STA663 Statistical Computation Final Project

Implementation of the Indian Buffet Process (IBP)

Christine P. Chai

Executive Summary

The Indian Buffet Process is a fundamental algorithm in Bayesian nonparametrics, and it models an infinite number of features to reveal the latent structure of the data. In this report, I implemented the algorithm in the paper "Infinite Latent Feature Models and the Indian Buffet Process" [9], performed code testing to ensure correctness, optimized the Python code, and compared my work with some other existing algorithms and/or packages. A simulated image dataset is used for generating the posterior sampling of the linear-Gaussian binary latent feature model with the Indian Buffet Process as the prior.

1 Introduction

The paper I selected is "Infinite Latent Feature Models and the Indian Buffet Process" (IBP) [9]. In unsupervised machine learning, discovering the hidden variables that generate the observations is important. Many statistical models [4, 8] can provide a latent structure in probabilitistic modeling, but the problem lies in the unknown dimensionality, i.e. how many classes/features to express the latent structure. Bayesian nonparametric methods are able to determine the number of latent features; the Chinese Restaurant Process (CRP) is an example [3], but it assigns each customer to a single component (table). The Indian Buffet Process allows each customer to be assigned to multiple components (dishes), and the process can serve as a prior for an potentially infinite array of objects. In my implementation, IBP is regarded as a prior for the linear-Gaussian binary latent feature model, and I referred to some Matlab code online [14, 13].

1.1 Algorithm Description

The Indian Buffet Process is a metaphor of Indian restaurants offering buffets with a close-to-infinite number of dishes, and the number of dishes sampled by a customer is a Poisson distribution. Assume N customers enter a restaurant one after another, and the first customer takes a Poisson(α) of dishes. Starting from the second person, the ith customer takes dish k with probability $\frac{m_k}{i}$, where m_k is the number of previous customers who have sampled that dish. In this way, the ith customer samples dishes proportional to their popularity. After reaching the end of all previously sampled dishes, the ith customer tries a Poisson($\frac{\alpha}{i}$) number of new dishes. Which customer sampled which dish is recorded in a binary array Z with N rows (representing customers) and infinitely many columns (representing dishes), where $z_{ik} = 1$ if customer i sampled the dish k. Note that the customers are not exchangeable, i.e. the dishes a customer samples is dependent on whether previous customers have sampled that

dish [9].

In terms of probability,

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

The subscript $_{-\mathbf{i},\mathbf{k}}$ indicates dish k and all customers except for the ith one. If the number of dishes is truncated to K, then the above equation becomes

$$P(z_{ik} = 1|\mathbf{z}_{-\mathbf{i},\mathbf{k}}) = \frac{m_{-i,k} + \frac{\alpha}{K}}{N + \frac{\alpha}{K}} \tag{2}$$

The N customers can be viewed as objects, and the K dishes can be regarded as features. Formally writing, $Z \sim \text{IBP}(\alpha)$, and

$$P(Z|\alpha) = \frac{\alpha^K}{\prod_{h=1}^{2^N-1} K_h!} \exp(-\alpha H_N) \prod_{k=1}^{K} \frac{(N - m_k)!(m_k - 1)!}{N!} \tag{3}$$

α is a variable influencing the number of features (denoted as D in later sections); m_k is the number of objects with feature k; K_h is the number of features with history h (whether the N objects possess this feature, $2^N - 1$ possibilities in total); H_N is the N^{th} harmonic number, i.e. $H_N = \sum_{k=1}^{N} \frac{1}{k}$.

1.2 Applications and Evaluation

Many applications and variations of the Indian Buffet Process exist. For example, the linear-Gaussian binary latent feature model I implemented [14] can be used to model "noisy" matrices and reveal the latent features. In this way, image data can be processed because we can interpret binary matrices with structured representations. For another example, Yildirim and Jacob [15] proposed an IBP-based Bayesian nonparametric approach to multisensory perception in an unsupervised manner. Furthermore, variations of the Indian Buffet Process include focused topic modeling [12], hierarchical beta processes [12], and variational inference [6].

The advantages and disadvantages of IBP are clear. Using a Poisson distribution, IBP is able to model an infinite sequence of integers, and the sequence can be truncated as needed. In the implementation of IBP, the advantages of Gibbs sampling and Metropolis-Hastings (MH) can be combined. Nevertheless, IBP relies on the assumption that datapoints (dishes) in a single string are exchangeable; each dish is assumed to be equally desired by customers. Another drawback is that the number of parameters increase as the dataset gets large, but Bayesian nonparameteric methods generally have this problem [14].

2 Code Structure and Simulated Data

To implement the linear-Gaussian binary latent feature model [9, 14] with IBP as the prior, a Gibbs sampler is used to generate the posterior samples, and the graphical model is shown in Figure 1. The IBP function is described in Section 1.1, and denoted as $Z \sim \text{IBP}(\alpha)$, where Z is the binary matrix and $\alpha \sim Ga(1, 1)$.

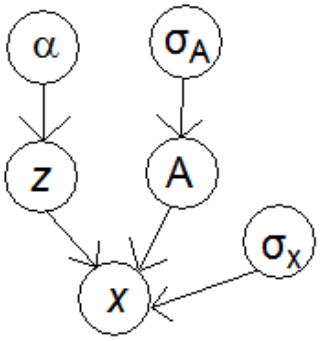

Figure 1: Graphical model for the linear-Gaussian binary latent feature model

2.1 Simulated Data for Likelihood

The likelihood involves simulated image data, and the variables are defined as follows:

- $N = 100$ is the number of images (customers or objects)
- $D = 6 \times 6 = 36$ is the length of vectors (dishes or features) for each image
- $K = 4$ is the number of basis images (latent or underlying variables)
- $\mathbf{X}$ represents the images generated by the K bases (each basis is present with probability 0.5), with white noises Normal$(0, \sigma_X^2 = 0.5^2)$ added

The likelihood function is

$$\mathbf{X}|(\mathbf{Z}, \mathbf{A}, \sigma_{\mathbf{X}}) \sim \text{Normal}(\mathbf{ZA}, \Sigma_X = \sigma_X^2 \mathbf{I}) \tag{4}$$

$$P(\mathbf{X}|\mathbf{Z}, \sigma_X, \sigma_A) = \frac{1}{(2\pi)^{ND/2} \sigma_X^{(N-K)D} \sigma_A^{KD} |\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}\mathbf{I}|^{D/2}} \exp\{-\frac{1}{2\sigma_X^2} \text{tr}(\mathbf{X}^T(\mathbf{I} - \mathbf{Z}(\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}\mathbf{I})^{-1}\mathbf{Z}^T)\mathbf{X})\} \tag{5}$$

Each object i has a D-dimensional vector of properties named x_i, where:

- $x_i \sim \text{Normal}(\mathbf{z_i A}, \Sigma_X = \sigma_X^2 \mathbf{I})$
- $\mathbf{z_i}$ is a K-dimensional binary vector (features)
- $\mathbf{A}$ is a $K \times D$ matrix of weights, with prior $\mathbf{A} \sim \text{Normal}(0, \sigma_A^2 \mathbf{I})$

The four basis images and an example of the simulated data are shown in Figure 2. Note that the likelihood involves close-to-zero probabilities, so the log likelihood is used in my code instead.

2.2 Gibbs Sampler for the Posterior Distribution

The full (posterior) conditional distribution is

$$P(z_{ik}|\mathbf{X}, \mathbf{Z_{-i,k}}, \sigma_X, \sigma_A) \propto P(\mathbf{X}|\mathbf{Z_{-i,k}}, \sigma_X, \sigma_A) P(z_{ik}|\mathbf{z_{-i,k}}) \tag{6}$$

When initializing the Gibbs sampler, set $\sigma_A = 1, \sigma_X = 1, \alpha \sim Ga(1, 1)$. Then the sampler does the following steps: (K in my code is denoted as K_+, to differentiate it from the true value.)

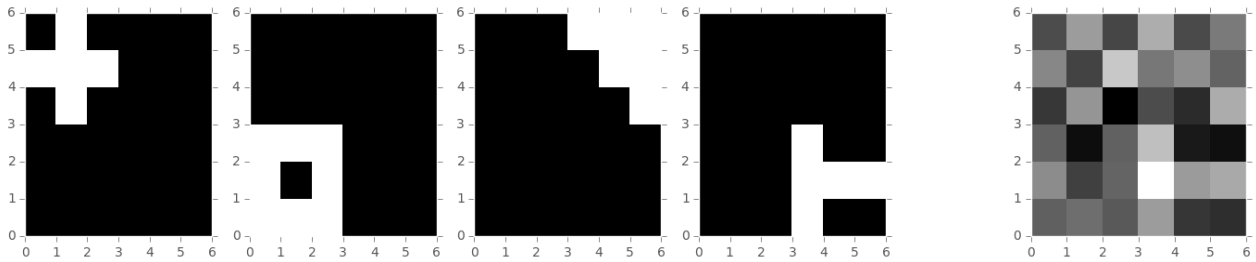

Figure 2: Simulated dataset: The four basis images (left) and an example image (right)

1. Generate $P(z_{ik}|\mathbf{X}, \mathbf{Z}_{-\mathbf{i},\mathbf{k}}, \sigma_X, \sigma_A)$ using the full conditional distribution
   (a) Remove singular features (at most one object has it); decrease K_+ by 1 for each feature removed
   (b) Determine each z_{ik} to be 0 or 1 by Metropolis
   (c) Add new features from Pois($\frac{\alpha}{i}$)
2. Sample $\sigma_X^* = \sigma_X + \epsilon$, where $\epsilon \sim \text{Unif}(-0.05, 0.05)$, and accept σ_X^* by Metropolis
3. Sample $\sigma_A^* = \sigma_A + \epsilon$, where $\epsilon \sim \text{Unif}(-0.05, 0.05)$, and accept σ_A^* by Metropolis
4. Generate $\alpha|Z \sim Ga(1 + K_+, 1 + \sum_{i=1}^{N} H_i)$, where K_+ is the number of features with $m_k > 0$

The Metropolis part for σ_A is demonstrated as follows (similar case for σ_X):

- Genenerate a candidate value $\sigma_A^* = \sigma_A + \epsilon$, with $\epsilon \sim \text{Unif}(-0.05, 0.05)$
- Generate a random number $r \sim \text{Unif}(0, 1)$
- Accept σ_A^* if $r < \min\{1, \frac{P(\sigma_A^*|\mathbf{Z}, \mathbf{X}, \sigma_X)}{P(\sigma_A|\mathbf{Z}, \mathbf{X}, \sigma_X)}\}$, where σ_A is the current value

The candidate value σ_A^* is always accepted when the likelihood ratio $\frac{P(\sigma_A^*|\mathbf{Z},\mathbf{X},\sigma_X)}{P(\sigma_A|\mathbf{Z},\mathbf{X},\sigma_X)}$ is larger than 1, i.e. $P(\sigma_A^*|\mathbf{Z}, \mathbf{X}, \sigma_X) > P(\sigma_A|\mathbf{Z}, \mathbf{X}, \sigma_X)$. Nevertheless, when the likelihood ratio is less than 1, there is still a non-zero probability to accept σ_A^*, so the sampler can "move forward". Note that in my code, the log likelihoods are used in the following way:

$$\min\{1, \frac{P(\sigma_A^*|\mathbf{Z}, \mathbf{X}, \sigma_X)}{P(\sigma_A|\mathbf{Z}, \mathbf{X}, \sigma_X)}\} = \exp(\min\{0, \log(P(\sigma_A^*|\mathbf{Z}, \mathbf{X}, \sigma_X)) - \log(P(\sigma_A|\mathbf{Z}, \mathbf{X}, \sigma_X))\}) \tag{7}$$

Finally, the posterior expectation of $\mathbf{A}$ is:

$$E(\mathbf{A}|\mathbf{X}, \mathbf{Z}) = (\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}\mathbf{I})^{-1}\mathbf{Z}^T\mathbf{X} \tag{8}$$

This is denoted as $\mathbf{A}_{\text{inf}}$, with size $D \times K_+$, and $\mathbf{A}_{\text{inf}}$ is the matrix of latent features (images) my code converges to.

3 Algorithm Output

My implementation of the linear-Gaussian binary latent feature model with the IBP prior generates the results in images and traceplots. The simulated dataset contains four latent features (see Figure 2), and my code reveals all of them in Figure 4. Note that my Gibbs sampler is sensitive to the random seed settings, and using another random seed in the code initialization reveals five (instead of four) latent features in Figure 5, three of which are the linear combinations of two latent features.

The traceplots in Figure 3 show my Gibbs sampler is converging: K_+ fluctuates between 5 and 8; the IBP parameter α is within (0.2,3.0); σ_X converges to the true value 0.5; σ_A oscillates around 0.4. A total of 1000 Gibbs sampling iterations were performed, but the values started to converge at the 100th iteration. Figure 6 shows the histogram of sampled K_+ values and the number of features for each object in the final iteration of $\mathbf{Z}$. The correct $K_+ = 4$ is not the sampled value with the highest probability, but the majority of objects only contain at most 4 features, so I conclude that $K_+ > 5$ in the middle of iterations are due to extreme values.

Moreover, the latent feature model is able to "reverse" the noisy images to the linear combination of latent features. Figure 7 is an example of the first four images in the simulated dataset: The top row contains the "reversed" images, and the latent features in each one can be clearly seen. The bottom row represents the original images, in which the latent features are obscured by the random noise.

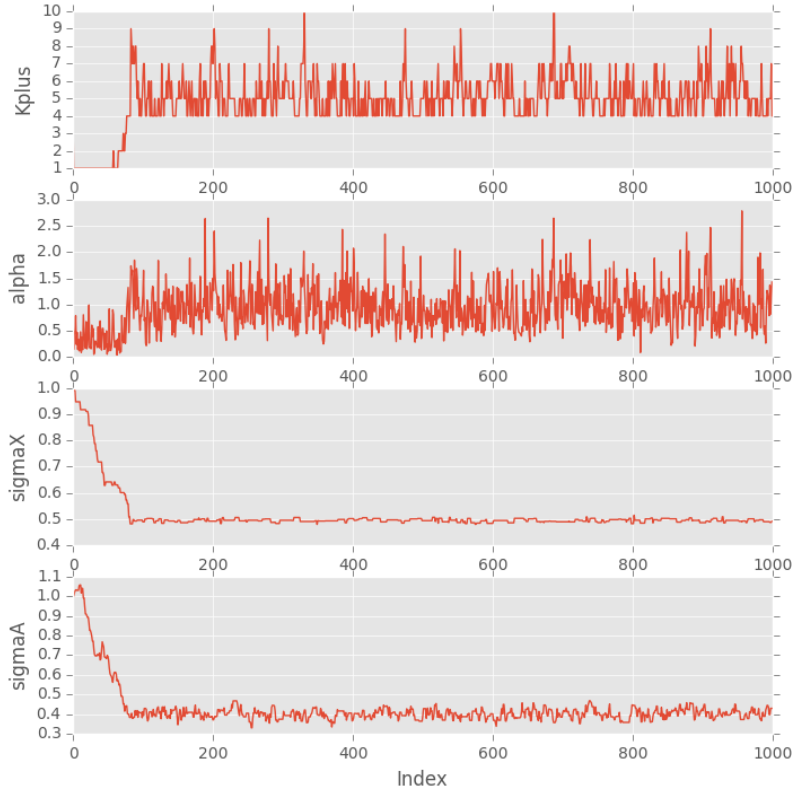

Figure 3: The traceplots for $K_+, \alpha, \sigma_X, \sigma_A$

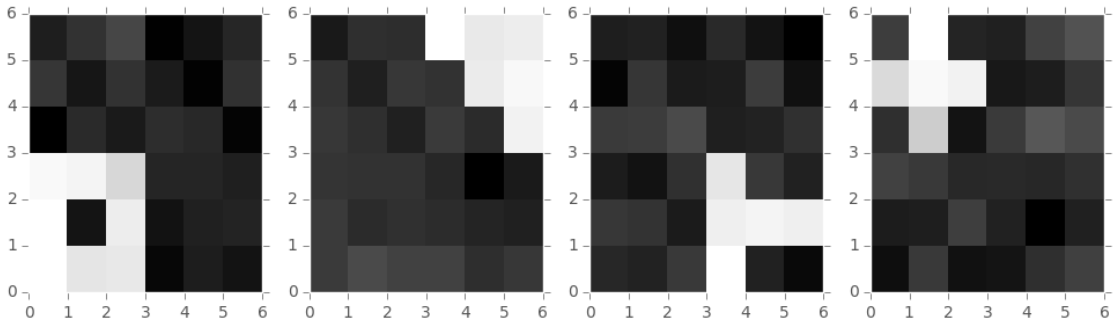

Figure 4: Simulated dataset: My results

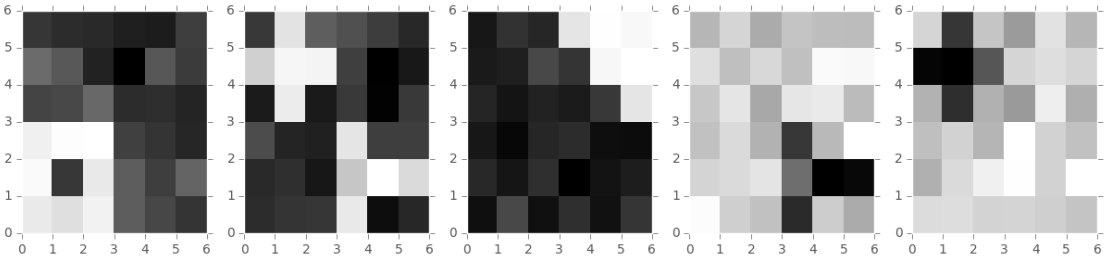

Figure 5: Simulated dataset: My results with another random seed

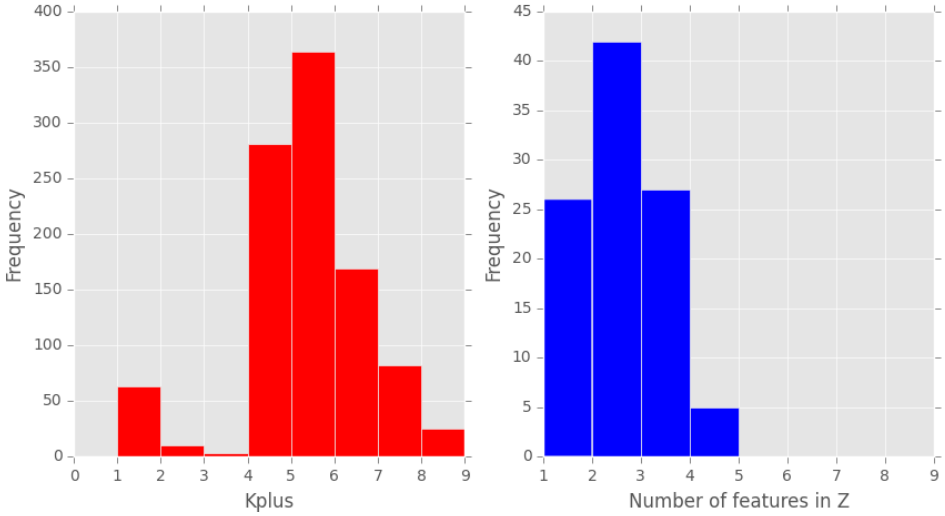

Figure 6: Histograms of K_+ (left) and the number of features in $\mathbf{Z}$ (right)

4 Code Testing

Code testing is performed by various methods. In the IBP prior, the `assert` command is used to verify $\frac{m_k}{i}$ to be a probability, i.e. between 0 and 1 – because the ith ($i > 2$) customer takes dish k with probability $\frac{m_k}{i}$ in the IBP algorithm. Before doing the Gibbs sampler, the parameters $K_+, N, D, \sigma_X, \sigma_A$ are `asserted` to be larger than 0. In many parts of my code, I used `np.dot` from `numpy` to do matrix multiplications even when the size of matrices is small, instead of multiplying each column/row one by one. In this way, the dimensions in matrix multiplications are assured to match each other.

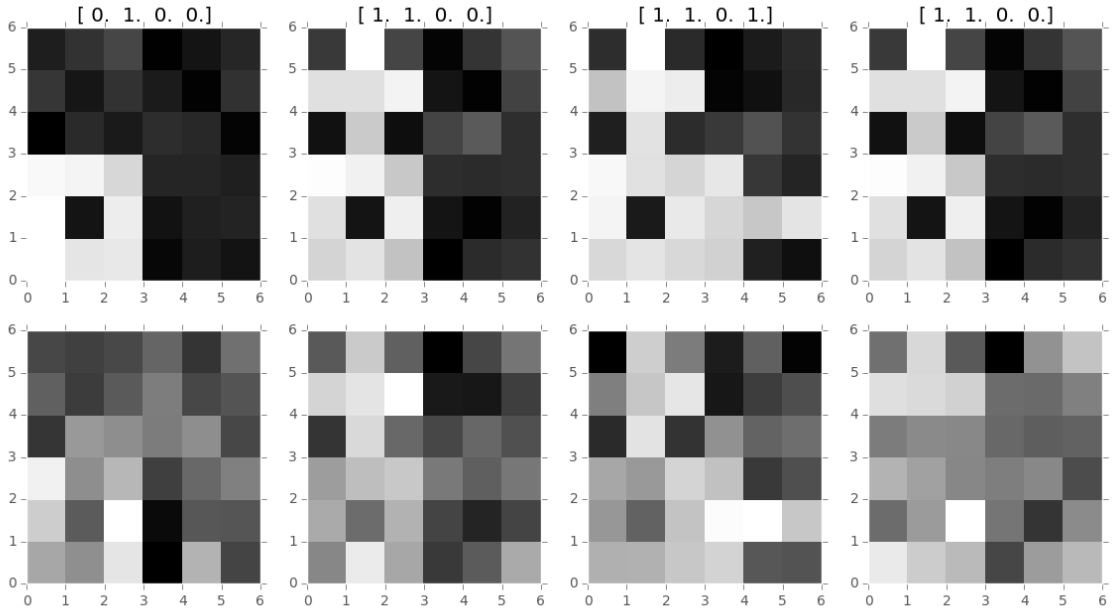

Figure 7: The latent features (top) corresponding to the simulated images (bottom)

For unit testing, the functions `calcM`, `calInverse_orig`, `calInverse` are tested by an external file. The first function `calcM` calculates $\mathbf{M} = (\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}\mathbf{I})^{-1}$, and the correctness is ensured by showing that $\mathbf{M}\mathbf{M}^{-1} = \mathbf{I}$. Note that `calcM` involves inversion, so it may be a bottleneck of my IBP code.

Therefore, I attempted to expedite the calculation of $\mathbf{M}$ by using Equations (51)-(54) in Griffiths' and Ghahramani's paper [10], when only one value of the whole matrix $\mathbf{Z}$ is changed. Theoretically, this method below allows us to efficiently compute $\mathbf{M}$ when the ith row of $\mathbf{Z}$ (denoted as $\mathbf{z_i}$) has changed:

$$\text{Define } \mathbf{M}_{-i} = (\sum_{j \neq i} \mathbf{z}_j^T \mathbf{z}_j + \frac{\sigma_X^2}{\sigma_A^2}\mathbf{I})^{-1} \tag{9}$$

$$\mathbf{M}_{-i} = (\mathbf{M}^{-1} - \mathbf{z}_i^T\mathbf{z}_i)^{-1} = \mathbf{M} - \frac{\mathbf{M}\mathbf{z}_i^T\mathbf{z}_i\mathbf{M}}{\mathbf{z}_i\mathbf{M}\mathbf{z}_i^T - 1} \tag{10}$$

$$\mathbf{M} = (\mathbf{M}_{-i}^{-1} - \mathbf{z}_i^T\mathbf{z}_i)^{-1} = \mathbf{M}_{-i} - \frac{\mathbf{M}_{-i}\mathbf{z}_i^T\mathbf{z}_i\mathbf{M}_{-i}}{\mathbf{z}_i\mathbf{M}_{-i}\mathbf{z}_i^T + 1} \tag{11}$$

The function `calInverse_orig` returns $\mathbf{M} = \mathbf{M}_{-i} - \frac{\mathbf{M}_{-i}\mathbf{z}_i^T\mathbf{z}_i\mathbf{M}_{-i}}{\mathbf{z}_i\mathbf{M}_{-i}\mathbf{z}_i^T + 1}$, and the function `calInverse` returns $\mathbf{M} = (\mathbf{M}_{-i}^{-1} - \mathbf{z}_i^T\mathbf{z}_i)^{-1}$ just for comparison results. One drawback of this method is that numerical errors can be accumulated, leading to wrong results. Therefore, a full rank update of $\mathbf{M}$ should be performed occasionally. In fact, unit testing shows that they give different results from the original `calcM` function, and using the `calInverse` functions makes K_+ stuck at 2, which is incorrect.

I also tested the log-likelihood for the Poisson distribution in my IBP code by showing the probability

mass function $f(x)$ below integrates to 1.

$$x \sim \text{Pois}(\lambda) : f(x) = \frac{\lambda^x}{x!} e^{-\lambda}, x = 0, 1, 2, 3, \cdots \tag{12}$$

$$\log f(x) = x \log \lambda - \lambda - \log(x!) \tag{13}$$

Therefore, $\exp \log f(x) = 1$ is approximated by $\sum_{x=0}^{20} f(x)$ in the unit testing code.

5 Profiling and Optimization

In this section, I performed profiling and optimization on the IBP linear-Gaussian model. The optimization strategies include removing redundant calculations, better use of matrix multiplication, cythonizing Python code, and using the jit (just-in-time) compiler. Since the MCMC chain posesses the Markov property; that is, the nth sample only depends on the $(n-1)$th sample, it is unreasonable to parallelize the IBP code.

5.1 Profiling

Profiling is done to identify the bottlenecks; the code structure can be visualized as a tree in Figure 8. In one Gibbs sampling iteration, generating $Z|\alpha$ and sampling σ_X, σ_A are performed once each. In generating $Z|\alpha$, sampling dishes from K_+ and sampling new dishes are done for each customer (image or object), so they are each performed $N = 100$ times. In sampling dishes from K_+, calculation refers to the process of sampling the posterior distribution of $Z|K_+$, and initialization is the part of removing features which are all zero. Both calculation and initialization are performed $N \times K_+ \approx 500$ times for each iteration.

Table 1 shows the profiling results for my initial code. The calculation in sampling from K_+ for generating $Z|\alpha$ accounts for 70% of the time, approximately 1.4 seconds per iteration because it involves matrix inversion and likelihood calculation. Matrix inversion and determinant calculation are notoriously slow; they run between $O(n^2)$ and $O(n^3)$.

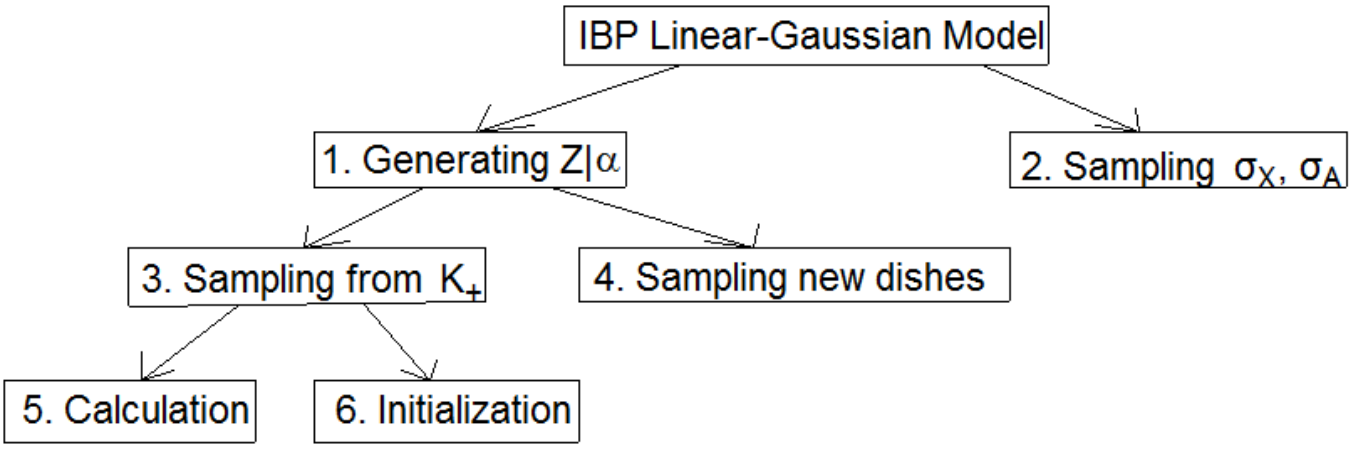

Figure 8: IBP code structure for profiling

5.2 Remove Redundant Calculations

To optimize the code, redundant calculations are removed first, and this version is named as "usable". When generating $Z|\alpha$, the inverted matrix $\mathbf{M} = (\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}\mathbf{I})^{-1}$ is only calculated directly before the likelihood computation, so more than $N = 100$ matrix inversions can be removed. The speed of generating $Z|\alpha$ is improved by 1.5%, 0.023 seconds per iteration, so this "usable" code is at least 23 seconds faster than the initial version. The profiling results are shown in Table 2.

5.3 Matrix Multiplication

Matrix multiplication is associative, say $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$, but the order of multiplication can affect the computation speed. According to dynamic programming [5], here is an example of how this works: The dimensions of matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are $4 \times 2, 2 \times 5, 5 \times 1$, respectively.

- $(\mathbf{AB})\mathbf{C}$: total multiplications $= 4 \times 2 \times 5 + 4 \times 5 \times 1 = 60$
- $\mathbf{A}(\mathbf{BC})$: total multiplications $= 2 \times 5 \times 1 + 4 \times 2 \times 1 = 18$

As a result, the order of multiplications can make a huge difference in computation.

In my IBP code, some matrix multiplications have the potential to be computed faster, but it turned out that either I have already selected the faster way, or the number of total multiplications are the same for both methods. First, to calculate the resulting features matrix $\mathbf{A}_{\text{inf}} = (\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}\mathbf{I})^{-1}\mathbf{Z}^T\mathbf{X}$, I can start from either the first two or the last two matrices. $\mathbf{Z}$ has size $N \times K_+ = 100 \times 4$; hence the first term $\mathbf{M} = (\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}\mathbf{I})^{-1}$ has size $K_+ \times K_+ = 4 \times 4$; the second term $\mathbf{Z}^T$ has size $K_+ \times N = 4 \times 100$, and the third term $\mathbf{X}$ has size $N \times D = 100 \times 36$.

- $(\mathbf{MZ}^T)\mathbf{X}$: total multiplications $= \mathbf{4 \times 4 \times 100} + 4 \times 100 \times 36$
- $\mathbf{M}(\mathbf{Z}^T\mathbf{X})$: total multiplications $= \mathbf{4 \times 4 \times 36} + 4 \times 100 \times 36$

Multiplying $\mathbf{Z}^T$ and $\mathbf{X}$ first is faster than doing the other way.

In addition to $\mathbf{A}_{\text{inf}}$, the kernel of the log-likelihood function involves calculating $\mathbf{X}^T(\mathbf{I} - \mathbf{ZMZ}^T)\mathbf{X}$. However, the middle term $(\mathbf{I} - \mathbf{ZMZ}^T)$ is a 100×100 square matrix, so multiplying the three matrices in either order requires $36 \times 100 \times 100 \times 2$ multiplications.

5.4 Cythonized Code

The "usable" version code can also be cythonized (converted from Python to C), and Table 3 is a summary of profiling results, but the Cythonized version only improved the speed about 1.5%.

5.5 Using jit (just-in-time compiler)

The jit (just-in-time compiler) is from the Python package `numba`, which generates optimized machine code from the LLVM compiler infrastructure. The jit in Python is claimed to have similar performance to C/C++ without switching languages [11]. The speed comparison table is shown in Table 4, but this version performs almost the same as the Python "usable" version.

5.6 Comparison Tables

All four tables summarizing which actions take how much time are here for ease of comparison.

	Time (seconds)/action	Times performed	Total time (seconds)
Generating Z given alpha	1.595599	1	1.595599
Sampling sigmaX, sigmaA	0.003429	1	0.003429
Sampling from K+	0.011332	100	1.133166
Sampling new dishes	0.004525	100	0.452525
Calculation	0.002427	500	1.213608
Initialization	0.000007	500	0.003504

Table 1: Initial code: Profiling results per iteration

	Time (seconds)/action	Times performed	Total time (seconds)
Generating Z given alpha	1.572500	1	1.572500
Sampling sigmaX, sigmaA	0.003756	1	0.003756
Sampling from K+	0.011194	100	1.119370
Sampling new dishes	0.004529	100	0.452930
Calculation	0.002219	500	1.109398
Initialization	0.000007	500	0.003420

Table 2: Usable code: Profiling results per iteration

	Time (seconds)/action	Times performed	Total time (seconds)
Generating Z given alpha	1.549314	1	1.549314
Sampling sigmaX, sigmaA	0.003515	1	0.003515
Sampling from K+	0.011046	100	1.104567
Sampling new dishes	0.004446	100	0.444569
Calculation	0.002154	500	1.077215
Initialization	0.000007	500	0.003283

Table 3: Cythonized code: Profiling results per iteration

6 Comparative Analysis

The Indian Buffet Process (IBP) is compared with the Chinese Restaurant Process [3], which is also a Bayesian nonparametric method to discover latent features in a given dataset. My implementation with simulated data is also compared with a Matlab version [13] and a Python version [2] online.

	Time (seconds)/action	Times performed	Total time (seconds)
Generating Z given alpha	1.575537	1	1.575537
Sampling sigmaX, sigmaA	0.005048	1	0.005048
Sampling from K+	0.011275	100	1.127496
Sampling new dishes	0.004478	100	0.447848
Calculation	0.002384	500	1.192065
Initialization	0.000007	500	0.003296

Table 4: Using jit (just-in-time compiler): Profiling results per iteration

6.1 Indian Buffet Process (IBP) vs Chinese Restaurant Process (CRP)

As mentioned in Section 1, the Chinese Restaurant Process (CRP) [3] is an algorithm of customers' seating in a Chinese restaurant with infinite capacity. The first customer sits at an empty table with probability 1. Then starting from time 2, a new customer chooses randomly at either to the left of one of the previous customers, or at a new, unoccupied table.

Both IBP and CRP model latent factors and perform dimensionality reduction (reduce the images or objects to latent features). They also both allow an infinite array of objects. Nevertheless, they solve different problems: IBP allows each customer to be assigned to multiple components (dishes), while CRP assigns each customer to a single component. Figure 9 from Gershman's and Blei's paper [8] illustrates the difference between draws of IBP and CRP.

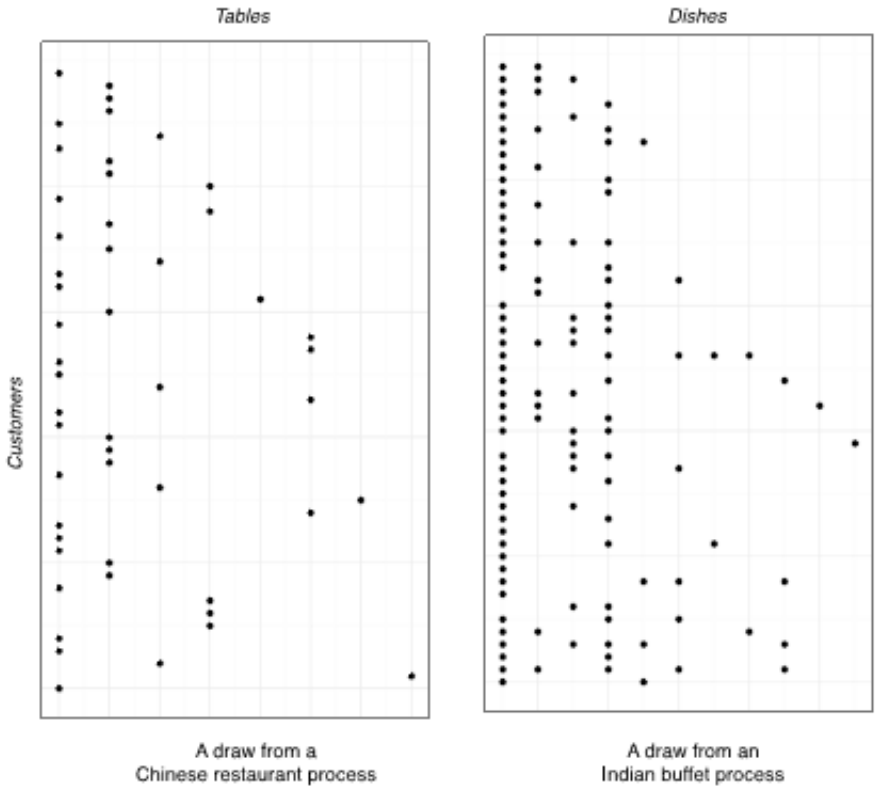

Figure 9: Indian Buffet Process (IBP) vs Chinese Restaurant Process (CRP) [8]

6.2 Indian Buffet Process: Another Matlab Version Online

The Matlab version I compared with is Yildirim's IBP sample code [13], and the simulated dataset in both versions of code are the same as in Section 2.1. For 1000 iterations of Gibbs sampling, the Matlab code takes about 400 seconds to run, while my Python version takes approximately 2000 seconds. The Matlab code is not only five times faster than my Python code, but also gives the correct four latent features, as in Figure 11.

The truncated profiling results using the Matlab tool "Run and Time" for the functions `sampler, likelihood, calInverse` are listed in Figure 10. The column "Self time" indicates the time spent in a function, but it includes the overhead time of profiling and excludes the time spent in its child functions. The function `sampler` refers to the whole Gibbs sampler; `likelihood` is the likelihood calculation, and `calInverse` is the implementation from Griffiths' and Ghahramani's paper[10], to compute $\mathbf{M}$ faster when only one $\mathbf{z_i}$ is changed.

Profile Summary

Generated 24-Apr-2015 17:48:49 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
sampler	1	402.363 s	66.328 s	
likelihood	1322364	267.251 s	248.235 s	
calcInverse	819364	51.405 s	51.405 s	

Figure 10: Matlab code: Profiling results (truncated)

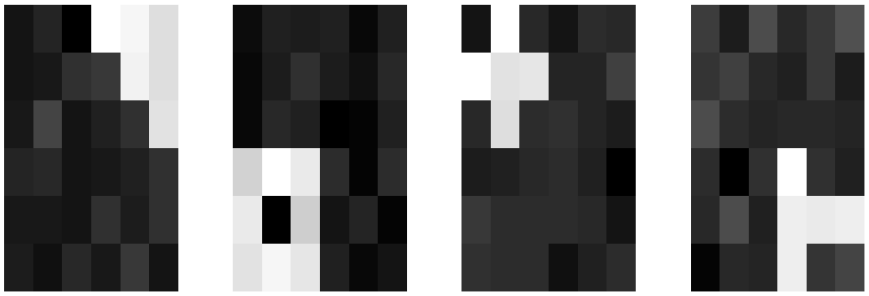

Figure 11: Matlab code: Latent features discovered

6.3 Indian Buffet Process: Another Python Version Online

The other Python version I compared with is Andrzejewski's PyIBP [2] on GitHub. This version has two advantages – speed and organization, but the two drawbacks are result inconsistency and difficulty

in execution (lack of Makefile). Note that these two disadvantages of PyIBP can be removed by adding small lines of code.

First, the accelerated Gibbs sampling [1, 7] makes the code much faster, and only five iterations are needed to generate the results. The accelerated Gibbs sampling not only exploits the conjugate normal prior and likelihood by rank-one $\mathbf{M}$ updates, but also uses slice sampling [1] to decompose sampling from the unnormalized posterior distribution of $\mathbf{X}$ into discrete steps of uniform distributions. For example, sampling from an arbitrary unnormalized distribution $\widetilde{p}(y)$ can be performed by the following steps, given a current value y and window boundaries (L, R) where y lies within:

1. Sample $u \sim \text{Unif}(0, \widetilde{p}(y))$
2. Sample $\widehat{y} \sim \text{Unif}(L, R)$
3. Accept new $\widehat{y}$ value if $\widetilde{p}(y) > u$, else reject

In this way, the expensive matrix multiplication in the normal likelihood kernel can be avoided.

Second, the PyIBP code is organized; the author generated the modules `PyIBP.py` and `scaledimages.py`, and the user can run the example file without needing to learn much about the IBP process.

However, the two drawbacks can cause problems in implementation, but these problems are easy to solve. To begin with, sometimes the PyIBP code produces excellent results like Figure 12, but sometimes the discovered latent features are noisy, such as in Figure 13. In both figures, the top four images are the ground-truth features, and the bottom shows the generated results. Setting a fixed random seed can ensure the results to be reproducible. For another disadvantage, a Makefile does not exist in the original PyIBP code, so it takes some time for users to figure out how to execute the PyIBP example. Therefore, I wrote a Makefile for convenience of execution.

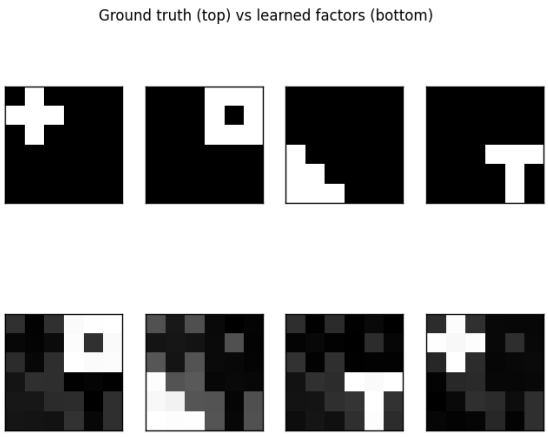

Figure 12: PyIBP code: Best results

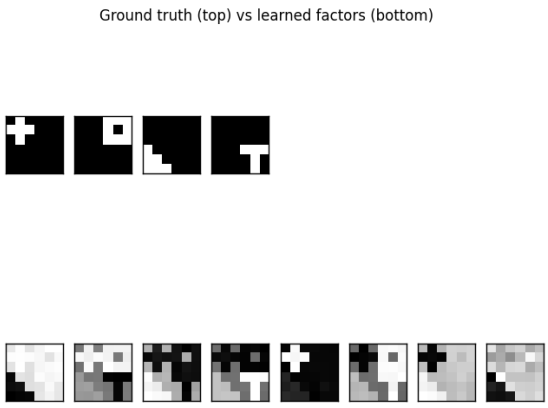

Figure 13: PyIBP code: Worst results

7 Conclusion

The Indian Buffet Process (IBP) is a Bayesian nonparametric method to discover the latent structure of datasets, and it serves as the prior for many infinite latent feature models. In fact, IBP is a widely used approach in unsupervised machine learning. However, one limitation is that the results of using IBP are sensitive to the starting-point random seed settings, so one should run the code with multiple random seeds and select the one with best performance.

Several methods can make the IBP implementation code faster, such as removing redundant calculations and Cythonizing Python code. But to gain significant improvement in speed, matrix inversions need to be done in a less computationally-expensive approach because they take the most time and resources to perform one calculation.

References

[1] D. M. Andrzejewski. Accelerated gibbs sampling for infinite sparse factor analysis. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2011.

[2] David Andrzejewski. Python ibp (pyibp). `https://github.com/davidandrzej/PyIBP`. Online; accessed 2015.

[3] D. M. Blei, T. L. Griffiths, M. I. Jordan, and J. B. Tenenbaum. Hierarchical topic models and the nested chinese restaurant process. *Advances in neural information processing systems*, 16:17, 2004.

[4] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[5] Yao-Wen Chang. Unit 4: Dynamic programming. `http://cc.ee.ntu.edu.tw/~ywchang/Courses/Alg/unit4.pdf`, 2014. Online; accessed 2015.

[6] Finale Doshi, Kurt Tadayuki Miller, Jurgen Van Gael, and Yee Whye Teh. Variational inference for the indian buffet process. In *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics*, volume 5, 2008.

[7] Finale Doshi-Velez and Zoubin Ghahramani. Accelerated sampling for the indian buffet process. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 273–280. ACM, 2009.

[8] Samuel J Gershman and David M Blei. A tutorial on bayesian nonparametric models. *Journal of Mathematical Psychology*, 56(1):1–12, 2012.

[9] Thomas Griffiths and Zoubin Ghahramani. Infinite latent feature models and the indian buffet process. In *Advances in Neural Information Processing Systems*, volume 18. NIPS Proceedings, 2005.

[10] Thomas Griffiths and Zoubin Ghahramani. Infinite latent feature models and the indian buffet process. Technical report, Gatsby Computational Neuroscience Unit, 2005.

[11] Numba package in Python. Numba. http://numba.pydata.org/. Online; accessed 2015.

[12] Sinead Williamson, Chong Wang, Katherine Heller, and David Blei. Focused topic models. In *NIPS workshop on Applications for Topic Models: Text and Beyond, Whistler, Canada*, 2009.

[13] Ilker Yildirim. Indian buffet process – sample code. http://www.mit.edu/~ilkery/. Online; accessed 2015.

[14] Ilker Yildirim. Bayesian statistics: Indian buffet process. http://www.bcs.rochester.edu/people/robbie/jacobslab/cheat_sheet/IndianBuffetProcess.pdf, 2012. Online; accessed 2015.

[15] Ilker Yildirim and Robert A Jacobs. A bayesian nonparametric approach to multisensory perception. In *The annual meeting of the Cognitive Science society*, pages 2633–2638, 2010.