STA663 Final Project
Implementation of Infinite Feature Models and the Indian Buffet Process

Yizheng Wang & Muyao Sun

May 1, 2017

Abstract

The Indian buffet process, a non-parametric Bayesian approach, defines a probability distribution over equivalence classes of sparse and unbounded number of features binary matrices. It is commonly applied as a prior in probabilistic models to represent objects with infinite number of features. In this project, we implement the linear-Gaussian binary latent feature model using IBP as a prior to detect features and reconstruct images. We also derive a Markov Chain Monte Carlo sampling algorithm for Bayesian inference, and provide experiments on both synthetic and real-world data. The experimental results show that the linear-Gaussian binary latent feature model performs well on accurately predicting the latent features and detecting them from images with noise.

Keywords: Bayesian Nonparametrics, Dirichlet Process, Linear Gaussian, Latent Variable Models, Gibbs Sampling

1 Introduction

A common goal of unsupervised learning is to discover the latent structure responsible for generating observed properties of a set of objects. The most non-parametric Bayesian statistics approaches examining latent variables are based on an assumption that the amount of latent variables is potentially unbounded and the "observed objects only manifest a sparse subset of those classes or features" (Griffiths and Ghahramani, 2010). The Chinese restaurant process (CRP) is a commonly used discrete-time stochastic process, analogous to seating customers at infinite number of tables in a Chinese restaurant. In CRP, each object in the data set is associated with a latent variable that comes from a single class or be produced by a single parameter. However, in the Indian buffet process (IBP) introduced by Thomas L. Griffiths and Zoubin Ghahramani in "Infinite Latent Feature Models and the Indian Buffet Process," each object is allowed to have infinite number of latent classes.

1.1 Indian Buffet Process

The Indian buffet process is a metaphor of Indian restaurant lunchtime buffet with infinite number of dishes. The authors define a distribution over infinite binary matrices by specifying a procedure by which customers (objects) choose dishes (features). In the Indian buffet process, N customers enter a Indian restaurant and each of them starts at the left of the buffet to take a serving from

Poisson(α) number of dishes until his plate is full. For the ith customer, the customer serves himself with probability $\frac{m_k}{i}$ where m_k representing the number of previous customers having chose the dish and with probability Poisson($\frac{\alpha}{i}$) choosing new dishes. Using a binary matrix Z with N rows and infinite number of columns to represent customers ' choices on dishes, the probability of Z follows IBP, equivalently Z $\sim$ IBP(α):

$$P(Z) = \frac{\alpha^{K+}}{\prod_{i=1}^{N} K_1^{(i)}!} exp(-\alpha H_N) \prod_{k=1}^{K+} \frac{(N-m_k)!(m_k-1)!}{N!}$$

where $m_K = \sum_{i=1}^{N} z_{ik}$ denotes the number of objects possessing feature k, $K+$ denotes the number of features for which $m_k > 0$, $K_1^{(i)}$ denotes the number of new dishes chosen by ith customer and $H_N = \sum_{j=1}^{N} \frac{1}{j}$ denotes the Nth harmonic number.

The next step before defining a Gibbs sampler is to formulate the conditional distribution on feature assignments. Integrating π_k to compute the conditional distribution for any z_{ik}:

$$P(z_{ik} = 1 | z_{-i,k}) = \frac{m_{-i,k} + \frac{\alpha}{K}}{N + \frac{\alpha}{K}}$$

where $-i$ indicates all the objects except the ith object.

Both the probability distribution for Z and conditional distribution for any z_{ik} are in the finite case. By taking the limit of both equations as $K \to \infty$, the probability distribution for Z becomes

$$P(Z) = \frac{\alpha^{K+}}{\prod_{h=1}^{2^N-1} K_h!} exp(-\alpha H_N) \prod_{k=1}^{K+} \frac{(N-m_k)!(m_k-1)!}{N!}$$

where K_h denotes the number of features possessing the history h, and the conditional distribution for z_{ik} becomes

$$P(z_{ik} = 1 | z_{-i,k}) = \frac{m_{-i,k}}{N}$$

as K goes to infinite, $\frac{\alpha}{K}$ goes to zero.

1.2 Application

The Indian Buffet Process is closely related to the Dirichlet processes therefore commonly applied to non-parametric Bayesian methods as a prior to capture researchers' beliefs. IBP can also be applied to a wide array of fields, such as image reconstruction, text modeling and population clustering. In the Implementation section, we will use IBP in the linear-Gaussian binary latent feature model to reveal its latent structure and further apply it to recognize and reconstruct the Duke University logo image. In order to prove its reconstructing power, we use clear(without noise) and unclear(with noise) images as the bases.

2 Implementation

The Indian buffet process (IBP) can be used as a prior in models for unsupervised learning. An example is given by Griffiths and Ghahramani to implement IBP in a linear-Gaussian latent feature model.

2.1 Linear-Gaussian Latent Feature Model

In the linear-Gaussian latent feature model, a binary $N \times K$ feature matrix Z generated from the IBP will be used to represent which features are possessed by each object in X. In the finite case, the D-dimensional vector of an object x_i is generated from a Gaussian distribution:

$$x_i|z_i, A, \sigma_X^2 \sim N(z_i A, \sigma_X^2 I)$$

where z_i is a K-dimensional binary vector, A is a $K \times D$ matrix of weights also generated from a Gaussian distribution:

$$A|\sigma_A^2 \sim N(0, \sigma_A^2 I)$$

and σ_X^2 is a covariance matrix generating noise to the image. By integrating out A, the likelihood can be derived:

$$\begin{aligned} p(X|Z, \sigma_X, \sigma_A) = & \frac{1}{(2\pi)^{ND/2}]\sigma_X^{(N-K)D}\sigma_A^{KD}|Z^T Z + \frac{\sigma_X^2}{\sigma_A^2} I|^{D/2}} \\ & exp(-\frac{1}{2\sigma_X^2} tr(X^T(I - Z(Z^T Z + \frac{\sigma_X^2}{\sigma_A^2} I)^{-1} Z^T)X)). \end{aligned} \tag{1}$$

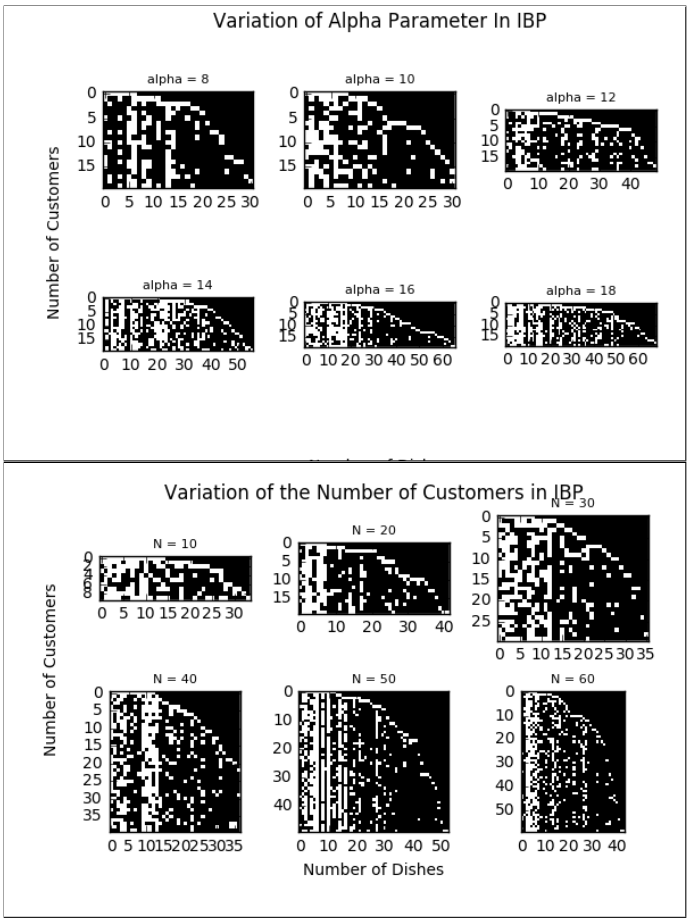

Figure 1: Binary latent feature matrices generated by IBP with different α and N

2.2 Gibbs Sampler for the Full Conditionals

2.2.1 Hyperparameters

- Z: binary feature matrix
- α: hyperparameter of Poission
- σ_X: covariance matrix of X
- σ_A: covariance matrix of A

2.2.2 Gibbs Sampler Algorithm

The full conditional of each z_{ik} is defined by

$$p(z_{ik}|X, Z_{-ik}, \sigma_X, \sigma_A) \propto p(X|Z, \sigma_X, \sigma_A)p(z_{ik}|z_{-ik}) \tag{2}$$

where z_{-ik} represents the set of assignments of all the objects expect object i for the feature k. The first distribution on the right side is given in Eq. 1 and the second on is given in Section 1.1. The algorithm is outlined as following:

1. Initialize the values of α, σ_X and σ_A
2. Generate z_{ik} from its full conditional (Eq. 2)
3. Sample σ_X by Metropolis with acceptance rate $r < min(1, \frac{p(\sigma_X^{new}|Z,X,\sigma_A)}{p(\sigma_X|Z,X,\sigma_X)})$ where $\sigma_X^{new} = \sigma_X + noise$
4. Sample σ_A by Metropolis with acceptance rate $r < min(1, \frac{p(\sigma_A^{new}|Z,X,\sigma_X)}{p(\sigma_A|Z,X,\sigma_X)})$ where $\sigma_A^{new} = \sigma_A + noise$
5. Generate α from its posterior distribution $\alpha|Z, K+, H_N \sim Ga(a+K+, b+H_N)$ where $H_N = \sum_{i=1}^{N} \frac{1}{i}$
6. Repeat step 2-5 until converge (practically, step 2-5 will be run for a specific number n iterations)

2.2.3 Posterior Distribution of Weights (A)

After obtaining the converged values of Z,σ_X and σ_A, the posterior mean of A can be computed by

$$\mu_A|X, Z, \sigma_X, \sigma_A = (Z^T Z + \frac{\sigma_X^2}{\sigma_A^2} I)^{-1} Z^T X$$

where A is a $D \times K+$ matrix of latent features. The converged values for weight matrix A can be used to recreate images.

3 Unit Test

In order to guarantee the validity of our algorithm, several tests are conducted. As following:

- Test if N, K and D are all non-negative integers
- Test if α, σ_X and σ_A are all positive floats

- Test if **IBP** function returns an error if input non-positive α or non-integer N
- Test if **calcM** function returns an error if input non-positive σ_X or σ_A, or non-integer N or K
- Test if the result from regular **calcM** is same with result from optimized **calcM**
- Test if **calcM2** function returns an error if input non-positive σ_X or σ_A, or non-integer N or K
- Test if **log_likelihood** function returns an error if input non-positive σ_X or σ_A, or non-integer N or K or D
- Test if **log_likelihood** function returns positive likelihood

For example:

```
In [12]: testm=calcM(Z,testk,sX,sA)

In [13]: testm2=calcM_cython(Z,testk,sX,sA)

In [14]: np.testing.assert_almost_equal(testm,testm2)
         print("pass")

         pass
```

Figure 2: Example for testing code

The more details are in **Test.py** file.

4 Results and Comparison

In this section, we test the performance of linear-Gaussian binary latent feature model using a IBP prior on predicting latent features and detecting the presence of latent features in each object. We apply the linear-Gaussian binary latent feature model to both synthetic and real dataset and attempt to recreate the images based upon converged values of weight matrix from MCMC.

4.1 Simulated Data

The linear-Gaussian binary latent feature model is applied to our simulated data X:

- objects (N): 100
- dimension (D): $36 = 6 \times 6$
- features (K): 4
- Poisson hyperparameter $\alpha \sim$ Ga(1,1)
- covariance of X: 1.7
- covariance of A: 1.2
- noise $\sim$ Unif$(-\frac{1}{20}, \frac{1}{20})$
- MCMC iteration: 1000

The non-zero elements of weights matrix A are set to 1 and the elements of binary feature matrix X are sampled from a distribution with probability 0.5. All the initial settings are similar to Griffiths and Ghahramani (Section 5.3).

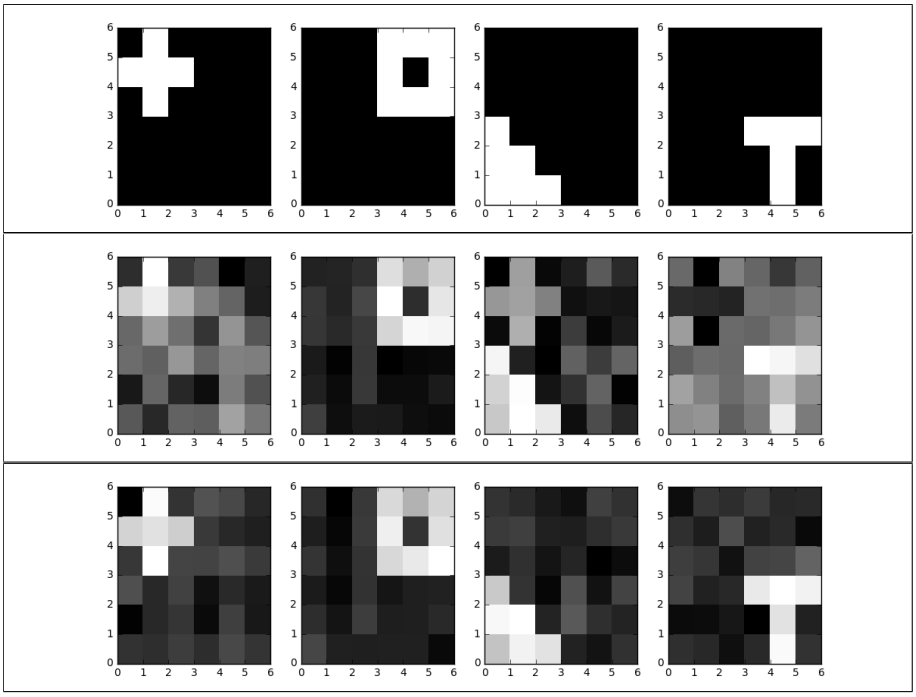

Figure 3: Simulated data and results for the demonstration of the linear-Gaussian binary latent feature model. Top: Four Basis images of simulated data; Middle: Reconstruction of the images based upon the posterior mean of weights before converge (180th iteration); Bottom: Reconstruction of the images based upon the posterior mean of weights after converge (999th iteration)

The simulated data and results are shown in Fig. 2. The intensity of color from white to black reflects the presence to absence of feature. In order to demonstrate the ability of linear-Gaussian binary latent feature model to detect latent features and reconstruct images, we place the basis images on the top of Fig. 2 and compare the reconstructed images using posterior mean of weight matrix (A) before converge and after converge. Apparently, the reconstructed images using 999th iteration A values are more vivid and clear.

The trace plots for MCMC simulation K, α, σ_X and σ_A are shown in Fig. 3. Griffiths and Ghahramani state that their values of parameters stabilize after approximately 100 iteration, but according to our plots, the algorithm reaches stable values for all of the parameters after approximately 200 iterations. The value of newly chosen features (K) converges to the interval [4,8], the value of Poisson parameter α converges to the interval [0.5,2.0], the value of covariance of simulated data σ_X converges to 0.5 and the value of covariance of weight matrix σ_A converges to 0.4.

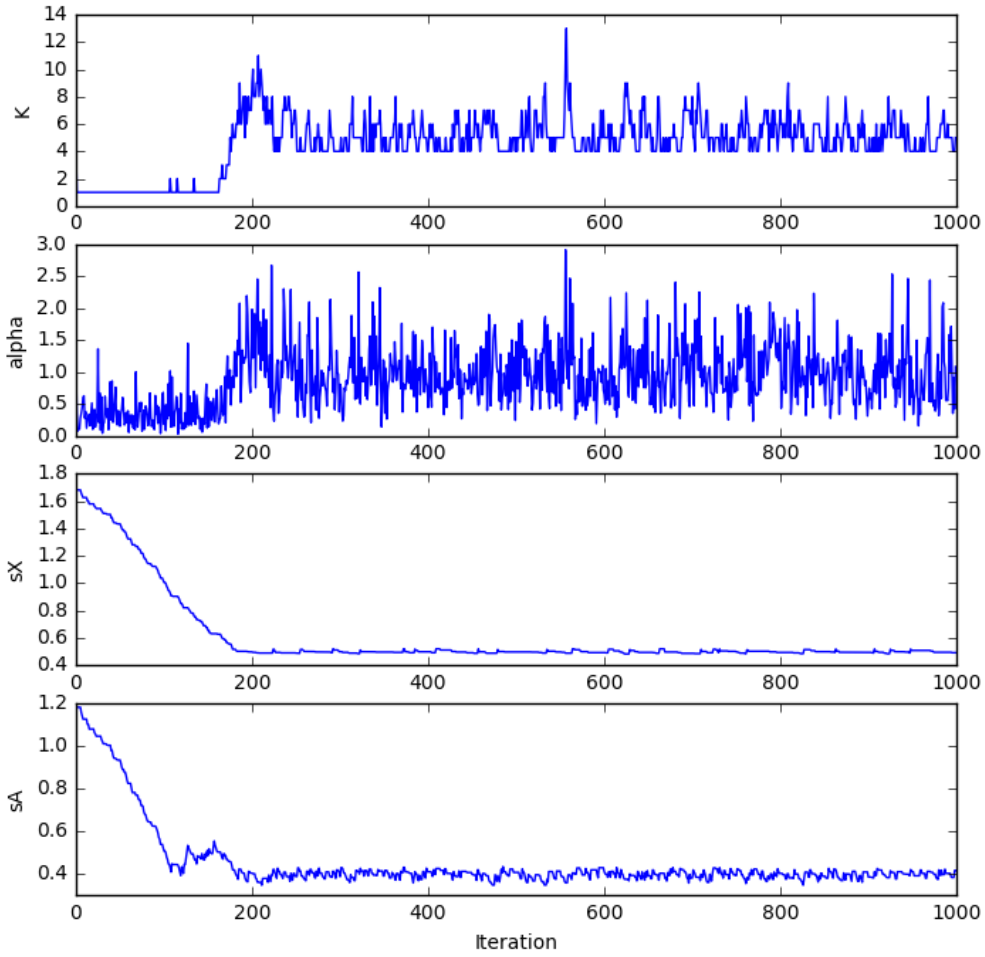

Figure 4: Trace plots for the dimensionality of the representation (K) and hyperparameters (α, σ_X and σ_A) over 1000 iterations

4.2 Real Data

We also apply the linear-Gaussian binary latent feature model to our real data, the Duke logo. We keep the same initial values for Gibbs sampler (α, σ_X and σ_A) and change the following parameters:

- objects (N): 80
- dimension (D): $625 = 25 \times 25$
- features (K): 1

The Gibbs sampler still run for 1000 iterations. The model successfully finds latent features to reconstruct the image is shown in Fig. 4.

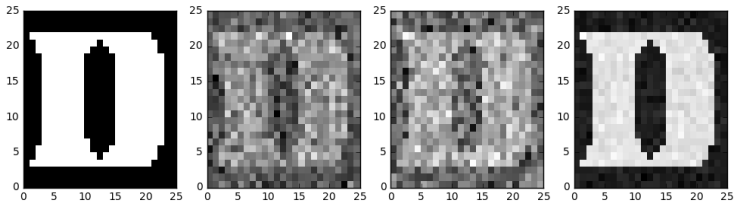

Figure 5: Real data and results for the demonstration of the linear-Gaussian binary latent feature model.1st: The basis image of Duke logo; 2nd, 3rd: Two simulated images using the latent feature; 4th: Reconstruction of Duke logo

Since we cannot directly add noise to our real data, we employ a noise basis image into our linear-Gaussian binary latent feature model. In this case the feature varible K is equal to 2. As we expected, the noise basis image disturbs the accurateness and clarity of recreated images (the second row of Fig. 5). But surprisingly, the convergences of four hyperparameters in the MCMC do not apparently change, which are still achieved at approximately 200 iterations.

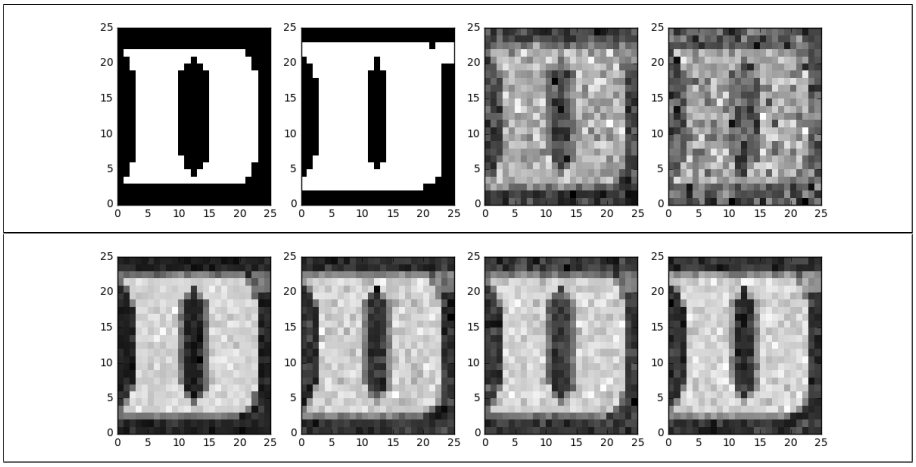

Figure 6: Real data with noise and results for the demonstration of the linear-Gaussian binary latent feature model. Top (1st, 2nd): The basis image of Duke logo and noise basis image; Top (3rd, 4th): Two simulated images using both latent feature and noise feature; Bottom: Reconstruction of the Duke logo

5 Code Optimization

It is easy to write code in python because it is dynamically typed, and we do not have to much worry about declaring variable types(e.g. integer, double,etc.). Also instead of compiled, it is interpreted. However, it does incur a major drawback: performance for some operations can be quite slow. Since not all operations can be vectorized, sometimes the **Numpy** array representation will not necessarily improve the calculation speed. Therefore, our only choice at this moment is to extension the code in C, but this is very complicated. It may require us to write much more additional code in order to communicate the python and C computation layers.

Cython is a great tool that allows us to easily connect python with the underlying C representations. In addition, **Numba** is a library that enables just-in-time(JIT) compiling of Python code. In order to do this, it uses the LLVM tool chain. As the code running, LLVM takes an intermediate representation of my code and compile it to highly optimized machine code, briefly. Hence, in this project, both these two techniques are used to speed up the python code. We write the functions (**calcM**, **calcM2** and **log_likelihood**) will be called in **main_solver** into three formats: regular (use vectorizing, numpy and avoid using loops as many as possible), improve (use JIT complier) and improve_cython (use cythonized code).

We first profile the **main_solver** code by **line_profiler** and check the bottleneck where slows the computational speed. The lines calling **log_likelihood** function get stuck and spend a lot of computational time. Since the profiling results are too long, we will include them in the github repository. We also compare the profiling results of three improved **main_solver** functions for one iteration:

```
Sun Apr 30 17:02:48 2017    work.prof

         156271 function calls in 0.498 seconds

   Ordered by: internal time, cumulative time
   List reduced from 87 to 3 due to restriction <'main'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.022    0.022    0.173    0.173 <ipython-input-410-c0fe87c3fd08>:1(mainsolver_improve)
        1    0.017    0.017    0.168    0.168 <ipython-input-412-ea64e498a5ee>:2(mainsolver)
        1    0.017    0.017    0.156    0.156 <ipython-input-411-bf358e053df3>:1(mainsolver_improve_cython)
```

Figure 7: Profiling results of three main_solver functions

Surprisingly, only cythonized code improves computational speed. In order to remain validity and generality, we use **timeit** module to find the best loop among 200 iterations. Since the computing time is sensitive to different seeds, we set the seed at **np.random.seed(123)**. The following table shows the details.

Category	Running Time
main_solver (regular)	best of 1: 55.4s per loop
main_solver improvement with JIT	best of 1: 1min 3s per loop
main_solver improvement with Cython	best of 1: 49.55s per loop

We vectorized the regular **main_solver** function and deployed those matrix calculation functions from Numpy library as much as possible. As mentioned previously, not all operations can be vectorized, and sometimes the Numpy array representation will not necessarily improve the calculation speed. This may be the reason why Cython significantly speed up the python code.

6 Conclusion

The Indian Buffet Process (IBP) is a non-parametric Bayesian approach in which objects are allowed to unbounded number of features. In this project, we implement the linear-Gaussian binary latent feature model using IBP as a prior to detect features and reconstruct images. Both for our simulated dataset and real dataset, the linear-Gaussian binary latent feature model performs well on correctly predicting the latent features and detecting them from images with noise. In order to

achieve faster computing speed, we apply Cython and just-in-time (JIT) to our code. It turn out that Cython indeed decrease the average computational time. In further research, we consider to derive a stick-breaking representation for IBP and then replace Gibbs sampler with slice sampler since Gibbs sampler requires conjugacy.

References

[1] Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation." *Journal of machine Learning research*. 3.Jan (2003): 993-1022.

[2] Blei, David M., Thomas L. Griffiths, Michael I. Jordan, and Joshua B. Tenenbaum. "Hierarchical topic models and the nested chinese restaurant process." *Advances in neural information processing systems*. 16 (2004): 17.

[3] Doshi-Velez, Finale. "The Indian buffet process: Scalable inference and extensions." Master's thesis, The University of Cambridge (2009).

[4] Griffiths, Thomas L., and Zoubin Ghahramani. "Infinite latent feature models and the Indian buffet process." *NIPS*. Vol. 18. 2005.

[5] Griffiths, Thomas L., and Zoubin Ghahramani. "Infinite latent feature models and the Indian buffet process." *Technical Report 2005-001*. Gatsby Computational Neuroscience Unit, 2005.

[6] Griffiths, Thomas L., and Zoubin Ghahramani. "The indian buffet process: An introduction and review." *Journal of Machine Learning Research*. 12.Apr (2011): 1185-1224.

[7] Teh, Yee Whye, Dilan Grür, and Zoubin Ghahramani. "Stick-breaking construction for the Indian buffet process." *Artificial Intelligence and Statistics*. 2007.

[8] Thibaux, Romain, and Michael I. Jordan. "Hierarchical Beta Processes and the Indian Buffet Process." *AISTATS*. Vol. 2. 2007.

[9] Williamson, Sinead, et al. "The IBP compound Dirichlet process and its application to focused topic modeling." *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010.

[10] Yildirim, Ilker. "Bayesian Statistics: Indian Buffet Process." (2012).

[11] Yildirim, Ilker. *Ilker Yildirim's Homepage*. N.p., 2012. Web. 30 Apr. 2017.