STA 663: Spring 2017 - Final Project
Graph-Coupled HMMs for Modeling the Spread of Infection

Yimeng Jia *
Duke University
Durham, NC 27708
yimeng.jia@duke.edu

Dewen Xu
Duke University
Durham, NC 27708
dewen.xu@duke.edu

Abstract

HMMs (Hidden Markov Models) model independent sequenced or discrete time-series data, and represent long-range dependencies between observations for each data point, mediated via the latent variables. In previous studies, researchers in epidemiology use HMMS to model the spread of infectious disease within a certain social network at the level of entire population over discrete timestamps. However, if researchers are interested in modeling the spread of disease at individual level (interactions among people in a community are considered), the independent assumption between each data sequence does not hold, hence simple HMMs is not appropriate to simulate the generative process. Taking this into consideration, the authors develop Graph-Coupled Hidden Markov Models (GCHMMs) which allows dependencies between state transitions across multiple Hidden Markov Models (HMMs) that properly reflects the spread of infectious disease with effect of individual in a certain group. In our final project, we will implement Gibbs sampler and Generalized Baum-Welch Algorithm for GCHMMs and compare the efficiency and results of two algorithms. Keywords: Flu Diffusion, GCHMMs, Gibbs Sampler, Generalized Baum-Welch

1 Introduction

In medical informatics, one of the important topics is disease diffusion modeling, which helps to take the control of spread of diseases and to provide relevant health advice for people. In the past, due to the limitations of technology, it was difficult to collect the data reflecting the disease spreading at the individual level. However, with the appearance of mobile health apps such as built-in Health apps in apple watch, researchers can get access to the desired data for predicting the disease on an individual level. One related data collection experiment was done by MIT lab: 84 selected undergraduate students who lived in the same dormitory were assigned cell phones with sensors and a designed health App. The sensors supported by Bluetooth could record the interactions among this group of students when any of them were in a circle with a diameter of 1 meter. This group of students were supposed to report their symptoms such as runny nose, and coughing via the health app every day for 107 days. Unlike the traditional social network data, which relies on self-reported network connections, the cell phones sensors guaranteed the completeness of the whole social network throughout the experiment, which reduces the chance of missing data and enhances the robustness of statistical inferences that would be built upon later.

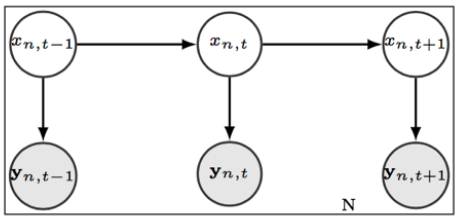
The two papers we selected to model the flu diffusion in the above situation are Graph-Coupled HMMs for Modeling the Spread of Infection written by [1] and Bayesian Models for Heterogeneous

*Our code is maintained in Github Repository, https://github.com/yimeng-jia/STA663_FinalProject_GCHMM

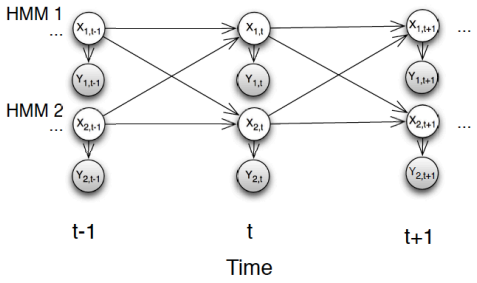
Personalized Health Data written by [2]. Traditional epidemiology research is mainly focused on the infectious simulation on a large population. However, the GCHMMs invented and developed by Dong and Kai allow researchers to model the spread of infectious disease locally within a smaller social network, such as a community or a dormitory. By incorporating a dynamic social network into a coupled HMMs, the GCHMMs can successfully predict health condition of each individual and monitor the path of illness transmission through individuals in a social network, which help researchers to understand how disease is transmitted locally within a network, and to give advice treatment on an individual level.

In CHMMs, the latent state of HMM i at time t is dependent on the latent states of all HMMs in the CHMM at time t-1. In GCHMM, the latent state of HMM i at time t depend on the any number of HMMs that have edges in the graph connected to node n.

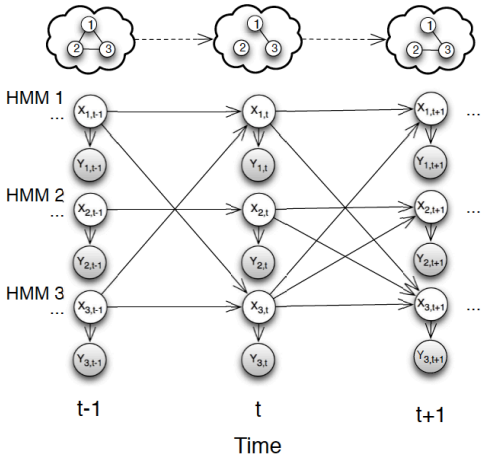
GCHMMs (figure 1c) is a dynamic model combining a number of HMMs (figure 1a) together by adding directed edges among hidden states. Each HMM represents one time-series data (a student in our epidemiology model) and its latent/observed state at time t only depends on the latent/observed state at time t-1. Hidden state is the latent variable indicating whether a student is infectious or not at current time, while observed state is the symptoms behavior. Both the two states are discrete data type in the model. Unlike Coupled-HMMs (figure 1b) which latent state of HMM i at time t is dependent on the latent states of all HMMs at time t-1, in the GCHMM the latent state of HMM i at time t depend on the any number of HMMs that have edges in the graph connected to node n. The edges among different HMMs at different timesteps are described by dynamic network graphs (cloud-shape figures above each time step in figure 1c), describing all the interactions between each consecutive timestamp. For example, during time t to $t+1$, student x_1 and student x_2 contacts to student x_3 which are reflected by the third network graph (node1 and node2 are connected to node3) above time stamp $t+1$.



(a) Simple Hidden Markov Model



(b) Coupled Hidden Markov Model



(c) Graph Coupled Hidden Markov Model

Figure 1: Various HMMs

Notations

- $n : 1, \ldots, N$ index for each HMMs
- $t : 1, \ldots, T$ index for timestamp
- $s : 1, \ldots, S$ index for symptoms of observation
- γ recovery probability if infectious at pervious time
- α probability being infected from some one outside networks
- β probability being infected from some one inside networks
- $x_{n,t} \in \mathcal{X} = \{0, 1\}$ the hidden state whether infectious
- $y_{n,t} \in \mathcal{Y} = \{0, 1\}^S$ the symptoms onset
- ξ hyper parameter for the prior Bernoulli distribution of $x_{n,1}$
- $\theta_{x_{n,t},s}$ emission probability of symptom s onset
- G_t the dynamic social network graph between time t and $t + 1$
- E_t the set of edge on G_t

2 Generative Model

2.1 Probablistic Model

A coupled hidden Markov model (CHMM) describes the dynamics of a discrete-time Markov process that links together a number of distinct standard hidden Markov models (HMMs). In a standard HMM, the value of the latent state at time t is dependent on only the value of the latent state at last time $t - 1$. In contrast, the latent state of GCHMMs at time t in the GCHMMs is dependent on the latent states of a subset of HMMs in the GCHMMs at time $t - 1$, where the subset is determined by the graph at time $t - 1$. Therefore, the generative model of GCHMMs can be readily stated.

$$
\begin{aligned}
\xi &\sim \text{Beta}(a_\xi, b_\xi) & \alpha &\sim \text{Beta}(a_\alpha, b_\alpha) \\
\beta &\sim \text{Beta}(a_\beta, b_\beta) & \gamma &\sim \text{Beta}(a_\gamma, b_\gamma) \\
\theta_{0,s} &\sim \text{Beta}(a_0, b_0) & \theta_{1,s} &\sim \text{Beta}(a_1, b_1) \\
x_{n,0} &\sim \text{Bernoulli}(\xi) & x_{n,t} &\sim \text{Bernoulli}\left(\phi_{n,x_{n'}:(n,n')\in G_t}(\alpha, \beta, \gamma)\right) \\
y_{n,t,s} &\sim \text{Bernoulli}(\theta_{x_{n,t},s})
\end{aligned}
$$

According to the definition of $\xi, \alpha, \beta, \gamma$, they all represent small probabilities so their prior should have a small mean in application. $\theta_{1,s}$ means the symptom appearance probability when got infectious, thus being reasonable to set a higher mean prior. $\theta_{0,s}$ is on the contrary. Notice that the transition probability $\phi_{n,x_{n'}:(n,n')\in G_t}(\alpha, \beta, \gamma)$ here is a function of three parameters. As the notations, this term is a little subtle because of its dependence on previous infection state. The exact definition will be introduced in the factor graph section.

2.2 Transition Function

According to Figure 1(c), the bayesian network has two type of factors (conditional probability), emission probability function $\phi_{n,t,y|x,s}(y_{n,t,s}|x_{n,t}) = p(y_{n,t,s}|x_{n,t})$ and transition probability function $\phi_{n,t-1,t}(x_{n,t-1}, x_{n',t-1:(n',n)\in E_{t-1}}, x_{n,t}) = p(x_{n,t}|x_{n,t-1}, x_{n',t-1:(n',n)\in E_{t-1}})$. Based on the generative model, the factors have the following expression.

$$
\phi_{n,t,y|x,s}(y_{n,t,s}|x_{n,t}) = \theta_{x_{n,t},s}^{y_{n,t,s}}(1 - \theta_{x_{n,t},s})^{y_{n,t,s}}
$$

$$
\phi_{n,t-1,t}(x_{n,t-1}, x_{n',t-1:(n',n)\in E_{t-1}}, x_{n,t}) = \begin{cases} \gamma & \text{if } x_{n,t-1} = 1, x_{n,t} = 0 \\ 1 - \gamma & \text{if } x_{n,t-1} = 1, x_{n,t} = 1 \\ 1 - (1 - \alpha)(1 - \beta)^{\sum_{n':(n',n)\in E_{t-1}} x_{n',t-1}} & \text{if } x_{n,t-1} = 0, x_{n,t} = 1 \\ (1 - \alpha)(1 - \beta)^{\sum_{n':(n',n)\in E_{t-1}} x_{n',t-1}} & \text{if } x_{n,t-1} = 0, x_{n,t} = 0 \end{cases}
$$

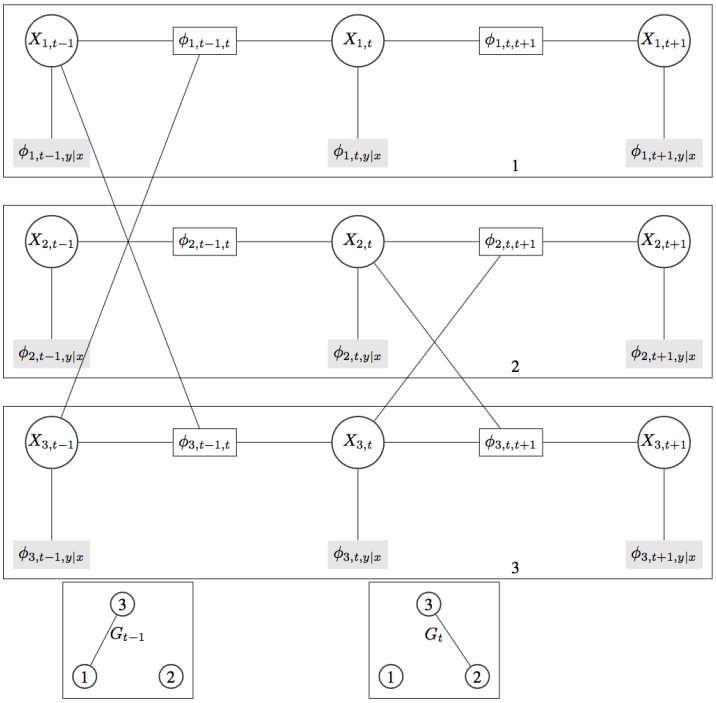


Figure 2: Factor Graph of GCHMMs

With the factor definition, the standard bayesian network can be easily represented to factor graph Figure 2. The factor graph is still shown by template formulation; each template represents a patient while the interaction is captured by transition factor. In fact, during the later EM algorithm, all parameters are unknown so that we can put parameter node containing α, β, γ at the top of the current whole graph, having connection edge with all transition factors. Similarly, parameter nodes ξ and θ_0, θ_1 can be added in the same way. For simplicity, we did not include them in our factor graph, though this parameter node trick is a commonly used method.

3 Gibbs Sampling Algorithm

3.1 Update latent sequence $x_{1:D}$

We can easily write the joint likelihood as follows.

$$P(\mathbf{Y}_{n,1:T}, \mathbf{X}_{n,0:T}) = \boldsymbol{\pi}_0(X_{n,0}) P(X_{n,1}|X_{n,0}, \phi) P(Y_{n,1}|X_{n,1}, \theta) \prod_{t=2}^{T} P(X_{n,t}|X_{n,t-1}, X_{n':\{n',n\}\in E_{t-1},t-1}, \phi) P(Y_{n,t}|X_{n,t}, \theta)$$

Thus, the posterior for X.

$$X_{n,t+1}|X, Y \setminus X_{n,t+1;\mu} \sim \text{Categorical}\left(\frac{P(X_{n,t+1})}{\sum_x P(X_{n,t+1=x})}\right)$$

3.2 Update parameters in transition distribution

The Baysian model is not conjugate for parameters, so we introduce an auxiliary latent variable R to make our model approximately conjugate. The augumented variable R means

$$R_{n,t} \sim \text{Categorical}\left(\frac{\alpha, \beta, ..., \beta}{\alpha + \beta \sum_{n'} 1_{(n',n)\in E_t \cap X_{n',t}=1}}\right)$$

$$\alpha|X \sim \text{Beta}(a + \textstyle\sum_{n,t} 1_{\{R_{n,t=1}\}}, b + \sum_{n,t:X_{n,t=0}} 1 - \sum_{n,t} 1_{\{R_{n,t=1}\}})$$

$$\beta|X \sim \text{Beta}(a' + \textstyle\sum_{n,t} 1_{\{R_{n,t>1}\}}, b' + \sum_{n,t:X_{n,t=0;n'}} 1_{(n',n)\in E_t \bigcap X_{n',t}=1-\sum_{n,t} 1_{\{\{R_{n,t>1}\}}})$$

$$\gamma|X \sim \text{Beta}(a'' + \textstyle\sum_{n,t:X_{n,t=1}} 1_{\{X_n,t+1=0\}},\ b'' + \sum_{n,t:X_{n,t=1}} 1 - \sum_{n,t:X_n,t=1} 1_{\{X_{n,t+1=0}\}})$$

3.3 Update parameters in emission distribution

$p(\boldsymbol{\theta_{k,s}}|Y) \sim \text{Beta}(a_k + \sum \mathbb{I}_{Y_{n,t,s}=1,X_{n,t}=k}, b_k + + \sum \mathbb{I}_{Y_{n,t,s}=1,X_{n,t}=k})$, for $k = 0, 1$

4 Generalized Baum-Welch (GBW) Algorithm

Two main contributions in Fan's generalized the Baum-Welch Algorithm for bounded GCHMMs are included in this section: the first is to use belief propagation to derive the forward-backward algorithm that is parallel in standard HMMs; the second is to apply an approximate EM algorithm for parameter estimation.

4.1 Single Node Belief Propagation Rules: Forward-backward

Notice that even if the Bayesian network Figure is not exactly a directed polytree (cyclic path exists if direction is omitted), marginals inference on each hidden node can still be approximated by belief propagation by factor graph for the sake of efficiency. In this section, we will show the derivation of the single node belief probation rules on Figure 2. Denote the message passing to the child, i.e. from $x_{m,t-1}$ to $x_{n,t}$ as $\pi_{x_{m,t-1}}(x_{n,t})$, and the message to parent, i.e. from $x_{k,t+1}$ to $x_{n,t}$ as $\lambda_{x_{k,t+1}}(x_{n,t})$. Then, the belief or probability distribution passed with all evidence shown is denoted as $BEL(x_{n,t}) = P(x_{n,t}|Y)$. Our derivation is based on Pearl's belief propagation algorithm [3]. All $\propto$ in the following imply the term should be normalized to 1 as a probability distribution.

It has been proved that belief propagation on standard HMMs is equivalent to forward-backward algorithm. It is still similar to write down forward-backward framework in our coupled case. However, the update step for message passing here is a little complicated since the single node dependence is generalized to multi-nodes. Here we need the assumption maximum degree of all G_ts is bound by M. In this case, the update of sum-product for message can be computed efficient, i.e. $\mathcal{O}(2^M)$ at each iteration.

$$\pi^{(i)}(x_{n,t}) = \sum_{x_{n,t-1}, x_{n',t-1:(n'n)\in E_t}} \phi_{n,t-1,t} \prod_{n\cup\{n':(n',n)\in E_{t-1}\}} \pi^{(i)}_{x_{n,t}}(x_{\cdot,t-1})$$

$$\lambda^{(i)}(x_{n,t}) = \prod_{s=1}^{S} \lambda_{y_{n,t,s}}(x_{n,t}) \prod_{n\cup\{n':(n,n')\in E_t\}} \lambda^{(i)}_{x_{\cdot,t+1}}(x_{n,t})$$

$$BEL^{(i)}(x_{n,t}) \propto \pi^{(i)}(x_{n,t})\lambda^{(i)}(x_{n,t})$$

where

$$\pi_{x_{n,t}}(x_{\cdot,t-1}) = p(x_{n,t}|x_{\cdot,t-1}) \text{ and } \lambda_{x_{\cdot,t+1}}(x_{n,t}) = p(x_{\cdot,t+1}|x_{n,t}).$$

Updating λ

$$\lambda^{(i+1)}_{x_{n,t}}(x_{n,t-1}) \propto \sum_{x_{n,t}} \lambda^{(i)}(x_{n,t}) \sum_{x_{n',t-1}:(n',n)\in E_{t-1}} \phi_{n,t-1,t} \prod_{n':(n',n)\in E_{t-1}} \pi^{(i)}_{x_{n,t}}(x_{\cdot,t-1})$$

$$\lambda_{x_{n,t}}(x_{n,t-1}=1) \propto \sum_{x_{n,t}} \lambda(x_{n,t})\gamma^{\mathbb{I}_{x_{n,t}=0}}(1-\gamma)^{\mathbb{I}_{x_{n,t}=1}}$$

$$\lambda_{x_{n,t}}(x_{n,t-1}=0) \propto \sum_{x_{n,t}} \lambda(x_{n,t}) \sum_{x_{n',t-1}:(n',n)\in E_{t-1}} \left[1-(1-\alpha)(1-\beta)^{\sum_{n':(n',n)\in E_{t-1}} x_{n',t-1}}\right]^{\mathbb{I}_{x_{n,t}=1}}$$
$$\left[(1-\alpha)(1-\beta)^{\sum_{n':(n',n)\in E_{t-1}} x_{n',t-1}}\right]^{\mathbb{I}_{x_{n,t}=0}} \prod_{n':(n',n)\in E_{t-1}} \pi_{x_{n,t}}(x_{\cdot,t-1})$$

$$\lambda^{(i+1)}_{x_{n,t}}(x_{n',t-1}) \propto \sum_{x_{n,t}} \lambda^{(i)}(x_{n,t}) \sum_{x_{n,t-1},x_{n'',t-1}\neq x_{n',t-1}:(n'',n)\in E_{t-1}} \phi_{n,t-1,t} \prod_{n\cup\{n''\neq n':(n'',n)\in E_{t-1}\}} \pi^{(i)}_{x_{n,t}}(x_{\cdot,t-1})$$

Updating π

$$\pi^{(i+1)}_{x_{n,t+1}}(x_{n,t}) \propto \prod_{s=1}^{S} \lambda_{y_{n,t,s}}(x_{n,t}) \prod_{n':(n,n')\in E_t} \lambda^{(i)}_{x_{\cdot,t+1}}(x_{n,t})\pi^{(i)}(x_{n,t}) = \frac{BEL^{(i)}(x_{n,t})}{\lambda^{(i)}_{x_{n,t+1}}(x_{n,t})}$$

$$\pi^{(i+1)}_{x_{n',t+1}}(x_{n,t}) \propto \prod_{s=1}^{S} \lambda_{y_{n,t,s}}(x_{n,t}) \prod_{n\cup\{n''\neq n':(n,n'')\in E_t\}} \lambda^{(i)}_{x_{\cdot,t+1}}(x_{n,t})\pi^{(i)}(x_{n,t}) = \frac{BEL^{(i)}(x_{n,t})}{\lambda^{(i)}_{x_{n',t+1}}(x_{n,t})}$$

$$\pi^{(i+1)}_{y_{n,t,s}}(x_{n,t}) \propto \prod_{s'\neq s} \lambda_{y_{n,t,s'}}(x_{n,t}) \prod_{n\cup\{n':(n,n')\in E_t\}} \lambda^{(i)}_{x_{\cdot,t+1}}(x_{n,t})\pi^{(i)}(x_{n,t}) = \frac{BEL^{(i)}(x_{n,t})}{\lambda^{(i)}_{y_{n,t,s}}(x_{n,t})}$$

Boundary Conditions

- Root nodes, i.e. $x_{n,0}$. Let π be the prior distribution, i.e. $\pi(x_{n,0}) = \xi^{x_{n,0}}(1-\xi)^{1-x_{n,0}}$.
- Evidence nodes, i.e. $y_{n,t,s}$

$$\begin{aligned}
\lambda(y_{n,t,s}) &= (\mathbb{I}_{y_{n,t,s}=0}, \mathbb{I}_{y_{n,t,s}=1}) \\
\pi(y_{n,t,s}) &= \sum_{x_{n,t}} \phi_{n,t,y|x,s}(y_{n,t,s}|x_{n,t})\pi_{y_{n,t,s}}(x_{n,t}) \\
BEL(y_{n,t,s}) &\propto \lambda(y_{n,t,s})\pi(y_{n,t,s}) \\
\lambda_{y_{n,t,s}}(x_{n,t}) &\propto \sum_{y_{n,t,s}} \lambda(y_{n,t,s})\phi_{n,t,y|x,s}(y_{n,t,s}|x_{n,t}) = \mathbb{I}_{y_{n,t,s}=0}(1-\theta_{x_{n,t},s}) + \mathbb{I}_{y_{n,t,s}=1}\theta_{x_{n,t},s}
\end{aligned}$$

- Initialization, all message from variable nodes to factor nodes, such as $\pi_{x_\cdot}(x_\cdot), \lambda_{x_\cdot}(x_\cdot)$, can be set all 1s. Actually, random initialization would affect much of the result because they all will be washed out after a few iterations.

4.2 Message Passing for Approximate EM Algorithm

In this section, we will put forward two algorithms for parameter learning, particularly in the generalized Baum-Welch Algorithm for GCHMMs. Gibbs sampling algorithm is another widely investigated in this phrase [1]. A comparison between the two algorithms will be discussed in the experiment section.

4.2.1 E-step

$$\Pr(X,Y;\Theta)=\prod_{n=1}^{N}\left\{p(x_{n,0};\xi)\left(\prod_{s=1}^{S}\phi_{n,0,y|x,s}\right)\prod_{t=1}^{T}\left[\phi_{n,t-1,t}\left(\prod_{s=1}^{S}\phi_{n,t,y|x,s}\right)\right]\right\}$$

$$\log\Pr(X,Y;\Theta)=\sum_{n=1}^{N}\left\{\log p(x_{n,0};\xi)+\sum_{t=1}^{T}\log\phi_{n,t-1,t}+\sum_{t=1}^{T}\sum_{s=1}^{S}\log\phi_{n,t,y|x,s}\right\}$$

$$Q(\Theta,\Theta^{old})=\sum_{X}\sum_{n=1}^{N}\left\{x_{n,0}\log\xi+(1-x_{n,0})\log(1-\xi)+\sum_{t=1}^{T}\log\phi_{n,t-1,t}+\sum_{t=1}^{T}\sum_{s=1}^{S}\log\phi_{n,t,y|x,s}\right\}\Pr(X|Y,\Theta^{old})$$

4.2.2 Exact M-step

$$\frac{\partial Q(\Theta,\Theta^{old})}{\partial\xi}=\sum_{x_{1:N,0}\in\{0,1\}^N}\left(\frac{\sum_{n=1}^{N}x_{n,0}}{\xi}-\frac{\sum_{n=1}^{N}(1-x_{n,0})}{1-\xi}\right)p(x_{1:N,0}|Y,\Theta^{old})=0 \quad (1)$$

$$\Rightarrow\xi=\frac{\sum_{x_{1:N,0}\in\{0,1\}^N}\left(\sum_{n=1}^{N}x_{n,0}\right)p(x_{1:N,0}|Y,\Theta^{old})}{N}=\frac{\sum_{n=1}^{N}\mathbb{E}[x_{n,0}]}{N} \quad (2)$$

$$\frac{\partial Q(\Theta,\Theta^{old})}{\partial\theta_{0,s}}=0\Rightarrow\theta_{0,s}=\frac{\sum_{x_{1:N,1:T}\in\{0,1\}^{N^2}}\left(\sum_{n=1}^{N}\sum_{t=1}^{T}y_{n,t,s}\mathbb{I}_{x_{n,t=0}}\right)p(X|Y,\Theta^{old})}{\sum_{x_{1:N,1:T}\in\{0,1\}^{N^2}}\left(\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{x_{n,t=0}}\right)p(X|Y,\Theta^{old})}$$

Similarly, $\theta_{1,s}=\frac{\sum_{x_{1:N,1:T}\in\{0,1\}^{N^2}}\left(\sum_{n=1}^{N}\sum_{t=1}^{T}y_{n,t,s}\mathbb{I}_{x_{n,t=1}}\right)p(X|Y,\Theta^{old})}{\sum_{x_{1:N,1:T}\in\{0,1\}^{N^2}}\left(\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{x_{n,t=1}}\right)p(X|Y,\Theta^{old})}$

$$\frac{\partial Q(\Theta,\Theta^{old})}{\partial\gamma}=0\Rightarrow\gamma=\frac{\sum_{x_{1:N,1:T}\in\mathcal{X}^{N^2}}\left(\sum_{n=1}^{N}\sum_{t=2}^{T}\mathbb{I}_{x_{n,t-1}=1,x_{n,t}=0}\right)p(X|Y,\Theta^{old})}{\sum_{x_{1:N,1:T}\in\mathcal{X}^{N^2}}\left(\sum_{n=1}^{N}\sum_{t=2}^{T}\mathbb{I}_{x_{n,t-1}=1}\right)p(X|Y,\Theta^{old})}$$

Notice that it does not matter if we change the $p(X|Y,\Theta^{old})$ to $\Pr(X,Y|\Theta^{old})$. Conditions $\frac{\partial Q(\Theta,\Theta^{old})}{\partial\alpha}=0$ and $\frac{\partial Q(\Theta,\Theta^{old})}{\partial\beta}=0$ can be further derived for the iteration step for parameters estimation by EM. However, except for ξ, the exact computation complexity of other parameters iteration step is an intractable message passing, exponentially increasing with N or N^2.

4.2.3 Approximate M-step

If we approximate $P(X|Y,\Theta^{old})=\prod_{n,t}p(x_{n,t}|Y,\Theta^{old})$ as a fully factorized form, then all the M-step for θ would be allowed to update easily, because $p(x_{n,t}|Y,\Theta^{old})$ (i.e. $BEL(x_{n,t})$) can be computed by forward-backward algorithm derived before.

$$\theta_{0,s}=\frac{\sum_{n=1}^{N}\sum_{t=1}^{T}y_{n,t,s}\mathbb{E}[1-x_{n,t}]}{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{E}[1-x_{n,t}]} \quad (3)$$

$$\theta_{1,s}=\frac{\sum_{n=1}^{N}\sum_{t=1}^{T}y_{n,t,s}\mathbb{E}[x_{n,t}]}{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{E}[x_{n,t}]} \quad (4)$$

Updating for γ,α,β is tricky here. First, we first introduce the approximation for γ which will make the two later parameters more understandable. Even if the full factorization assumption holds, the update step for γ associates two successive variables $x_{n,t-1}$ and $x_{n,t}$. A natural idea is to use Monte Carlo method to sample $\{\tilde{x}_{1:N,1:T}\}\in\mathcal{X}^{N^2}$ from $P(X|Y,\Theta^{old})=\prod_{n,t}p(x_{n,t}|Y,\Theta^{old})$. Then we count the number that event $x_{n,t-1}=1,x_{n,t}=0$ happens. For simplicity, we can directly assign

the simulated sample by Bayesian decision strategy according to each $p(x_{n,t}|Y,\Theta^{old})$ instead of sampling. That is to say, we only need to set the sample $x_{n,t} = \arg\max_{\tilde{x}_{n,t}=\{0,1\}} p(x_{n,t}|Y,\Theta^{old})$. Therefore, the update for γ is straightforward.

$$\gamma = \frac{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{\tilde{x}_{n,t-1}=1,\tilde{x}_{n,t}=0}}{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{E}[x_{n,t-1}]} \approx \frac{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{\tilde{x}_{n,t-1}=1,\tilde{x}_{n,t}=0}}{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{\tilde{x}_{n,t-1}=1}} \tag{5}$$

The same trick can be applied to update α, β. However, one more trick needed here. Denote $\tau_i = (1-\alpha)(1-\beta)^i$; then apparently $\alpha = 1 - \tau_0$. Then the update step for α is similar as γ.

$$\begin{aligned}\alpha &= \frac{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{\tilde{x}_{n,t-1}=0,\tilde{x}_{n,t}=1}\mathbb{I}_{\sum_{n':(n,n')\in E_{t-1}}\tilde{x}_{n',t-1}=0}}{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{E}[1-x_{n,t-1}]} \\ &\approx \frac{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{\tilde{x}_{n,t-1}=0,\tilde{x}_{n,t}=1}\mathbb{I}_{\sum_{n':(n,n')\in E_{t-1}}\tilde{x}_{n',t-1}=0}}{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{\tilde{x}_{n,t-1}=0}}\end{aligned} \tag{6}$$

Following the trick of α, we can easily update $\tau_i, i = 1, ..., M$.

$$\begin{aligned}\tau_i &= \frac{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{\tilde{x}_{n,t-1}=0,\tilde{x}_{n,t}=0}\mathbb{I}_{\sum_{n':(n,n')\in E_{t-1}}\tilde{x}_{n',t-1}=i}}{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{E}[1-x_{n,t-1}]} \\ &\approx \frac{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{\tilde{x}_{n,t-1}=0,\tilde{x}_{n,t}=0}\mathbb{I}_{\sum_{n':(n,n')\in E_{t-1}}\tilde{x}_{n',t-1}=i}}{\sum_{n=1}^{N}\sum_{t=1}^{T}\mathbb{I}_{\tilde{x}_{n,t-1}=0}}\end{aligned}$$

Obviously, $\beta = 1 - \left(\frac{\tau_i}{1-\alpha}\right)^{\frac{1}{i}}$, thus meaning we have M estimations for βs. How to combine these βs to obtain a better estimation may vary in different applications. So the below algebra average can be adjusted under various circumstances.

$$\beta = 1 - \frac{1}{M}\sum_{i=1}^{M}\left(\frac{\tau_i}{1-\alpha}\right)^{\frac{1}{i}}. \tag{7}$$

To sum up, we have our generalized Baum-Welch Algorithm, also called approximate EM Belief propagation(AEMBP), for GCHMMs.

(I) $\Theta^{(0)}$ initialization;

(II) Run one iteration forward-backward algorithm on Θ^{old} to obtain $p(x_{n,t}|Y,\Theta^{old})$;

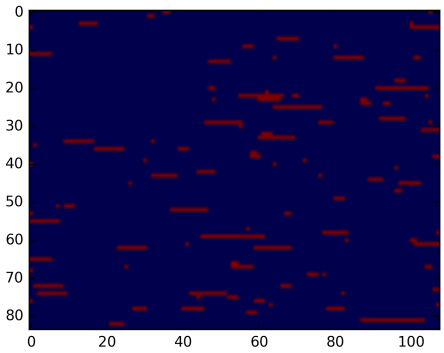
(III) Update Θ^{new} based on equation (2)-(7);

(IV) If converge, break; else back to (II).

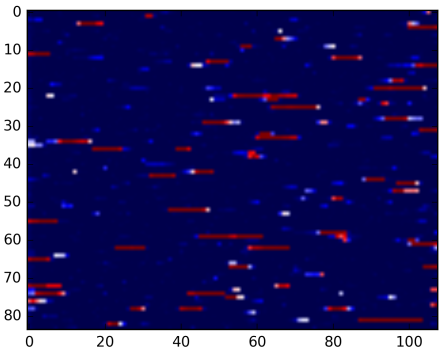
5 Experimental Results and Comparative Analysis

5.1 Application to Real Data Set

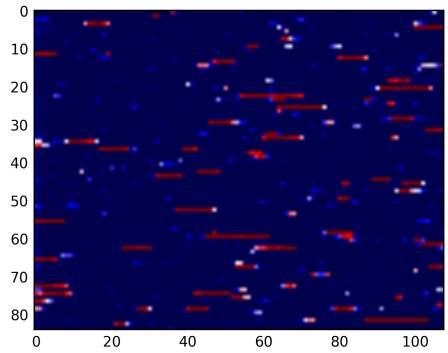
Latent States Prediction With the dynamic social network data $G_{84\times84\times107}$, we run the data on two algorithm Generalized BW (GBW) and Gibbs sampling under two cases, with parameter unknown. Both algorithms will only take Y but not X as arguments but will predict an X. To observe the performance on real data set, we construct heat maps of the predicted latent states (picture on the middle and right) and compare it with the true latent states (picture on the left). Redness indicates that the student (on the y-axis) get infected on a particular day (x-axis); blueness indicate no infection; and whiteness indicate that the probability of getting infected is 0.5. Overall, Gibbs sampler and Generalized Baum-Welch algorithm achieve very similar predictions of the latent states as the real case, except a few whiteness at the end of an infection line, indicating that the models are uncertain whether the infected student is ready to recover. This makes sense in the real situation because sometimes doctors are unsure when students should be diagnosed as completely recovered at the end of an infection period. Overall, both algorithms achieve comparable performance.



(a) True Latent States



(b) Predicted Latent States from Gibbs



(c) Predicted Latent States from Approx. EM B-W

Figure 3: Latent States (X) Predictions

5.2 Comparative Analysis

The parameter estimation results by Gibbs sampling with 500 iterations and GBW algorithm with 15 iterations is shown in Table 1. Notice that Gibbs sampling implemented in the experiments burns in half of the total number of iterations. Both algorithms result in similar parameter estimations. Generalized BW algorithm shows better performance than Gibbs sampling for less iterations (5-10 iterations for GBW, 500 iterations for Gibbs), while Gibbs sampling converges with less time (will be discussed in section Profiling and Optimization). Although as we will discussed in section Code Testing, Gibbs Sampling is subjective to prior choices, an excellent performance of GBW is even largely dependent on the initialization of parameters. If the initializations are chosen inappropriate, GBW may harm the result, which is the common disadvantage of EM algorithm.

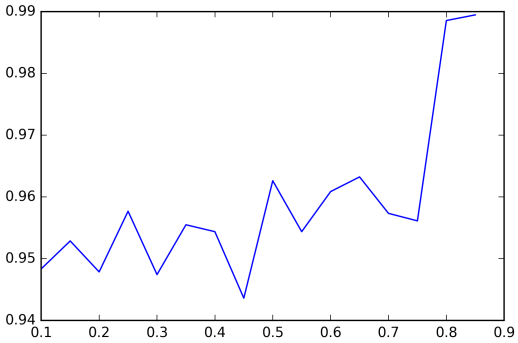
Table 1: Parameter Estimation

Θ	ξ	α	β	γ	$\theta_{1,1}$	$\theta_{1,2}$	$\theta_{1,3}$	$\theta_{1,4}$	$\theta_{1,5}$	$\theta_{1,6}$	$\theta_{0,1}$	$\theta_{0,2}$	$\theta_{0,3}$	$\theta_{0,4}$	$\theta_{0,5}$	$\theta_{0,6}$
Gibbs-500	0.133	0.009	0.016	0.223	0.666	0.597	0.683	0.471	0.596	0.638	0.207	0.196	0.315	0.230	0.416	0.067
GBW-15	0.095	0.007	0.002	0.181	0.680	0.624	0.703	0.494	0.606	0.680	0.209	0.197	0.316	0.230	0.416	0.068

5.3 Application to Simulated Data Set

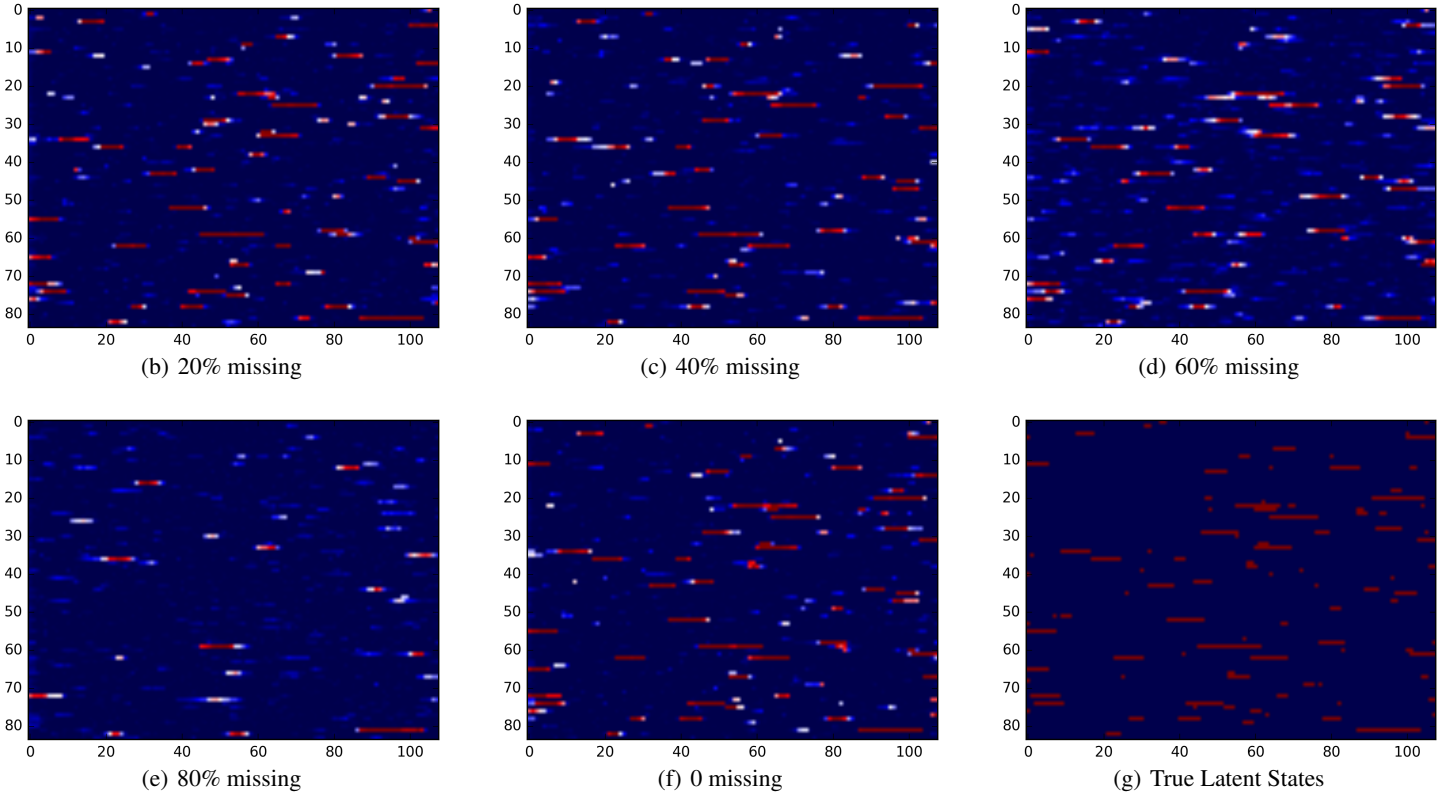
Missing observed states imputation Due to privacy of the participants, we do not have access to the original clinical data. The dataset we have at hand is already cleaned with missing values imputed. To mimic the real situation when students are too lazy to report the 6 symptoms everyday, we simulate from this data set a proportion of student-day combinations and set the observed states (Y) of those as missing. Meanwhile, we add an additional step in Gibbs Sampler loops to sample these observed states (Y) from Bernoulli distribution with certain probability (θ_1 if infected, θ_0 if not infected). Since the actual number of students who have and also report these symptoms is sparse, the majority positions of Y are 0's. Therefore we are interested in whether our model commits type II error – predicting none when the student truly has a symptom, and we compare the performances with missing y using FNR (False Negative Rate) as we have true value of Y for the predicted ones in this case. As we can see, FNR increases modestly with small missing rate, and dramatically when missing rate larger than 0.7. This is because the Y matrix is sparse and therefore the simulated missing are mostly 0s as well. Setting those as missing does not affect our prediction much. However, when missing rate is as high as 0.8, we are losing valuable information by ignoring a considerable portion of 1s as missing.

Latent states prediction with missing Y Our explanation is also verified by the latent states prediction heat maps with missing values. When missing rate is 20% or 40%, the model achieves more or less the same result. When missing rate is 60%, the model has more noise (light blueness



(a) False Negative Rate

indicating that the student still has a less than 0.5 probability of being infected). When missing rate is 80%, the model is very unconfident in predicting anyone as infected



(b) 20% missing (c) 40% missing (d) 60% missing

(e) 80% missing (f) 0 missing (g) True Latent States

Figure 4: Prediction of X with different Y missing rates.

6 Profiling and Optimization

In this part, we performed both profiling and optimization on the Gibbs Samplers and Generalized Baum-Welch algorithm.

6.1 Profiling and Optimization for Gibbs Sampler

6.1.1 Naive Version

For the Gibbs Samplers, we wrote our algorithm in numpy for large parts, but left the core matrix multiplication function NumPreInf in nested for loop form for the further optimization. Our first naive version code consisted of one main function and seven small functions.

```
p = pstats.Stats('work_naive.prof')
p.sort_stats('ncalls').print_stats(10)
pass
```

```
         894724 function calls in 2113.522 seconds

   Ordered by: call count
   List reduced from 74 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   214005    0.711    0.000    0.711    0.000 {built-in method numpy.core.multiarray.zeros}
   110004    0.241    0.000    0.241    0.000 {method 'reshape' of 'numpy.ndarray' objects}
   107000 2093.518    0.020 2094.229    0.020 <ipython-input-9-70fe7ab18de4>:1(NumPreInf)
    54500    0.816    0.000    0.816    0.000 {method 'rand' of 'mtrand.RandomState' objects}
    53000   11.580    0.000   12.555    0.000 <ipython-input-8-f7f4b9e9c716>:91(updateIntermediaX)
    47295    0.062    0.000    0.062    0.000 {built-in method numpy.core.multiarray.array}
    20743    0.004    0.000    0.004    0.000 {built-in method builtins.len}
    20000    0.005    0.000    0.005    0.000 /opt/conda/lib/python3.5/site-packages/numpy/lib/stride_tricks.py:62(<g
enexpr>)
    18512    0.249    0.000    0.249    0.000 {method 'reduce' of 'numpy.ufunc' objects}
    17030    0.009    0.000    0.009    0.000 /opt/conda/lib/python3.5/site-packages/numpy/lib/stride_tricks.py:193(<
genexpr>)
```

We used over 2000 second to complete profiling on this naive version and found the NumPreInf function taking up almost all the running time. From this result, it was infeasible to write the whole code in for loop, and we were focus on optimizing NumPreInf function by serval methods.

6.2 JIT Optimization

we just put the JIT decorator on the chunk of NumPreInf function, and the total running times was reduced to 16.169 seconds.

```
p = pstats.Stats('work_jit.prof')
p.sort_stats('ncalls').print_stats(10)
pass
```

```
         790390 function calls (768602 primitive calls) in 16.619 seconds

   Ordered by: call count
   List reduced from 1331 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   110004    0.111    0.000    0.111    0.000 {method 'reshape' of 'numpy.ndarray' objects}
    54500    0.302    0.000    0.302    0.000 {method 'rand' of 'mtrand.RandomState' objects}
    53000    6.175    0.000    6.518    0.000 <ipython-input-26-765b98198d20>:91(updateIntermediaX)
    47295    0.061    0.000    0.061    0.000 {built-in method numpy.core.multiarray.array}
    37758    0.012    0.000    0.015    0.000 {built-in method builtins.isinstance}
    23132    0.006    0.000    0.006    0.000 {built-in method builtins.len}
    20000    0.006    0.000    0.006    0.000 /opt/conda/lib/python3.5/site-packages/numpy/lib/stride_tricks.py:62(<g
enexpr>)
    18512    0.247    0.000    0.247    0.000 {method 'reduce' of 'numpy.ufunc' objects}
    17030    0.009    0.000    0.009    0.000 /opt/conda/lib/python3.5/site-packages/numpy/lib/stride_tricks.py:193(<
genexpr>)
    15518    0.016    0.000    0.043    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/numeric.py:484(asanya
rray)
```

6.3 Cython Optimization

Second, we re-wrote this function into Cython format, and the total running times was 16.545 seconds.

```
p = pstats.Stats('work_cython.prof')
p.sort_stats('ncalls').print_stats(10)
pass
```

```
         680724 function calls in 16.545 seconds

   Ordered by: call count
   List reduced from 74 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   110004    0.152    0.000    0.152    0.000 {method 'reshape' of 'numpy.ndarray' objects}
   107000    3.332    0.000    3.332    0.000 {_cython_magic_2e9979e66fda0156cc93982206afab6e.NumPreInf}
    54500    0.335    0.000    0.335    0.000 {method 'rand' of 'mtrand.RandomState' objects}
    53000    7.342    0.000    7.757    0.000 <ipython-input-25-7e3e57ebef89>:91(updateIntermediaX)
    47295    0.064    0.000    0.064    0.000 {built-in method numpy.core.multiarray.array}
    20743    0.005    0.000    0.005    0.000 {built-in method builtins.len}
    20000    0.006    0.000    0.006    0.000 /opt/conda/lib/python3.5/site-packages/numpy/lib/stride_tricks.py:62(<g
enexpr>)
    18512    0.259    0.000    0.259    0.000 {method 'reduce' of 'numpy.ufunc' objects}
    17030    0.009    0.000    0.009    0.000 /opt/conda/lib/python3.5/site-packages/numpy/lib/stride_tricks.py:193(<
genexpr>)
    15518    0.016    0.000    0.044    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/numeric.py:484(asanya
rray)
```

6.4 Broadcasting with Numpy

Third, we used numpy broadcasting to replace the nested for loops which reduced the total running time to 15.517 seconds.

```
p = pstats.Stats('work_numpy.prof')
p.sort_stats('ncalls').print_stats(10)
pass
```

```
         678718 function calls in 15.517 seconds

   Ordered by: call count
   List reduced from 74 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   110004    0.134    0.000    0.134    0.000 {method 'reshape' of 'numpy.ndarray' objects}
   107000    3.369    0.000    3.369    0.000 <ipython-input-12-228c30ad9db4>:1(NumPreInf)
    54500    0.302    0.000    0.302    0.000 {method 'rand' of 'mtrand.RandomState' objects}
    53000    6.512    0.000    6.875    0.000 <ipython-input-11-f495c0f15b83>:91(updateIntermediaX)
    47295    0.064    0.000    0.064    0.000 {built-in method numpy.core.multiarray.array}
    20743    0.005    0.000    0.005    0.000 {built-in method builtins.len}
    20000    0.005    0.000    0.005    0.000 /opt/conda/lib/python3.5/site-packages/numpy/lib/stride_tricks.py:62(<g
enexpr>)
    18512    0.253    0.000    0.253    0.000 {method 'reduce' of 'numpy.ufunc' objects}
    17030    0.009    0.000    0.009    0.000 /opt/conda/lib/python3.5/site-packages/numpy/lib/stride_tricks.py:193(<
genexpr>)
    15518    0.017    0.000    0.046    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/numeric.py:484(asanya
rray)
```

6.5 Merging Small Functions

All of these methods produced similar optimization results which greatly decreased the total running time. In order to further optimize the running time, we tried to reduce the total number of function calls by integrating small functions into the main function. After re-constructing our algorithm, we only left NumPreInf function in numpy version and one big function 'Gibbs Whole'. By doing so, we reduced the total numbers of function call from 678,718 times to 524,723 times, which further improved the total running time to 10.767 seconds.

```
p = pstats.Stats('Gibbs_GCHMM.prof')
p.sort_stats('time', 'cumulative').print_stats(10)
pass
```

```
         524722 function calls in 10.767 seconds

   Ordered by: internal time, cumulative time
   List reduced from 63 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    6.268    6.268   10.767   10.767 <ipython-input-14-54eb1aa86164>:1(Gibbs_GCHMM)
   107000    3.104    0.000    3.104    0.000 <ipython-input-14-54eb1aa86164>:83(NumPreInf)
    13512    0.515    0.000    0.515    0.000 {method 'reduce' of 'numpy.ufunc' objects}
    54500    0.259    0.000    0.259    0.000 {method 'rand' of 'mtrand.RandomState' objects}
      500    0.116    0.000    0.116    0.000 {method 'repeat' of 'numpy.ndarray' objects}
   110004    0.095    0.000    0.095    0.000 {method 'reshape' of 'numpy.ndarray' objects}
     3006    0.045    0.000    0.363    0.000 /opt/conda/lib/python3.5/site-packages/scipy/stats/_distn_infrastructur
e.py:909(rvs)
    35295    0.035    0.000    0.035    0.000 {built-in method numpy.core.multiarray.array}
     4000    0.033    0.000    0.044    0.000 /opt/conda/lib/python3.5/site-packages/numpy/lib/stride_tricks.py:57(_b
roadcast_to)
     3006    0.030    0.000    0.181    0.000 /opt/conda/lib/python3.5/site-packages/scipy/stats/_distn_infrastructur
e.py:789(_argcheck_rvs)
```

6.6 Comparison Table for Gibbs Sampling

Below is a comparison table for the different algorithms mentioned above.

Table 1: Gibbs Sampler Optimization

	Gibbs_Navie	Gibbs_Cython	Gibbs_JIT	Gibbs_NP	Gibbs_GCHMM
Total Time(s)	2113.522	16.545	16.619	15.517	10.767

6.7 Profilling and Optimization for Generalized Baum-Welch Algorithm

For the generalized Baum-Welch algorithm (i.e., approximate EM belief propagation, AEMBP), we directly wrote whole algorithm in numpy with many broadcastings, since from the experience of the Gibbs Sampler, we found the numpy broadcasting was outperform than JIT and Cython. In addition, the generalized Baum-Welch algorithm is too complex to write in for loops first, which may take incredible long time to run the naive version. Based on the pre-optimized the algorithm, we did optimization by removing many redundant calculations.

```
p = pstats.Stats('AEMBP.prof')
p.print_stats(10)
pass
```

6.7.1 Pre-computing - Reduce Redundant Iterations

After profiling the numpy version code, we found the np.sum in fGX, SumProd, and sp functions cost lots of time. After exploring our code, we found that we lost much efficiency on calculating np.sum of different length lists by 1 axis in nested for loops. Thanks to the special property of our network which the maximum degree of node is 11, it was feasible to compute all np.sum of different length lists by 1 axis, which depended on degree of node outside the function and passed it as a parameter into our function. By doing so, we reduced the total function calls from 24,804,611 times to 24,109,445 times for 5 iterations, which shortened the total running time from 66.411 seconds to 56.983 seconds. If our algorithm is running a large data set which requires a large numbers of iteration to converge, our optimization will be very promising and saving lots of running time.

```
p = pstats.Stats('AEMBP_GCHMM.prof')
p.sort_stats('time','cumulative').print_stats(15)
pass
```

6.8 Comparison Table for Generalized Baum-Welch

Below is a comparison table for the different algorithms mentioned above.

```
         24804611 function calls in 66.411 seconds

   Ordered by: internal time, cumulative time
   List reduced from 67 to 15 due to restriction <15>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   576640   15.691    0.000   42.730    0.000 <ipython-input-33-a74d365c9cea>:1(sp)
  2832449    9.775    0.000    9.775    0.000 {method 'reduce' of 'numpy.ufunc' objects}
     2140    5.726    0.003   14.378    0.007 <ipython-input-32-959b20fc46eb>:1(SumProd)
        1    5.718    5.718   66.411   66.411 <ipython-input-29-132daf26b55b>:7(AEMBP)
  1500600    5.081    0.000   13.703    0.000 /opt/conda/lib/python3.5/site-packages/numpy/matlib.py:310(repmat)
  2079994    4.722    0.000   13.463    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/fromnumeric.py:1743(s
um)
  3001205    4.390    0.000    4.390    0.000 {method 'repeat' of 'numpy.ndarray' objects}
   375150    3.656    0.000    3.656    0.000 <ipython-input-29-132daf26b55b>:62(<lambda>)
  4503420    2.329    0.000    2.329    0.000 {method 'reshape' of 'numpy.ndarray' objects}
   375150    2.298    0.000    2.298    0.000 <ipython-input-29-132daf26b55b>:61(<lambda>)
   752440    1.685    0.000    5.064    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/fromnumeric.py:2433(p
rod)
  1510247    1.475    0.000    1.930    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/numeric.py:484(asanya
rray)
  2079994    1.131    0.000    8.018    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/_methods.py:31(_sum)
  2082240    0.723    0.000    0.723    0.000 {built-in method builtins.isinstance}
   578780    0.699    0.000    0.699    0.000 {method 'copy' of 'numpy.ndarray' objects}
```

```
         24109445 function calls in 56.983 seconds

   Ordered by: internal time, cumulative time
   List reduced from 66 to 15 due to restriction <15>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   576640   15.419    0.000   33.727    0.000 <ipython-input-6-347b45838ca2>:1(sp)
  2658791    7.791    0.000    7.791    0.000 {method 'reduce' of 'numpy.ufunc' objects}
        1    6.108    6.108   56.982   56.982 <ipython-input-2-ac81889efacc>:7(AEMBP_revise)
  1500600    4.763    0.000   12.396    0.000 /opt/conda/lib/python3.5/site-packages/numpy/matlib.py:310(repmat)
     2140    4.501    0.002   10.989    0.005 <ipython-input-5-49c0020f5080>:1(SumProd)
  1906336    3.878    0.000   10.100    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/fromnumeric.py:1743(s
um)
  3001205    3.770    0.000    3.770    0.000 {method 'repeat' of 'numpy.ndarray' objects}
  4503420    2.152    0.000    2.152    0.000 {method 'reshape' of 'numpy.ndarray' objects}
   752440    1.668    0.000    5.360    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/fromnumeric.py:2433(p
rod)
  1510247    1.307    0.000    1.735    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/numeric.py:484(asanya
rray)
   375150    1.289    0.000    1.289    0.000 <ipython-input-2-ac81889efacc>:61(<lambda>)
  1906336    1.005    0.000    5.535    0.000 /opt/conda/lib/python3.5/site-packages/numpy/core/_methods.py:31(_sum)
   375150    0.800    0.000    0.800    0.000 <ipython-input-2-ac81889efacc>:60(<lambda>)
  1908582    0.687    0.000    0.687    0.000 {built-in method builtins.isinstance}
   578780    0.660    0.000    0.660    0.000 {method 'copy' of 'numpy.ndarray' objects}
```

Table 2: Generalized Baum Welch Optimization

	AEMBP	AEMBP_GCHMM
Total Time(s)	66.411	56.983

7 Code Testing

7.1 Test on Prior Choices

```
assert(ap < bp & aa < ba & ab < bb & ar < br & a1 > b1 & a0 < b0)
```

Remember our priors elicitation:

Priors on emission parameters: $\theta_0 \sim \text{Beta}(a_0, b_0), \theta_1 \sim \text{Beta}(a_1, b_1)$

Priors on transition parameters: $\alpha \sim \text{Beta}(a_\alpha, b_\alpha), \beta \sim \text{Beta}(a_\beta, b_\beta), \gamma \sim \text{Beta}(a_\gamma, b_\gamma), \xi \sim \text{Beta}(a_\xi, b_\xi)$

We set these priors to be: $(a_\alpha, b_\alpha) = (a_\beta, b_\beta) = (a_\gamma, b_\gamma) = (a_\xi, b_\xi) = (a_0, b_0) = (2, 5), (a_1, b_1) = (5, 2)$ when actually implementing the algorithm.

γ stands for recovery probability if infectious at pervious time α stands for probability being infected from some one outside networks β stands for probability being infected from some one inside networks ξ is the prior probability for a Bernoulli distribution of $X_{n,1}$, i.e., the probability that a student get infected on the first day. These are very small probability values, and therefore should have a left-skewed beta distribution. We used 'assert' function to make sure these priors are on a reasonable scale.

θ_1 is the probability of presenting a symptom under the condition that the student gets infected. It should be larger than θ_0, the probability of presenting a symptom under the condition that the student did NOT get infected. The assertion is important as we tested on boundary conditions when θ_1 and θ_0 has symmetric beta prior, i.e., $(a_0, b_0) = (a_1, b_1)$, the resulting heat map shows label switching problem. The algorithm cannot determine which one stands for the probability of getting infected.

```
assert(ap < bp & aa < ba & ab < bb & ar < br & a1 > b1 & a0 < b0)
```

7.2 Unit test

As we discussed in last section, the optimized code only involves the large function of 'Gibbs', and the small function 'NumPreInf', the latter is tested by an external file for unit testing. The correctness is ensured by showing the function results in an unique value (of a 1-d numpy array).

```
import unittest
import numpy as np

def NumPreInf(Xt, Gt):
    return ((Gt + Gt.T) > 0) @ Xt

class TestUM(unittest.TestCase):

    def setUp(self):
        pass

    def test_matrix(self):
        self.assertEqual( NumPreInf(np.array([1,0]),
                                    np.array([[0, 1],
                                              [0, 0]])).all(),
                          np.array([0,1]).all())

if __name__ == '__main__':
    unittest.main()
```

```
wl-10-190-96-203:Desktop Emily$ python test.py
.
```

```
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

8 Conclusion

In this project, we implemented the two algorithms Gibbs sampling and Generalized Baum-Welch using approximate EM belief propagation to model the spread of infection within small community. In the actual dynamic social network data, maximum degree of $G_{84\times84\times107}$ is bounded by constant $M = 11$, meaning that the number of connections per student per day cannot exceed 11. Several methods have been used to make the GCHMM implementation code faster, such as removing redundant calculations by converting 0 to 11 into binary sequence outside the function. However, the computation complexity is bounded by the maximum number of node in the graph. If the maximum number of node is allowed to be even larger, the computation complexity will grow drastically. The boundary condition is a limitation to both the model setup and computation efficiency. In spite of that, Graph-Coupled HMMs is still a popular choice and has already been wrapped up in mobile application to model flu diffusions.

Acknowledgments

The author would like to thank **Kai Fan** for helpful discussions.

References

[1] Wen Dong, Alex Pentland, and Katherine A Heller. Graph-coupled hmms for modeling the spread of infection. *arXiv preprint arXiv:1210.4864*, 2012.

[2] Kai Fan, Allison E Aiello, and Katherine A Heller. Bayesian models for heterogeneous personalized health data. *arXiv preprint arXiv:1509.00110*, 2015.

[3] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.