

# STA663 Final Project

## Indian Buffet Process and its application in the Infinite Latent Feature Model

Dipesh Gautam

April 29, 2015

## 1 Background

### 1.1 Indian Buffet Process (IBP)

The Indian Buffet is an adaptation of Chinese Buffet Process where each object instead of being associated with a single latent class can be associated with multiple classes. This is particularly useful when each object has multiple latent features and by associating objects with a single class we cannot partition them into homogeneous subsets.

In the Indian buffet process,  $N$  customers enter a restaurant one after another. Each customer encounters a buffet consisting of infinitely many dishes arranged in a line. The first customer starts at the left of the buffet and takes a serving from each dish, stopping after a  $\text{Poisson}(\alpha)$  number of dishes. The  $i$ th customer moves along the buffet, sampling dishes in proportion to their popularity, taking dish  $k$  with probability  $\frac{m_k}{i}$ , where  $m_k$  is the number of previous customers who have sampled that dish. Having reached the end of all previous sampled dishes, the  $i$ th customer then tries a  $\text{Poisson}(\frac{\alpha}{i})$  number of new dishes. Which customer chose which dishes is indicated using a binary matrix  $\mathbf{Z}$  with  $N$  rows and infinitely many columns (corresponding to the infinitely many selection of dishes), where  $z_{ik} = 1$  if the  $i$ th customer sampled  $k$ th dish.

IBP can be used as a prior in models for unsupervised learning. An example of which is presented in the paper by Griffiths and Ghahramani, where IBP is used as a prior in linear-Gaussian binary latent feature model.

## 2 Implementation

### 2.1 Infinite Latent Features Model and Algorithm

- Gamma prior for  $\alpha$

$$\alpha \sim \text{Gamma}(1, 1)$$

- Prior on  $\mathbf{Z}$  is obtained by IBP as:

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

- Likelihood is given by

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

After we have the likelihood and the prior given by IBP,

- full conditional posterior for  $\mathbf{Z}$  can be calculated as:

$$P(z_{ik}|X, Z_{-(i,k)}, \sigma_X, \sigma_A) \propto P(X|Z, \sigma_X, \sigma_A) * P(z_{ik} = 1|\mathbf{z}_{-i,k})$$

To sample the number of new features for observation  $i$ , we use a truncated distribution, computing probabilities for a range of values  $K_1^{(i)}$  up to an upper bound (say 4). The prior on number of features is given by  $Poisson(\frac{\alpha}{N})$ . Using this prior and the likelihood, we sample the number of new features.

- Full conditional posterior for  $\alpha$  is given by:

$$P(\alpha|Z) \sim Gamma(1 + K_+, 1 + \sum_{i=1}^N H_i)$$

- For  $\sigma_X$  and  $\sigma_A$ , we use MH algorithm as follows:

$$\epsilon \sim Uniform(-.05, .05) \quad (2)$$

$$\sigma_X^* = \sigma_X + \epsilon \quad (3)$$

$$(4)$$

Accept this new  $\sigma_X$  with probability given by:

$$AR = \min\{1, \frac{Likelihood(X|\sigma_X, \dots)}{Likelihood(X|\sigma_X, \dots)}\}$$

Where AR is the acceptance ratio. We use similar algorithm to sample  $\sigma_A$

## 2.2 Simulated Data

## 2.3 Unit Testing

# 3 Profiling and Optimization

We profiled the code using *cProfile* to figure out bottlenecks. The result is shown in *Profiling result*. We see that most of the computational time is spent on calculating the *log likelihood(ll)* and matrix inversion. Due to this fact, one of the first things we looked at were ways to reduce computation time for likelihood and/or inverse calculation.

### Profiling Result

2075808 function calls in 10.873 seconds

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno:function
154080	3.985	0.000	3.985	0.000	{method 'dot' of 'numpy.ndarray' objects}
30816	1.540	0.000	9.411	0.000	<ipython-input-144-816e3f6a3e53>:211
30816	0.770	0.000	1.353	0.000	linalg.py:455:inv
1	0.726	0.726	10.873	10.873	<ipython-input-145-4efe9a6e9287>:1:sampler
30816	0.542	0.000	1.071	0.000	linalg.py:1679:det
61632	0.382	0.000	0.963	0.000	numeric.py:2125:identity
61632	0.367	0.000	0.580	0.000	twodim_base.py:190:eye
30816	0.332	0.000	0.332	0.000	{method 'trace' of 'numpy.ndarray' objects}
86971	0.267	0.000	0.267	0.000	{numpy.core.multiarray.zeros}
61632	0.183	0.000	0.314	0.000	linalg.py:139:commonType
122449	0.171	0.000	0.171	0.000	{numpy.core.multiarray.array}
20964	0.135	0.000	0.135	0.000	{method 'reduce' of 'numpy.ufunc' objects}

30816	0.114	0.000	0.114	0.000	{method 'astype' of 'numpy.ndarray' objects}
92448	0.113	0.000	0.251	0.000	numeric.py:394asarray
30816	0.107	0.000	0.107	0.000	{method 'astype' of 'numpy.generic' objects}
71965	0.104	0.000	0.104	0.000	{max}
15000	0.092	0.000	0.092	0.000	{numpy.core.multiarray.concatenate}
61632	0.089	0.000	0.147	0.000	linalg.py:209_assertNdSquareness
30000	0.084	0.000	0.166	0.000	shape_base.py:8atleast_1d
61632	0.073	0.000	0.086	0.000	linalg.py:198_assertRankAtLeast2
184896	0.071	0.000	0.071	0.000	{issubclass}
30816	0.067	0.000	0.500	0.000	fromnumeric.py:1233trace
20964	0.053	0.000	0.231	0.000	fromnumeric.py:1631sum
123264	0.052	0.000	0.080	0.000	linalg.py:111isComplexType
30816	0.052	0.000	0.052	0.000	linalg.py:101get_linalg_error_extobj
15000	0.049	0.000	0.307	0.000	shape_base.py:230hstack
30816	0.048	0.000	0.129	0.000	linalg.py:106_makearray
10980	0.040	0.000	0.040	0.000	{method 'uniform' of 'mtrand.RandomState' objects}
30816	0.037	0.000	0.037	0.000	linalg.py:219_assertNoEmpty2d
61632	0.036	0.000	0.048	0.000	linalg.py:124_realType
30000	0.030	0.000	0.063	0.000	numeric.py:464asanyarray
20964	0.026	0.000	0.026	0.000	{isinstance}
121632	0.024	0.000	0.024	0.000	{len}
5000	0.019	0.000	0.019	0.000	{sum}
20964	0.017	0.000	0.152	0.000	_methods.py:31_sum
61732	0.015	0.000	0.015	0.000	{min}
61632	0.012	0.000	0.012	0.000	{method 'get' of 'dict' objects}
20000	0.011	0.000	0.011	0.000	{math.factorial}
15152	0.011	0.000	0.011	0.000	{range}
30816	0.010	0.000	0.010	0.000	{getattr}
30000	0.009	0.000	0.009	0.000	{method 'append' of 'list' objects}
30816	0.007	0.000	0.007	0.000	{method '__array_prepare__' of 'numpy.ndarray' objects}
1	0.002	0.002	0.005	0.005	<ipython-input-143-ed70069a6371>:2sampleIBP
100	0.000	0.000	0.000	0.000	{method 'poisson' of 'mtrand.RandomState' objects}
50	0.000	0.000	0.000	0.000	{method 'gamma' of 'mtrand.RandomState' objects}
4	0.000	0.000	0.000	0.000	{numpy.core.multiarray.copyto}
4	0.000	0.000	0.000	0.000	{numpy.core.multiarray.empty}
4	0.000	0.000	0.000	0.000	numeric.py:141ones
1	0.000	0.000	0.000	0.000	{method 'seed' of 'mtrand.RandomState' objects}
1	0.000	0.000	10.873	10.873	<string>:1<module>
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

### 3.1 Matrix Inversion

We tried the matrix inversion method described in Griffiths and Ghahramani(2005, eq 51-54), where the method reduces the runtime by allowing us to perform rank one updates instead when only one value is changed. We implemented the algorithm and were able to speed up the process as shown in Table 1.

Table 1: Comparison of matrix inverse methods

	Time
linalg.inverse	0.000080
calcInverse	0.000038

Even though we were able to improve the performance, due to some numerical errors, we were not able to obtain a stationary MCMC chain using this method. This could be achieved by spending some more time on it but due to time constraints we decided that to look at fixing this at a later time.

### 3.2 Likelihood function

While working on the optimized matrix inversion, we noticed that the matrix that we’re inverting i.e.  $(Z^T Z + \frac{\sigma_X^2}{\sigma_A^2} I)$  actually appears twice inside the likelihood function. So, we looked at removing the redundancy by calculating the matrix and storing it. We were able to gain some improvement using this method as shown in Table 2 and Table 3. Since the likelihood function is called numerous times, even the small gain shown in Table 2 was translated into a substantial gain as shown in Table 3.

Table 2: Runtime Comparision

	Time
original ll function	0.000403
Proposed ll function	0.000337

### 3.3 Cython

Another way we looked at improving the performance of the code was by cythonizing the code. We again looked at improving the performance of the likelihood function by cythonizing it. As shwon in Table 3, we were not able to gain substantial improvements from it.

Table 3: Runtime Comparision

	Total Time
Initial Code	539.818143
Improved ll	489.207815
Cythonized	507.522965

### 3.4 Parallelization and CUDA

Since our algorithm is an MCMC algorithm with serial dependence, parallelization does not seem to be a good idea. One of the ways, parallelization can be done is by splitting the chain into multiple smaller chains and combining them back. We tested it and it showed some improvement in the code but decided against using it as the gain wasn’t significant enough as we had to take care of multiple burn-in periods and ignore the loss of Markov property due to multiple chains. Also, parallelizing the density calculation for likelihood wasn’t an option for our algorithm as we had a discrete density with just two points.

We also looked at optimizing the code through using CuBLAS library from CUDA. We tested the performance of matrix multiplication using *np.dot* and *cublas.Blas* and found the CuBLAS multiplication to be slower. This can be attributed to the fact that we are working with relatively small matrices of dimensions like  $K \times K$ ,  $100 \times K$  where  $K$  is smaller than 10 and  $100 \times 36$ . So, the overhead in was larger than the gain from gpu calculation resulting in slower time for CuBLAS matrix multiplication.

## 4 Results

We ran our sampler with the improved likelihood calculation for 1000 iterations. Our data was simulated with 4 latent features. The distribution of  $K_+$ (total features detected at each iteration) is shown in Fig. 1(a). Although we see that the mode of the number of features is 5, with significant iterations where the number of features is 7, we can see from Fig. 1(b) that most of the objects actually had between 1 and 4 features with very few of them with 5 or more features. These outliers can be attributed to the noise and

variance in the data simulation and ignored. So, we can conclude that the algorithm correctly predicts and detects the existence of 4 latent features in the simulated data.

Trace plot for  $\sigma_X$ ,  $\sigma_A$ , and  $\alpha$  is shown in Fig. 2. We see that  $\sigma_X$  is converging to it's true value of 0.5 and the other parameters also show proper convergence validating the authenticity of the algorithm.

Figure 1: Distribution of Kplus and mean number of features per object

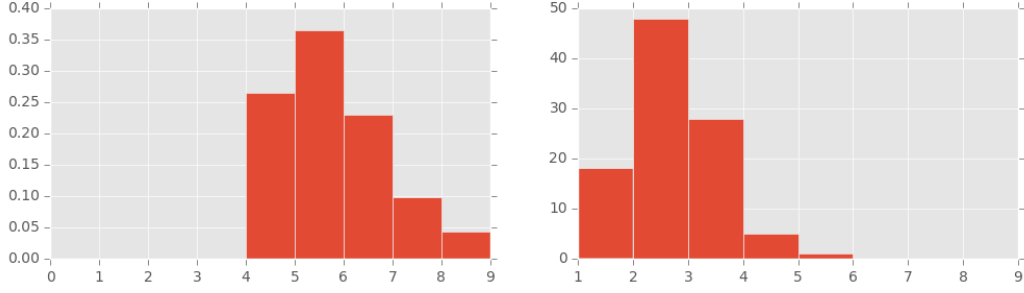


Figure 2: Traceplots for  $\sigma_X$ ,  $\sigma_A$  and  $\alpha$  after burn-in

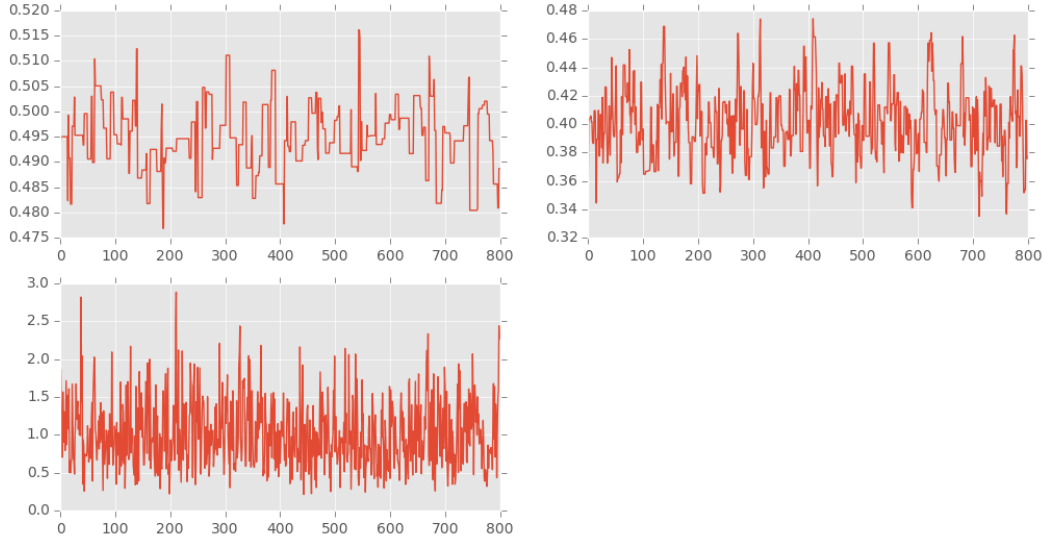


Figure 3: Original Features and First four simulated objects

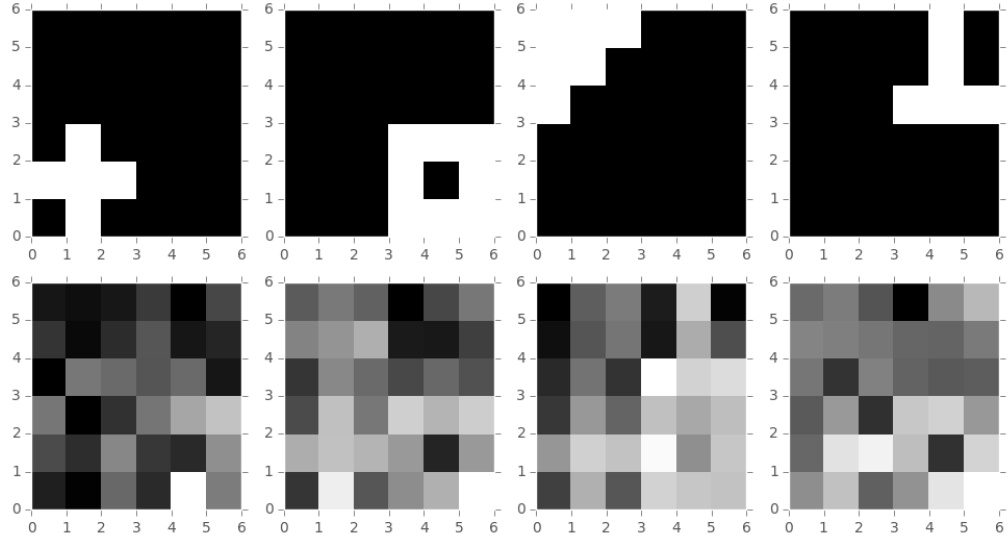


Table 4: Features present in first four original images

	F1	F2	F3	F4
1st image	0	1	0	0
2nd image	1	1	0	0
3rd image	1	1	0	1
4th image	1	1	0	0

Figure 4: Features Detected after MCMC and First four recreated objects

