
            x = np.random.randn(self.n, self.p)
        else:
            x = np.diag(sqrt(stats.chi2.rvs(self.n-np.arange(self.p))))
            x[np.triu_indices_from(x,1)] = np.random.nrandn(self.p*(self.p-1)/2)
        R = np.linalg.qr(x, 'r')
        T = scipy.linalg.solve_triangular(R.T, chol.T).T
        return np.dot(T, T.T)

    def sample_inv(self):
        chol = cholesky(self.V)
        if self.n <= 81 + self.p: #direct computation
            X = np.dot(chol, np.random.normal(size=(self.p, self.n)))
        else:
            A = np.diag(sqrt(np.random.chisquare(self.n - np.arange(0,self.p), size=self.p)))
            A[np.tri(self.p, k=-1, dtype=bool)] = np.random.normal(size=(self.p*(self.p-1)/2.))
            X = np.dot(chol, A)
        return np.dot(X, X.T)
```

```
import pylab
import matplotlib.pyplot as plt
import matplotlib.cm as cm

import numpy
import cPickle as pickle

class ProbabilisticMatrixFactorization():

    def __init__(self, rating_tuples, latent_d=1):
        self.latent_d = latent_d
        self.learning_rate = .0001
        self.regularization_strength = 0.1

        self.ratings = numpy.array(rating_tuples).astype(float)
        self.converged = False

        self.num_users = int(numpy.max(self.ratings[:, 0]) + 1)
        self.num_items = int(numpy.max(self.ratings[:, 1]) + 1)

        print (self.num_users, self.num_items, self.latent_d)
        print self.ratings

        self.users = numpy.random.random((self.num_users, self.latent_d))
        self.items = numpy.random.random((self.num_items, self.latent_d))

        self.new_users = numpy.random.random((self.num_users, self.latent_d))
        self.new_items = numpy.random.random((self.num_items, self.latent_d))


    def likelihood(self, users=None, items=None):
        if users is None:
            users = self.users
        if items is None:
            items = self.items

        sq_error = 0

        for rating_tuple in self.ratings:
            if len(rating_tuple) == 3:
                (i, j, rating) = rating_tuple
                weight = 1
            elif len(rating_tuple) == 4:
                (i, j, rating, weight) = rating_tuple

            r_hat = numpy.sum(users[i] * items[j])

            sq_error += weight * (rating - r_hat)**2

        L2_norm = 0
        for i in range(self.num_users):
            for d in range(self.latent_d):
                L2_norm += users[i, d]**2
```

```
    for i in range(self.num_items):
        for d in range(self.latent_d):
            L2_norm += items[i, d]**2

    return -sq_error - self.regularization_strength * L2_norm


def update(self):

    updates_o = numpy.zeros((self.num_users, self.latent_d))
    updates_d = numpy.zeros((self.num_items, self.latent_d))

    for rating_tuple in self.ratings:
        if len(rating_tuple) == 3:
            (i, j, rating) = rating_tuple
            weight = 1
        elif len(rating_tuple) == 4:
            (i, j, rating, weight) = rating_tuple

        r_hat = numpy.sum(self.users[i] * self.items[j])

        for d in range(self.latent_d):
            updates_o[i, d] += self.items[j, d] * (rating - r_hat) * weight
            updates_d[j, d] += self.users[i, d] * (rating - r_hat) * weight

    while (not self.converged):
        initial_lik = self.likelihood()

        print "  setting learning rate =", self.learning_rate
        self.try_updates(updates_o, updates_d)

        final_lik = self.likelihood(self.new_users, self.new_items)

        if final_lik > initial_lik:
            self.apply_updates(updates_o, updates_d)
            self.learning_rate *= 1.25

            if final_lik - initial_lik < .1:
                self.converged = True

            break
        else:
            self.learning_rate *= .5
            self.undo_updates()

        if self.learning_rate < 1e-10:
            self.converged = True

    return not self.converged


def apply_updates(self, updates_o, updates_d):
    for i in range(self.num_users):
        for d in range(self.latent_d):
```

```
                self.users[i, d] = self.new_users[i, d]

        for i in range(self.num_items):
            for d in range(self.latent_d):
                self.items[i, d] = self.new_items[i, d]


    def try_updates(self, updates_o, updates_d):
        alpha = self.learning_rate
        beta = -self.regularization_strength

        for i in range(self.num_users):
            for d in range(self.latent_d):
                self.new_users[i,d] = self.users[i, d] + \
                                      alpha * (beta * self.users[i, d] + updates_o[i, d])
        for i in range(self.num_items):
            for d in range(self.latent_d):
                self.new_items[i, d] = self.items[i, d] + \
                                      alpha * (beta * self.items[i, d] + updates_d[i, d])


    def undo_updates(self):
        # Don't need to do anything here
        pass


    def print_latent_vectors(self):
        print "Users"
        for i in range(self.num_users):
            print i,
            for d in range(self.latent_d):
                print self.users[i, d],
            print

        print "Items"
        for i in range(self.num_items):
            print i,
            for d in range(self.latent_d):
                print self.items[i, d],
            print


    def save_latent_vectors(self, prefix):
        self.users.dump(prefix + "%sd_users.pickle" % self.latent_d)
        self.items.dump(prefix + "%sd_items.pickle" % self.latent_d)


def plot_ratings(ratings):
    xs = []
    ys = []

    for i in range(len(ratings)):
        xs.append(ratings[i][1])
        ys.append(ratings[i][2])
```

```
    pylab.plot(xs, ys, 'bx')
    pylab.show()


def plot_latent_vectors(U, V):
    fig = plt.figure()
    ax = fig.add_subplot(121)
    cmap = cm.jet
    ax.imshow(U, cmap=cmap, interpolation='nearest')
    plt.title("Users")
    plt.axis("off")

    ax = fig.add_subplot(122)
    ax.imshow(V, cmap=cmap, interpolation='nearest')
    plt.title("Items")
    plt.axis("off")

def plot_predicted_ratings(U, V):
    r_hats = -5 * numpy.ones((U.shape[0] + U.shape[1] + 1,
                              V.shape[0] + V.shape[1] + 1))

    for i in range(U.shape[0]):
        for j in range(U.shape[1]):
            r_hats[i + V.shape[1] + 1, j] = U[i, j]

    for i in range(V.shape[0]):
        for j in range(V.shape[1]):
            r_hats[j, i + U.shape[1] + 1] = V[i, j]

    for i in range(U.shape[0]):
        for j in range(V.shape[0]):
            r_hats[i + U.shape[1] + 1, j + V.shape[1] + 1] = numpy.dot(U[i], V[j]) / 10

    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.imshow(r_hats, cmap=cm.gray, interpolation='nearest')
    plt.title("Predicted Ratings")
    plt.axis("off")
```

```
from __future__ import division
import numpy as np
import cPickle as pickle

import pylab
import matplotlib.pyplot as plt

from numpy import exp
from numpy.linalg import pinv
from numpy.random import multivariate_normal
from numpy.random import normal
from numpy.random import dirichlet
from numpy.random import multinomial

from joblib import Parallel, delayed

class M3F_TIB:
    def __init__(self, X, N, M, W_0, v_0, D, lambda_0, sigmaSqd_0, sigmaSqd, c_0, d_0, chi_0, K_U, K_
    self.X = X #X:= data is a dictionary because the data is sparse
        self.N = N
        self.M = M
        self.n_iter = n_iter
        self.D = D

        W_0_shape = W_0.shape
        assert W_0_shape[0] == W_0_shape[1], 'W_0 is not a square matrix'
        assert W_0_shape[0] == d, 'W_0 is not a square matrix of dimensionality d'
        assert v_0 > d-1

        # take inputs
        self.alpha = alpha
        self.lambda_0 = lambda_0
        self.W_0 = W_0
        self.v_0 = v_0
        self.sigmaSqd_0 = sigmaSqd_0
        self.sigmaSqd = sigmaSqd
        self.c_0 = c_0
        self.d_0 = d_0
        self.chi_0 = chi_0 # fixed global bias
        # number of topics
        self.K_U = K_U
        self.K_V = K_V
        self.mu_0 = mu_0
        # biases
        self.c = np.zeros((self.N, self.K_U))
        self.d = np.zeros((self.M, self.K_V))
        # assignments
        self.z_U = np.zeros((self.N, self.M))
        self.z_V = np.zeros((self.N, self.M))
        # probability distribution of topics
        self.theta_U = np.repeat(np.zeros(self.K_U, self.N)).reshape((self.N, self.K_U))
        self.theta_K = np.repeat(np.zeros(self.K_V, self.N)).reshape((self.M, self.K_U))

        def initial_sample(self):
```

```
    '''
    draw random sample from model prior
    initialize latent variables to model means
    Didn't do this!! initialize static factors to MAP estimates trained using stochastic gard
    set remaining variables to model means
    '''
    # posterior v's
    self.v_N = self.v_0 + self.N
    self.v_M = self.v_0 + self.M
    # precision matrices
    self.lambda_U = self.v_0 * self.W_0
    self.lambda_V = self.v_0 * self.W_0
    # mean vectors
    self.mu_U = self.mu_0
    self.mu_V = self.mu_0

    self.c = self.c + self.c_0
    self.d = self.d + self.d_0

    self.sample_u = self.mu_0
    self.sample_v = self.mu_0

    self.U = np.repeat(self.sample_u, self.N).reshape((self.N, self.D))
    self.V = np.repeat(self.sample_v, self.M).reshape((self.D, self.M))


def sample_hyperparameters(self):
    for t in xrange(self.n_iter):
        u = self.U - self.sample_u
        v = self.V - self.sample_v
        m_u = self.mu_0 - self.sample_u
        m_v = self.mu_0 - self.sample_v
        u_sample_cov = np.outer(u, u)
        v_sample_cov = np.outer(v, v)
        W_u = pinv(pinv(self.W_0) + u_sample_cov + self.lambda_0*self.N/(self.lambda_0+self.N
        W_v = pinv(pinv(self.W_0) + v_sample_cov + self.lambda_0*self.M/(self.lambda_0+self.N
    # sample hyperparameters
        # Wishart precision matrices
        self.precision_U = Wishart(W_u, self.v_N).sample()
        self.precision_V = Wishart(W_v, self.v_M).sample()
        # Gaussian means
        self.mean_U = multivariate_normal((self.lambda_0*self.mu_0 + self.N*self.sample_u)/(s
        self.mean_U = multivariate_normal((self.lambda_0*self.mu_0 + self.M*self.sample_v)/(s


def sample_topics(self):
    for i in xrange(self.N):
        for k in xrange(self.K_U):
            z_sum = 0
            mean_sum = 0
            for j in I_U[i]:
                z_sum += self.z_U[i,j]
                resid = X[(i,j)] - self.chi_0 - self.d[j,k] - np.dot(self.U[i,:], self.V[:,j]
                mean_sum += self.z_U[i,j]*resid
```

```
            std = 1./(1./self.sigmaSqd_0 + z_sum/self.Sqd)
            mean = (self.c_0/self.sigmaSqd_0 + mean_sum/self.Sqd)*std
            self.c[i,k] = normal(mean, std)

    for j in xrange(self.M):
        for k in xrange(self.K_V):
            z_sum = 0
            mean_sum = 0
            for i in I_V[j]:
                z_sum += self.z_V[i,j]
                resid = X[(i,j)] - self.chi_0 - self.c[i,k] - np.dot(self.U[i,:], self.V[:,j]
                mean_sum += self.z_V[i,j]*resid
            std = 1./(1./self.sigmaSqd_0 + z_sum/self.Sqd)
            mean = (self.c_0/self.sigmaSqd_0 + mean_sum/self.Sqd)*std
            self.d[j,k] = normal(mean, std)


def sample_user_parameters(self):
    for i in xrange(self.N):
        resid_sum = np.zeros(self.D)
        outer_product_sum = np.zeros((self.D, self.D))
        for j in I_U[i]:
            v = self.V[:,j]
            outer_product_sum += np.outer(v, v)
            resid = X[(i,j)] - self.chi_0 - self.c[i, self.z_U[i,j]] - self.d[i, self.z_U[i,j
            resid_sum += v*resid
        lambda_U_star = self.precision_U + outer_product_sum/self.sigmaSqd
        self.U[i,:] = multivariate_normal(pinv(lambda_U_star)*(self.precision_U*self.mu_U + r


def sample_item_parameters(self):
    for j in xrange(self.M):
        resid_sum = np.zeros(self.D)
        outer_product_sum = np.zeros((self.D, self.D))
        for j in I_V[i]:
            u = self.V[i,:]
            outer_product_sum += np.outer(u, u)
            resid = X[(i,j)] - self.chi_0 - self.c[i, self.z_V[i,j]] - self.d[self.z_V[i,j],
            resid_sum += u*resid
        lambda_V_star = self.precision_V + outer_product_sum/self.sigmaSqd
        self.V[:,j] = multivariate_normal(pinv(lambda_V_star)*(self.precision_V*self.mu_V + r


def sample_topic_parameters(self):
    for i in xrange(self.N):
        z_sum = 0
        for j in I_U[i]:
            z_sum += self.z_U[i,j]
        self.theta_U[i] = dirichlet(self.alpha/self.K_U + z_sum)

    for j in xrange(self.M):
        z_sum = 0
        for i in I_V[j]:
            z_sum += self.z_V[i,j]
```

```
        self.theta_V[j] = dirichlet(self.alpha/self.K_V + z_sum)

    def sample_topic_assignments(self):
        # user topics
        for i in xrange(self.N):
            for j in I_U[i]:
                theta_U_star = np.zeros(self.K_U)
                for k_idx, k in enumerate(self.z_U[i,:]):
                    theta_U_star[k_idx] = self.theta_U[i,j]*exp(-(X[(i,j)] - self.chi_0 - self.c
                self.z_U[i,j] = multinomial(1, theta_U_star/sum(theta_U_star))

        # item topics
        for j in xrange(self.M):
            for i in I_V[j]:
                theta_V_star = np.zeros(self.K_V)
                for k_idx, k in enumerate(self.z_V[:,j]):
                    theta_V_star[k_idx] = self.theta_V[i,j]*exp(-(X[(i,j)] - self.chi_0 - self.c
                self.z_V[i,j] = multinomial(1, theta_V_star/sum(theta_V_star))


    def Gibbs(self):
        self.sample_hyperparameters()
        self.sample_topics()
        self.sample_user_parameters()
        self.sample_item_parameters()
        self.sample_topic_parameters()
        self.sample_topic_assignments()

    def main(self):
        self.initial_sample()
        for i in xrange(self.n_iters):
            self.Gibbs()


    def compute_ratings(self, i, j):
        r_ij = normal(self.chi_0 + self.c[i,self.z_V[i,j]] + self.d[self.z_U[i,j], j] + np.dot(se
        return r_ij
```