

# STA663: Final Project

## Infinite Latent Feature Models and the Indian Buffet Process

Radhika Anand

April 30, 2015

## 1 Background

### 1.1 Definition of the Indian Buffet Process

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  $m_k/i$ , 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  $\text{Poisson}(\alpha/i)$  number of new dishes.

### 1.2 Steps

- 1) Start by defining a probability distribution over equivalence classes of binary matrices with a finite number of rows and an unbounded number of columns. This distribution is suitable for use as a prior in probabilistic models that represent objects using a potentially infinite array of features
- 2) Next we see that the Indian Buffet Process is a simple generative process that results in the same distribution (as described in previous step) over equivalence classes because IBP has finite number of objects (people) and infinite number of features (dishes)
- 3) We then define a Gibbs Sampler for models using this concept of IBP
- 4) Further, to illustrate how IBP can be used as a prior in models for unsupervised learning, we derive and test a linear-Gaussian latent feature model in which the features are binary
- 5) We start with a finite realization and then take the infinite limit

## 2 Implementation

We illustrate how IBP can be used as a prior in models for unsupervised learning by deriving and testing an infinite Gaussian binary latent feature model, presented in Griffiths and Ghahramani (2005) [1] with further implementation in Yildirim (2012) [3].

### 2.1 Infinite Linear-Gaussian Binary Latent Feature Model

In this model, we consider a binary feature ownership matrix  $Z$  which illustrates the presence or absence of underlying features in the objects  $X$ . The  $D$ -dimensional vector of properties of an object  $i$ ,  $x_i$  is generated as  $x_i \sim N(z_i A, \Sigma_X)$ , where  $A$  is a  $K \times D$  matrix of weights,  $K$  represents the underlying latent features and  $\Sigma_X = \sigma_X^2 I$  introduces the white noise.

### 2.2 Algorithm

We use a combination of Gibbs Sampling and Metropolis Hastings to update the parameters of interest, which are:

- $Z$  - Feature ownership matrix
- $K_{newdishes}$  - New dishes/features sampled
- $\alpha$  - Poisson parameter
- $\sigma_X$  - Parameter for  $X$
- $\sigma_A$  - Parameter for weight matrix  $A$

#### 2.2.1 Likelihood

The likelihood is given by (integrating out  $A$ ):

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

#### 2.2.2 Gibbs Sampler

1) Gamma prior is used for  $\alpha$

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

2) Prior on  $Z$  is obtained by IBP (after taking the infinite limit) as:

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

where  $z_{-i,k}$  is the set of assignments of other objects, not including  $i$ , for feature  $k$ , and  $m_{-i,k}$  is the number of objects possessing feature  $k$ , not including  $i$ .

3) The prior on number of features is given by  $Poisson(\frac{\alpha}{N})$

4) Using the likelihood above and the prior given by IBP, full conditional posterior for  $Z$  can be calculated as:

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

5) To sample the number of new features,  $K_{newdishes}$ , for observation  $i$ , we use a truncated distribution, computing probabilities for a range of  $K_{newdishes}^{(i)}$  up to an upper bound.

6) Conditional posterior for  $\alpha$  is given by:

$$P(\alpha|Z) \sim Gamma(1 + K_+, 1 + H_N)$$

where  $H_N$  is the  $N$ th harmonic number given by  $H_N = \sum_{j=1}^N 1/j$  and  $K_+$  is the current number of features.

### 2.2.3 Metropolis Hastings

1) To update  $\sigma_X$  and  $\sigma_A$ , we use MH algorithm as follows:

$$\begin{aligned} \epsilon &\sim Uniform(-.05, .05) \\ \sigma_X^* &= \sigma_X + \epsilon \\ \sigma_A^* &= \sigma_A + \epsilon \end{aligned}$$

Accept this new  $\sigma_X^*$  with acceptance probability:

$$AP = \min\{1, \frac{P(X|Z, \sigma_X^*, \sigma_A)}{P(X|Z, \sigma_X, \sigma_A)}\}$$

Similarly for  $\sigma_A^*$ .

## 3 Profiling and Optimization

We profiled the basic version of the code to find the functions or parts of code taking significant amounts of time. The results of the profiler are shown below. We clearly see that the functions; sampler, likelihood and inverse take the maximum amount of time.

---

Profiler Results

---

2184313 function calls in 9.094 seconds

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno:function
286528	3.351	0.000	3.351	0.000	{numpy.core._dotblas.dot}
35816	1.212	0.000	6.131	0.000	<ipython-input-88-d1ee8a415007>:2:likelihood
1	0.988	0.988	9.094	9.094	<ipython-input-19-5b64a3c9505b>:3:sampler
35816	0.495	0.000	1.086	0.000	linalg.py:455:inv
107448	0.412	0.000	0.758	0.000	twodim_base.py:190:eye
112552	0.355	0.000	0.355	0.000	{numpy.core.multiarray.zeros}
35816	0.333	0.000	0.778	0.000	linalg.py:1679:det
35816	0.314	0.000	0.314	0.000	{method 'trace' of 'numpy.ndarray' objects}
21332	0.255	0.000	0.255	0.000	{sum}
71632	0.189	0.000	0.329	0.000	linalg.py:139:_commonType
112782	0.120	0.000	0.120	0.000	{numpy.core.multiarray.array}
71632	0.092	0.000	0.149	0.000	linalg.py:209:_assertNdSquareness
107448	0.090	0.000	0.191	0.000	numeric.py:394:asarray
82361	0.090	0.000	0.090	0.000	{max}
35816	0.081	0.000	0.081	0.000	{method 'astype' of 'numpy.generic' objects}
10980	0.077	0.000	0.077	0.000	{method 'uniform' of 'mtrand.RandomState' objects}
71632	0.075	0.000	0.090	0.000	linalg.py:198:_assertRankAtLeast2
214896	0.070	0.000	0.070	0.000	{issubclass}
35816	0.068	0.000	0.068	0.000	{method 'astype' of 'numpy.ndarray' objects}
35816	0.060	0.000	0.060	0.000	linalg.py:101:_get_linalg_error_extobj
143264	0.057	0.000	0.090	0.000	linalg.py:111:isComplexType
35816	0.057	0.000	0.451	0.000	fromnumeric.py:1225:trace
35816	0.053	0.000	0.120	0.000	linalg.py:106:_makearray
71632	0.038	0.000	0.054	0.000	linalg.py:124:_realType
35816	0.035	0.000	0.035	0.000	linalg.py:219:_assertNoEmpty2d
71732	0.017	0.000	0.017	0.000	{min}
71681	0.016	0.000	0.016	0.000	{method 'get' of 'dict' objects}
72024	0.015	0.000	0.015	0.000	{len}
25000	0.013	0.000	0.013	0.000	{math.factorial}
15201	0.013	0.000	0.013	0.000	{range}
35865	0.012	0.000	0.012	0.000	{getattr}
35816	0.007	0.000	0.007	0.000	{method '__array_prepare__' of 'numpy.ndarray' objects}
1	0.006	0.006	0.016	0.016	<ipython-input-15-51bada944323>:2:sampleIBP
343	0.002	0.000	0.002	0.000	{method 'send' of 'zmq.backend.cython.socket.Socket' objects}
196	0.002	0.000	0.005	0.000	encoder.py:212:iterencode
300	0.002	0.000	0.028	0.000	iostream.py:207:write
1274	0.001	0.000	0.002	0.000	{method 'sub' of '_sre.SRE_Pattern' objects}
49	0.001	0.000	0.002	0.000	uuid.py:101:_init__
1274	0.001	0.000	0.003	0.000	encoder.py:33:encode_basestring

---

### 3.1 Optimizing Matrix Inverse

We began by optimizing the matrix inverse function. We used the inverse method described in Griffiths and Ghahramani (2005) [1], eqns. 51-54, to code an inverse function which involved only rank 1 updates instead of full rank updates. We can see in Table 1 that this *calcInverse* takes nearly half the time taken by the *linalg.inv* function in python (tested for 1000 iterations). But while running this in our code we could not obtain a stable Markov Chain since this inverse is just a numerical approximation and accumulates numerical errors on the way. We, hence, used the basic python function itself.

Table 1: Runtimes for inverse functions (for 1000 loops)

	Time
linalg.inv	0.033343
calcInverse	0.025014

### 3.2 Optimizing Likelihood Function and the Sampler

In the basic version of the code, we had a few redundant calculations in the likelihood function. We calculated the inverse of  $Z^T Z + \frac{\sigma_X^2}{\sigma_A^2} I$  matrix in the sampler each time before sending it to the likelihood function. Then in the likelihood function we have determinant of this same matrix,  $Z^T Z + \frac{\sigma_X^2}{\sigma_A^2} I$ . To get rid of the redundancy, we remove all inverse calculations outside the likelihood function and instead just calculate  $Z^T Z + \frac{\sigma_X^2}{\sigma_A^2} I$  once in the likelihood function and then take its inverse and determinant. This reduced the time taken by the likelihood function by about half as can be seen in Table 2 and Table 3.

Next, we vectorized a basic loop inside the sampler and got rid of redundant if-else statements. Thereafter, we could not find scope for more vectorization or basic optimization.

Finally, Table 4 shows the total runtimes for 1000 iterations of the sampler. We see that there is a significant decrease in the time taken by the optimized version compared to the naive one.

Table 2: Runtimes for likelihood function (for 1000 loops)

	Time (in secs)
Old Likelihood	0.330290
New Likelihood	0.161378

Table 3: Runtimes for likelihood function (for 1000 loops)

	Time (in secs)
Old Likelihood	0.170027
New Likelihood	0.148215

### 3.3 Cythonizing

To further optimize the code, we cythonized the optimized likelihood function. From Table 4, we see that the optimization gain by cythonizing is not much.

Table 4: Total runtimes

	Time (in secs)
Naive	340.578384
Optimized	333.326707
Cythonized	347.602671

## 4 High Performance Computing

We tried multicore programming and GPU programming to further reduce the total run times.

### 4.1 Multicore Programming

The MCMC sampler is serially dependent in its iterations and hence it is not the best idea to parallelize it. But we saw that the sampler stabilized in around 200 iterations and hence instead of running 1000 iterations on the same core we ran 2 chains of 500 each on 2 cores. The combined samples (after burn-in on each core) would not satisfy the Markov property in the theoretical sense of it but would still help us approximate the posterior distributions correctly since both the chains were stable. This reduced the run-time slightly but not significantly, as we also had to take care of multiple burn-ins and multicore overhead. Further, splitting a single chain likelihood calculation into multiple cores is not an option for us since we are calculating the density only at 2 discrete points.

### 4.2 GPU Programming

Next we looked at using CuBLAS library from the CUDA package to optimize the matrix multiplications but since we are working with relatively small matrices the overhead was very large and the basic matrix multiplication function *np.dot* was found to be faster than CuBLAS matrix multiplication.

Thus, we use the optimized likelihood and sampler described in Section 3.2 as the final version.

## 5 Unit Testing

We create several unit tests (refer `unit_tests.py` file) to test the various functions we have used in the code. All the tests pass. They are:

- 1) Test the `calcInverse` function and see if it is almost equal to the inverse obtained by using `np.linalg.inv`. We see that sometimes it is very close (upto default 7 decimal places) and sometimes it isn't (i.e. it is equal to around 2 decimal places and is thus a numerical approximation as described in Section 3.1).
- 2) Test if the posterior mean of the parameters  $\alpha$  and  $\sigma_X$  are equal to their true values
- 3) Test if the likelihood is positive
- 4) Test if the likelihood throws an error if we pass a parameter that causes division by zero
- 5) Test if the likelihood throws an error if we pass negative values for parameters  $\sigma_X$  and  $\sigma_A$ , as they have to be positive
- 6) Test if each object sampled atleast one feature

## 6 Application and Results

We simulate a basic dataset and present and validate our results below.

### 6.1 Data Simulation

We simulate an image dataset to test our code. The data is similar to that used in Griffiths and Ghahramani (2005) [1]. The data is as follows:

- $N = 100$  is the number of images (customers in IBP or objects in general)
- $D = 6 \times 6$  (image dimension) = 36 is the length of vectors (dishes or features) for each image
- $K = 4$  is the number of basis images (or latent variables)
- $X$  represents the final images generated using the  $K$  basis images (each basis is present with 0.5 probability) and added white noise

Thus we simulate 100,  $6 \times 6$  images represented as a  $100 \times 36$  matrix where each image/object has a  $D$ -dimensional vector of properties,  $x_i$ :

- $x_i \sim N(z_i A, \sigma_X^2 I)$
- $z_i$  is a  $K$ -dimensional binary vector (for presence or absence of features)
- $A$  is a  $K \times D$  matrix of weights

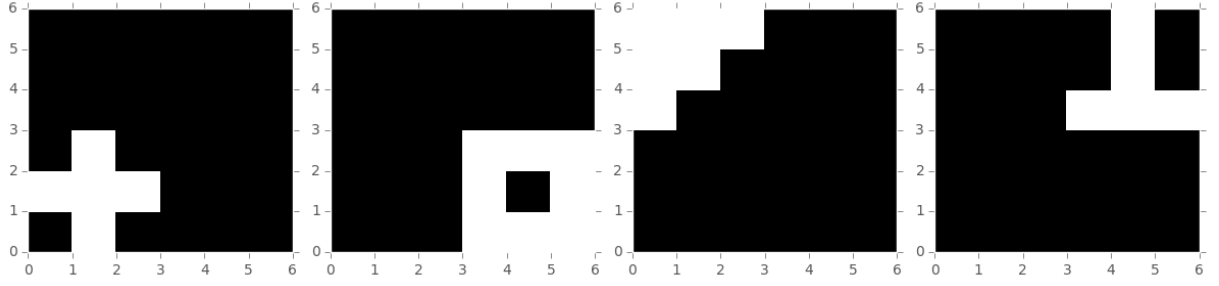


Figure 1: Features/basis images used to simulate data

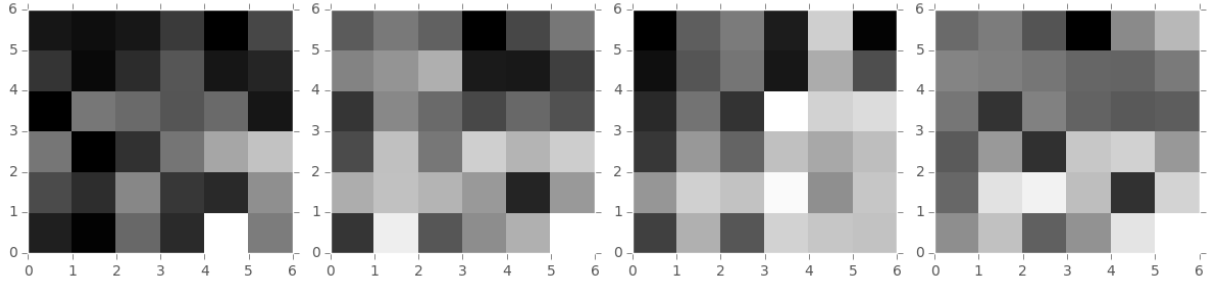


Figure 2: Simulated data (first four of 100 images)

Figure 1 shows the 4 features (basis images) used to generate our simulated data and Figure 2 shows first four of the 100 simulated images which have one or more of the features and added noise.

## 6.2 Results

We ran our code for 1000 iterations of the sampler to get convergence to the true values, for  $K$ ,  $\alpha$ ,  $\sigma_X$  and  $\sigma_A$  as can be seen in the trace plots in Figure 3.

### 6.2.1 Detection of total number of latent features

In Figure 4 (a), we see that the mode of  $K$  is around six because the samples tended to include the four features used by a large number of images/objects and then a few features used by one one or two objects (which came in the form of added noise). Figure 4 (b) shows the mean frequency with which objects tended to possess the features. We clearly see that most of the objects possessed only features 1, 2, 3 and 4. The extra features (5, 6 etc.) are possessed by very few objects which confirms that they are because of noise and not actual features.

We, thus, conclude the posterior of  $K$  to be 4, i.e. our code detected 4 latent features to be present in the data, which is as we would expect because we used 4 features to simulate the data in the beginning.



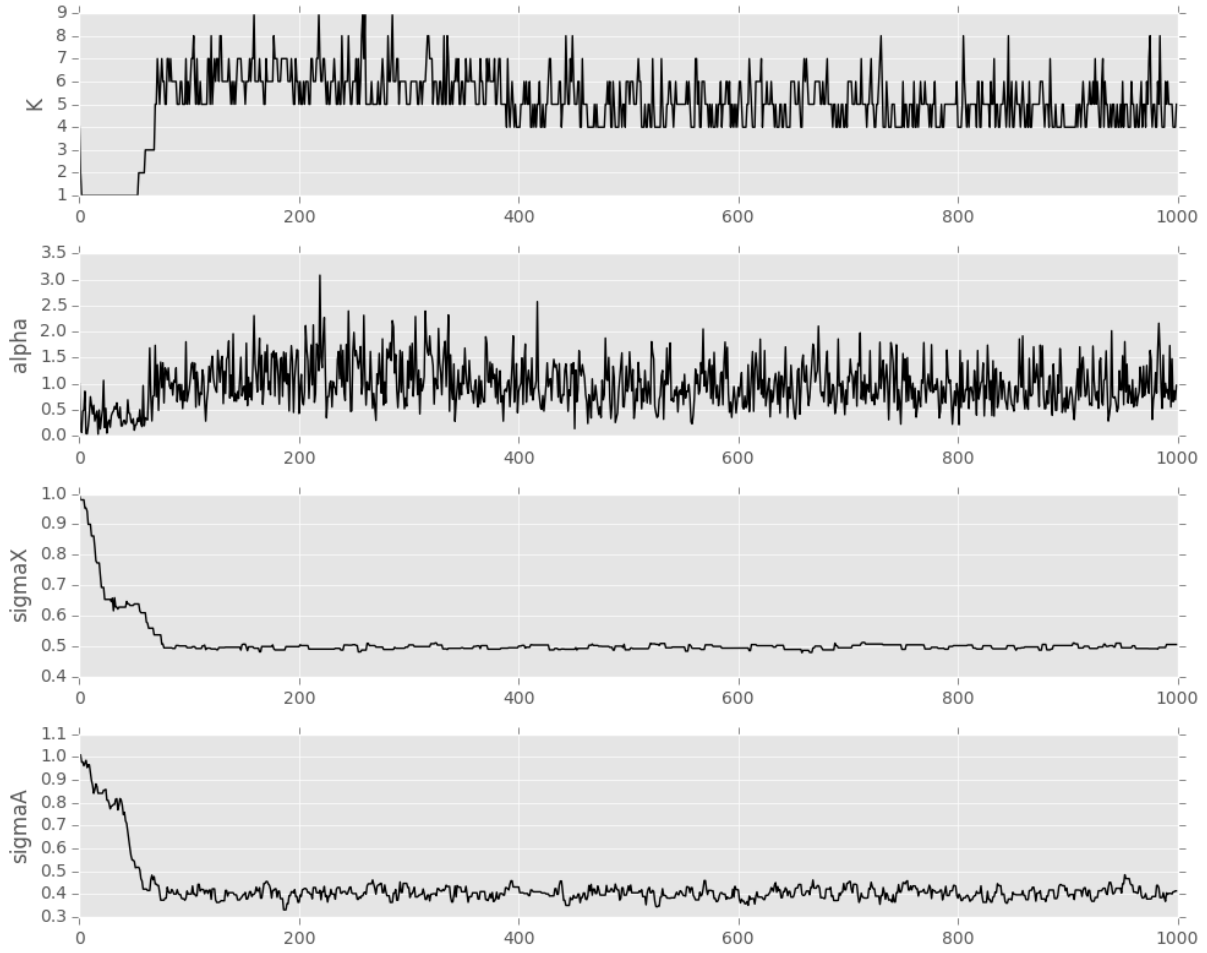


Figure 3: Trace plots for  $K$ ,  $\alpha$ ,  $\sigma X$  and  $\sigma A$

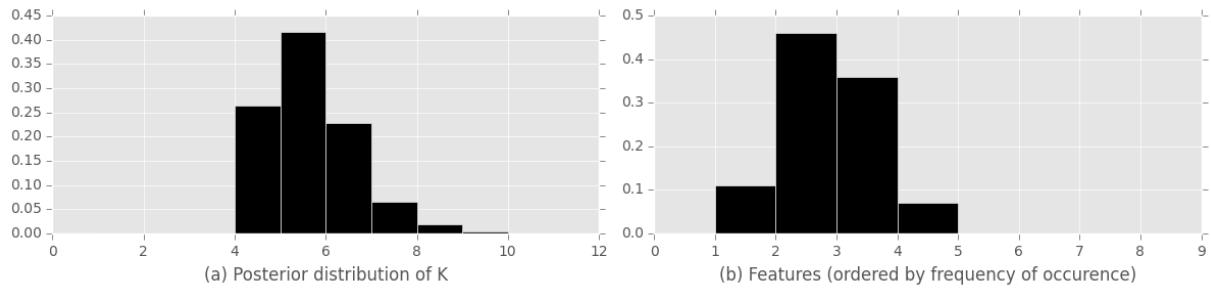


Figure 4: Histograms. (a) Posterior of  $K$  (b) Features (ordered by frequency of occurrence)

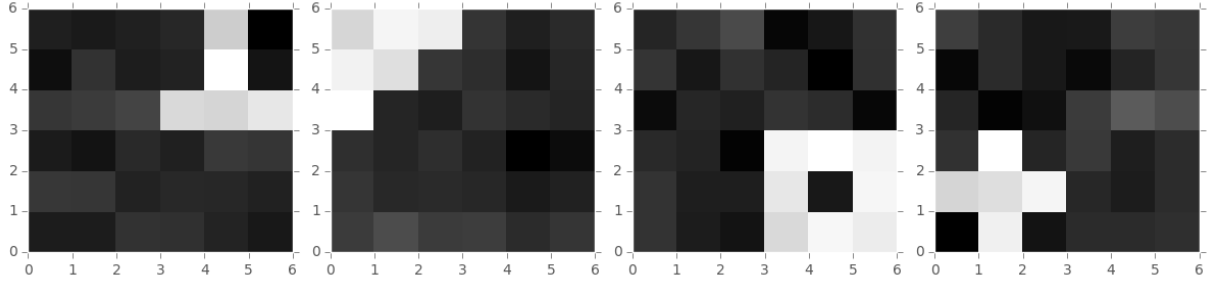


Figure 5: Features detected by code

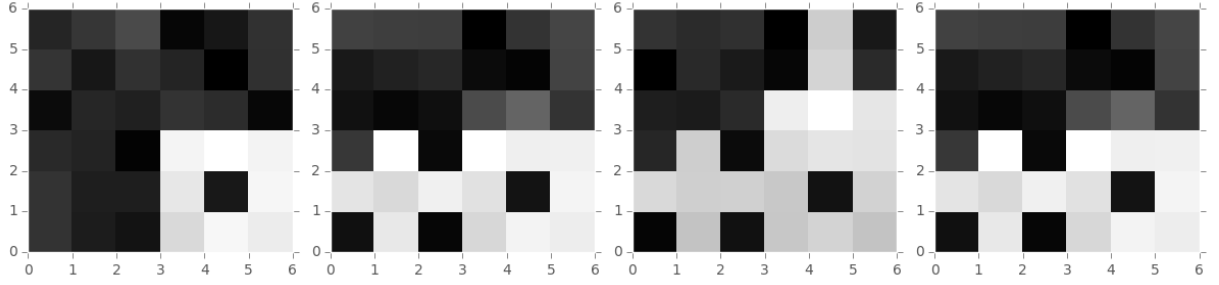


Figure 6: Reconstructed images

### 6.2.2 Detection of latent features present in each object and Object Reconstruction

Figure 5 shows the four most frequent features after the 1000 iterations of the sampler. We see that these features are the same as the features used to simulate the data as in Figure 1. They are just re-ordered.

Next, we reconstruct the images using  $X_i \sim N(Z_i A, 0)$ , where the posterior mean of the feature weights  $A$ , given  $X$  and posterior means of  $Z$ ,  $\sigma_A$  and  $\sigma_X$  is:

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

Figure 6 shows the posterior means of the reconstructions of the four original data images as in Figure 2. The reconstructions provided by the model in Figure 6 clearly pick out the relevant features present in each image, despite the high level of noise as seen in Figure 2.

### 6.2.3 Validation

To check the validity, Table 5 shows the features initially present (used to simulate) in the first four simulated images, where F1, F2, F3 and F4 refer to the order of features in Figure 1. We clearly see that the reconstructed images, as in Figure 6, pick exactly the same features. The first reconstructed image (refer Figure 6) picks feature 2, the 2nd picks 1 and 2, 3rd picks 1, 2

and 4 and 4th again picks 1 and 2 (feature numbering is as in Figure 1). This result shows that reconstructed images picked exactly the same features as were used to simulate them (Table 5). This validates our model.

Table 5: Presence/absence of latent features in the simulated data. 1 denotes presence, 0 denotes absence. F1, F2, F3, F4 refer to the 4 features as in Figure 1

	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

## 7 Comparison

We compare our algorithm with an implementation of the same algorithm in MATLAB. We also contrast our algorithm to another similar problem called Chinese Restaurant Process.

### 7.1 Comparison with MATLAB implementation

We compare our code and results to the MATLAB implementation of Indian Buffet Process provided by Yildirim [3]. The dataset he uses is the same as the one we have used. We got exactly similar results in terms of the features detected (Figure 5) and the reconstructed images (Figure 6). We then profiled his MATLAB code and got results as shown in Figure 7. We can see that the time taken for 1000 iterations of the sampler in MATLAB is 410 seconds which is significantly larger than the time taken by our most optimized version i.e. 285 seconds (see Table 4). Therefore, even though we have a lot of matrix calculations and MATLAB is the suited platform to run matrix intensive codes, we are able to write a much more efficient code in Python.

### 7.2 Comparison with Chinese Restaurant Process

Chinese Restaurant Process is an algorithm of customers seating in a Chinese Restaurant with infinite tables and infinite seats in each table. The customers enter one after the other and choose a table at random. In the CRP with parameter  $\alpha$ , each customer chooses an occupied table with probability proportional to the number of occupants and chooses the next vacant table with probability  $\alpha$ .

Both IBP and CRP model latent features and allow for infinite features but solve slightly different problems. IBP allows each customer to be assigned to multiple features (dishes), while CRP assigns each customer to a single feature (table). Figure 8 and 9, from Gershman and Blei (2012) [4], diagrammatically portray the difference between the two processes. Clearly, IBP solves a much wider problem in that it allows an object to have multiple features.

## Profile Summary

Generated 29-Apr-2015 19:03:55 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
<a href="#">sampler</a>	1	410.090 s	73.272 s	<div><div></div></div>
<a href="#">likelihood</a>	1365838	201.738 s	148.125 s	<div><div></div></div>
<a href="#">viaMtimes</a>	603367	63.163 s	63.163 s	<div><div></div></div>
<a href="#">trace</a>	1365838	53.613 s	53.613 s	<div><div></div></div>
<a href="#">calcInverse</a>	862838	45.357 s	45.357 s	<div><div></div></div>
<a href="#">factorial</a>	500000	24.943 s	24.943 s	<div><div></div></div>
<a href="#">num2str</a>	3001	0.680 s	0.297 s	
<a href="#">subplot</a>	12	0.484 s	0.010 s	
<a href="#">graph2d/...e/subplotHGUsingMATLABClasses</a>	12	0.473 s	0.408 s	
<a href="#">int2str</a>	2002	0.212 s	0.212 s	
<a href="#">hist</a>	1	0.196 s	0.015 s	
<a href="#">num2str&gt;handleNumericPrecision</a>	999	0.171 s	0.030 s	
<a href="#">bar</a>	1	0.161 s	0.116 s	
<a href="#">num2str&gt;convertUsingRecycledSprintf</a>	999	0.141 s	0.141 s	
<a href="#">mygamrnd</a>	1000	0.106 s	0.106 s	

Figure 7: Profiling results of matlab code for IBP

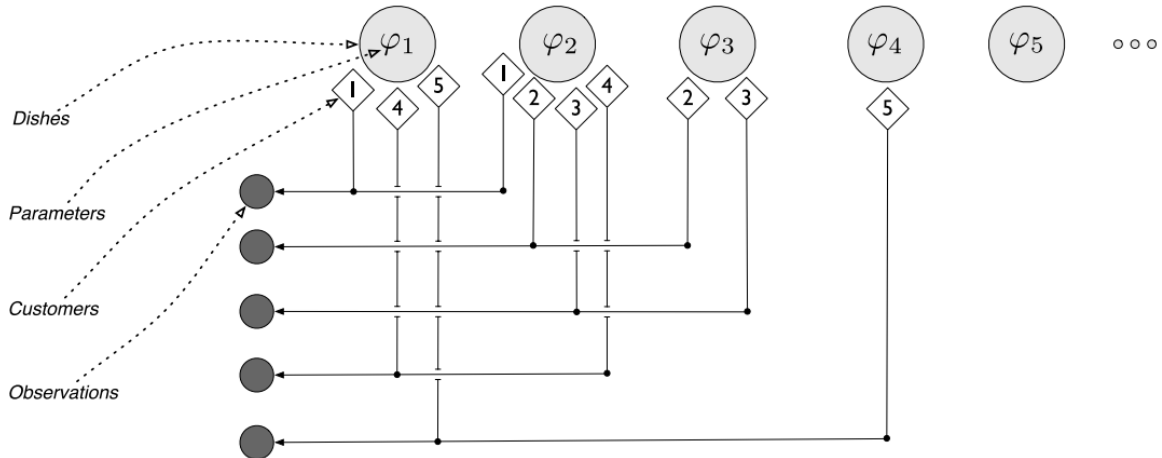


Figure 8: Chinese Restaurant Process

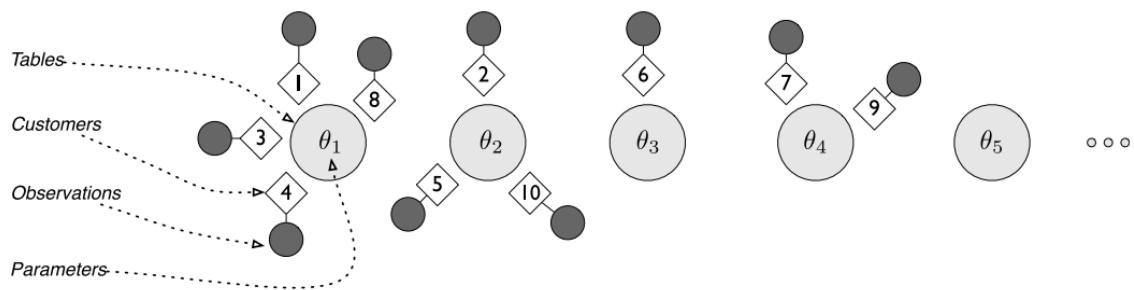


Figure 9: Indian Buffet Process

## 8 Conclusion

## References

- [1] Thomas Griffiths and Zoubin Ghahramani, *Infinite Latent Feature Models and the Indian Buffet Process*, Technical report, Gatsby Computational Neuroscience Unit, 2005.
- [2] 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.
- [3] Ilker Yildirim, *Bayesian Statistics: Indian Buffet Process*, homepage link (with sample code): <http://www.mit.edu/~ilkery/>
- [4] Samuel J Gershman and David M Blei, *A Tutorial on Bayesian Nonparametric Models*, Journal of Mathematical Psychology, 2012.