STA663 Final Project
Indian Buffet Process and its application in the Infinite Latent Feature Model

Dipesh Gautam

April 30, 2015

1 Introduction

Modeling objects is an interesting problem and one of the ways to do it is through unsupervised learning. The goal of unsupervised learning is to identify the underlying features that make up the object. Simplest way is to classify the objects into different subsets based on these features. It works well when the objects can be grouped into reasonable number of small groups [1]. When we cannot easily classify them into similar groups due to large range of features or when the objects might be better represented on their own with set of features that they have. For example, it might be more useful to describe a country as from Asia, predominantly Buddhist, tropical climate and low GDP instead of trying to classify it into some subset of countries [2]. When taking this approach, we need to consider the total features needed to represent the object. This is often treated as a model selection problem and some dimensionality is chosen based upon some measures [1]. An alternative used in nonparametric Bayesian models is to keep the dimensionality unbounded [3]. Chinese Restaurant Process uses this method and assigns objects to classes from infinite choice. Indian Buffet Process extends this and instead of restricting objects to classes, assigns a subset of infinite features to the objects.

1.1 Indian Buffet Process (IBP)

IBP is best explained in terms of customers picking dishes from an Indian buffet restaurant with infinitely many dishes arranged in a line. N customers enter a restaurant one after another. The first customer starts at the left of the buffet and takes a serving from each dish, stopping after a Poisson(α) number of dishes. The ith 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 ith customer then tries a Poisson($\frac{\alpha}{i}$) number of new dishes. Customer choice of dishes is indicated using a binary matrix Z with N rows and infinitely many columns(corresponding to the infinitely many selection of dishes), where $z_{ik} = 1$ if the ith costumer sampled kth dish [1].

The probability of Z is given by:

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

where $K_1^{(i)}$ is the number of new dishes sampled by ith customer, H_N is the N^{th} harmonic number given by $H_N = \sum_{i=1}^{N} \frac{1}{i}$.

To be able to use IBP in Gibbs sampler as in the example described in the next section, we need conditional distribution on feature assignments which given by:

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

where $z_{-i,k}$ is the assignment of feature k for all objects except the ith object, $m_{-i,k}$ is the number of objects with feature k and K is the total number of features.

Taking the infinite limit, we get the following probability in the infinite features case.

$$P(z_{ik} = 1|z_{-i,k}) = \frac{m_{-i,k}}{N} \tag{3}$$

2 Implementation

IBP can be used as a prior in models for unsupervised learning. An example of which is given in Griffiths and Ghahramani(2005) [1] and Yildirim(2012) [5], where IBP is used as a prior in infinite linear gaussian binary latent feature model.

2.1 Infinite Latent Feature Model

We used the linear-gaussian feature model as derived in Griffiths and Ghahramani(2005) [1]. We consider a binary feature ownership matrix Z which represents the presence or absence of the underlying features of the observations X. Thus each D-dimensional object x_i is generated from a Gaussian distribution:

$$x_i \sim Normal(z_i A, \Sigma_X)$$

Where A is KXD matrix of weight representing the K latent features and $\Sigma_X = \sigma_x^2 I$ is the covariance matrix that introduces white noise to the images. The weight matrix A itself has a prior with mean 0, and covariance $\sigma_A^2 I$. Thus, the likelihood is given for the data is given by:

$$P(X|Z,\sigma_X,\sigma_A) = \frac{1}{(2\pi)^{ND/2}(\sigma_X)^{(N-K)D}(\sigma_A)^{KD}(|Z^TZ + \frac{\sigma_X^2}{\sigma_A^2}I|)^{D/2}} \\ exp\{-\frac{1}{2\sigma_X^2}tr(X^T(I - Z(Z^TZ + \frac{\sigma_X^2}{\sigma_A^2}I)^{-1}Z^T)X)\} \tag{4}$$

2.2 Algorithm for Gibbs sampling and Metropolis-Hastings

2.2.1 Parameters

The parameters of interest are:

- Z: feature ownership matrix
- K_{new}: Number of new features
- α: parameter K_{new}
- σ_X
- σ_A

Full conditionals can be obtained for Z, K_{new} and $alpha$, so, we can update them using Gibbs sampling. For σ_X and σ_A, we'll use random walk MH algorithm for updating.

2.2.2 Priors for Z, K_{new} and α

Gamma prior is set for α

$$\alpha \sim Gamma(1,1) \tag{5}$$

Prior on **Z** is obtained by IBP as:

$$P(z_{ik} = 1|\mathbf{z}_{-i,k}) = \frac{n_{-i,k}}{N} \tag{6}$$

K_{new} has a poisson prior:

$$K_{new} \sim Poisson(\frac{\alpha}{N}) \tag{7}$$

2.2.3 Full conditional posteriors on Z, K_{new} and α

Full conditional posterior for Z can be directly computed using Equations (4) and (6) as given below:

$$P(z_{ik}|X, Z_{-(i,k),}, \sigma_X, \sigma_A) \propto P(X|Z, \sigma_X, \sigma_A) * P(z_{ik} = 1|\mathbf{z}_{-i,k}) \tag{8}$$

Full conditional posterior for α is given by:

$$P(\alpha|Z) \sim Gamma(1 + K_+, 1 + H_N) \tag{9}$$

Where H_N is the N^{th} harmonic number given by $H_N = \sum_{i=1}^{N} \frac{1}{i}$

To sample the number of new features for observation i, we use a truncated distribution, computing probabilities for a range of values $K_{new}^{(i)}$ up to an upper bound (say 3). Using the likelihood and prior given by Equations (4) and (7) respectively, we can easily calculate the probability distribution for K_{new} and sample the number of new dishes accordingly.

2.2.4 Metropolis-Hastings for σ_X and σ_A

For σ_X, we use random-walk MH algorithm as follows:

$$\epsilon \sim Uniform(-.05, .05) \tag{10}$$

$$\sigma_X^* = \sigma_X + \epsilon \tag{11}$$

$$\tag{12}$$

Accept this new σ_X^* with probability given by:

$$AR = min\{1, \frac{P(X|Z, \sigma_X^*, \sigma_A)}{P(X|Z, \sigma_X, \sigma_A)}\} \tag{13}$$

Where AR is the acceptance ratio.

We use similar algorithm to sample σ_A where we replace σ_X by σ_A in the algorithm described above.

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:linenofunction
   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>:2ll
    30816    0.770    0.000    1.353    0.000 linalg.py:455inv
        1    0.726    0.726   10.873   10.873 <ipython-input-145-4efe9a6e9287>:1sampler
    30816    0.542    0.000    1.071    0.000 linalg.py:1679det
    61632    0.382    0.000    0.963    0.000 numeric.py:2125identity
    61632    0.367    0.000    0.580    0.000 twodim_base.py:190eye
    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}
```

3.1 Matrix Inversion

In the likelihood function, we need to perform the matrix inversion ot $(Z^T Z + \frac{\sigma_X^2}{\sigma_A^2} I)$ every time. We saw from profiling that this was one of slower processes. A matrix inversion method has been described by Griffiths and Ghahramani(2005) [2] eq. 51-54, where the method reduces the runtime by allowing us to perform rank one updates instead of full rank updates 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: Comparision of matrix inverse methods

	Time
linalg.inverse	0.000074
calcInverse	0.000034

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. Due to time constraints, we decided to look into 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 translated into a substantial gain as shown in Table 3.

Table 2: Comparision of different likelihood functions

	Time
original ll function	0.000339
Proposed ll function	0.000305

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	538.854970
Improved ll	496.545194
Cythonized	492.187156

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 when we considered multiple burn-in periods and the loss of markov property due to multiple chains. Also, parallelizing the density calculation for likelihood wasn't useful 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* using a GPU instance from AWS 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 KXK, $100XK$ where K is smaller than 10 and $100X36$. So, the overhead was larger than the gain from gpu calculation resulting in slower time for CuBLAS matrix multiplication.

4 Unit Testing

To test the validity of the code and functions we performed the following tests.

- test that calcInverse(function for speedy calculation of inverse) gives results numerically close to np.linalg.inv
- Test that the likelihood is positive.
- Make sure that likelihood function gives error when σ_A and σ_X are non-positive
- Test for convergence of the σ_X to the true value of 0.5
- Make sure that the all the recreated objects have at least one feature as we made that assertion while simulating data

5 Application and Results

5.1 Simulated Data

We used simulated data similar to Griffiths and Ghahramani(2005) [1] and Yildirim (2012) [5]. The data set consists of $100(N)$ objects X, where each object x_i is a vector of length $36(D)$ representing the $6X6$ dimension of each object(image). These images were created using $4(K)$ latent features (base images) which correspond to the rows of the weight matrix A. Each object has .5 probability for presence of each feature. Then the object was created by adding white noise corresponding to $\sigma_X^2 = 0.5$ as:

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

The basis image and first four simulated images are shown in Fig. 3, where the top row shows the features, bottom row shows the first four objects and the middle row represents presence or absence of each of the four features for the corresponding object below.

5.2 Results

5.2.1 Validation and K_+

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 6 or even 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 plots for σ_X, σ_A, and α are shown in Fig. 2. We see that σ_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

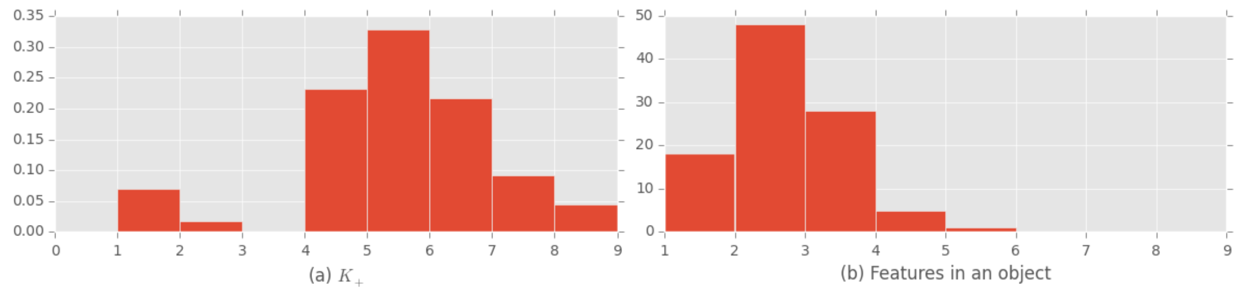
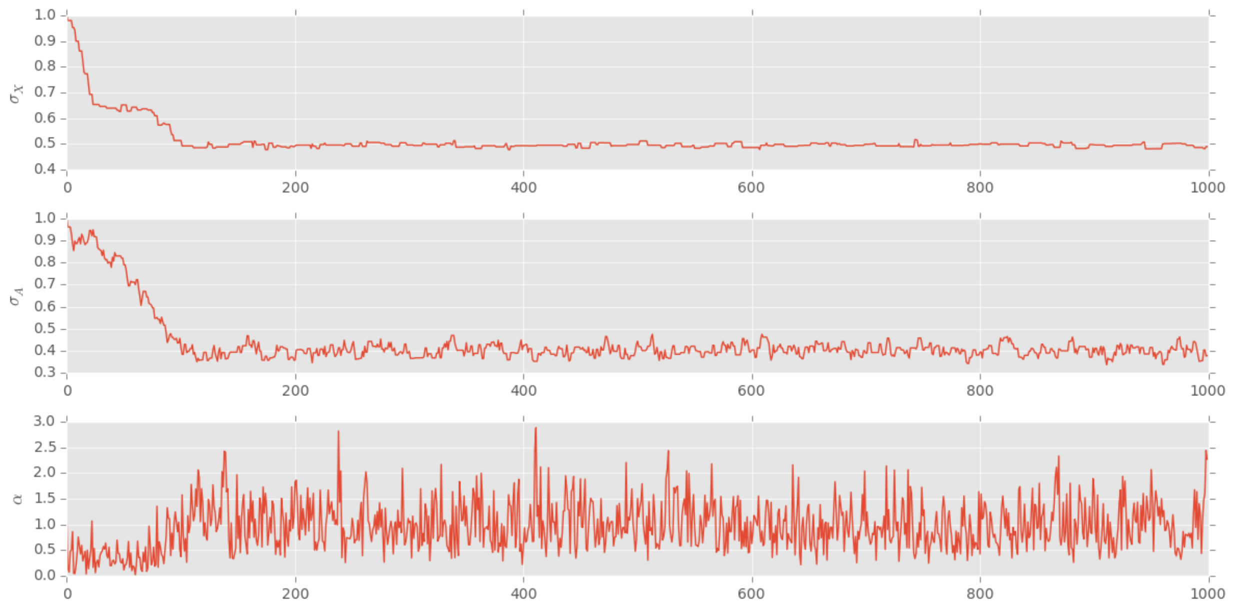


Figure 2: Traceplots for σ_X, σ_A and α



5.2.2 Latent features and recreating objects

Posterior estimation of A is given by:

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

as given in Griffiths and Ghahramani(2005) [2] eq.59. For this calculation, we used the final Z obtained from the MC with only the first four columns corresponding to the four detected features. Likewise, posterior means of σ_X and σ_A were used in the calculation of posterior estimation of weight matrix A.

With this information and the posterior Z, we were able to recreate the objects X as:

$$x_i \sim Normal(z_i A, 0)$$

We used zero variance to ignore the white noise in the recreated images. The results are shown in Fig. 4. By comparing with the original features and simulated objects as given in Fig. 3 with the detected features and recreated objects as given in Fig. 4, we can conclude that the algorithm was successfull in identifying all the latent features and successfully detecting the presense or absence of those features in the simulated objects which had white noise making detection difficult.

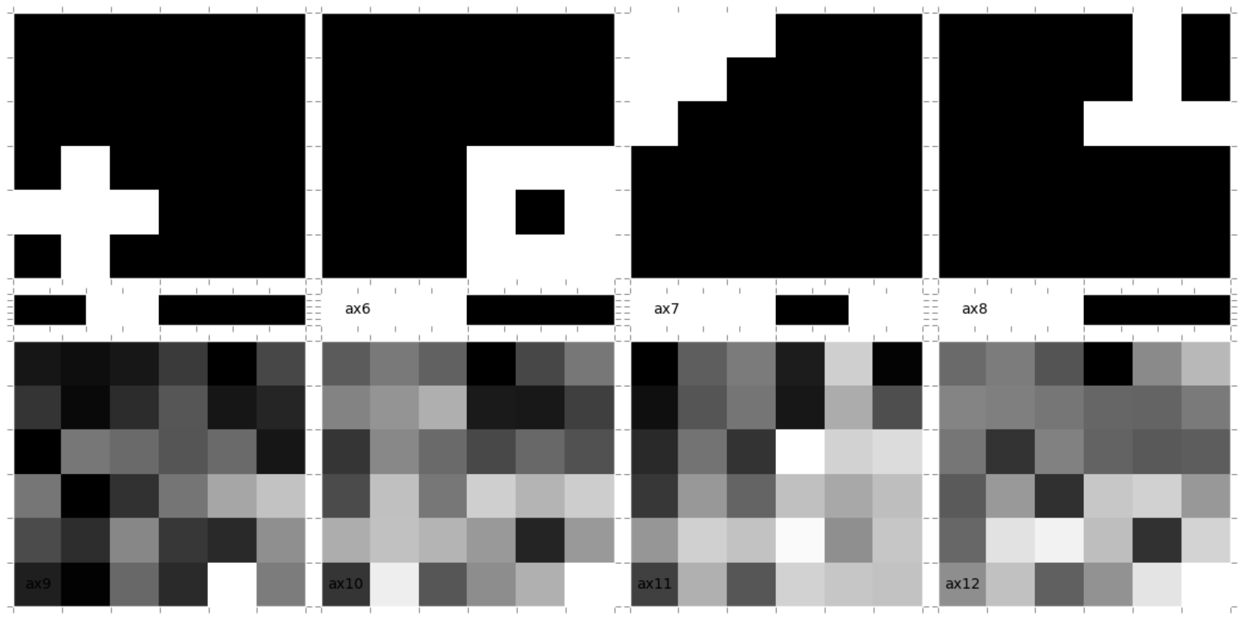


Figure 3: Original Features, First four simulated objects and the features present in each of the objects. First row shows the 4 latent features used to simulate the data. Third row shows the first four simulated objects and the middle row shows the presense or absence of the the latent features (in the order specified in the first row) in the corresponding objects below. Light signifies presence and dark signifies the absence of the feature for any given object.

6 Comparision

We compared our algorithm with an implementation of the algorithm in a different language. We also contrast IBP with similar algorithm that addresses the same problem i.e. Chinese Restaurant Process.

6.1 Comparision with MATLAB implementation of the same algorithm

We compared our code with the MATLAB code written by Yildirim [4] and provided on his website. We ran the MATLAB profiler on the code and part of the result is shown in Fig. 5. We see that we achieved similar runtime compared to the MATLAB code. Our code turned out to be slightly faster across different runs. Also, our results were consistent with those obtained from the MATLAB code we were comparing against.

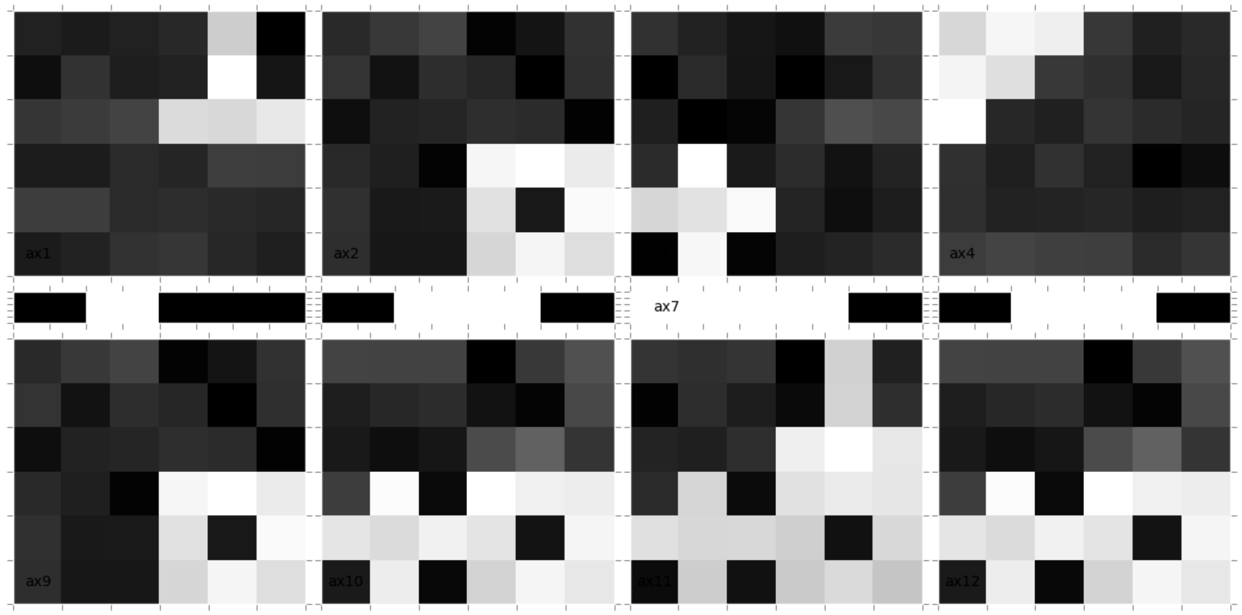


Figure 4: Detected Features, First four recreated objects and the features present in each of the objects. First row shows the 4 latent features used detected by MCMC. Third row shows the first four recreated objects and the middle row shows the presense or absence of the the latent features (in the order specified in the first row) in the corresponding objects below. Light signifies presence and dark signifies the absence of the feature for any given object.

6.2 IBP and Chinese Restaurant Process

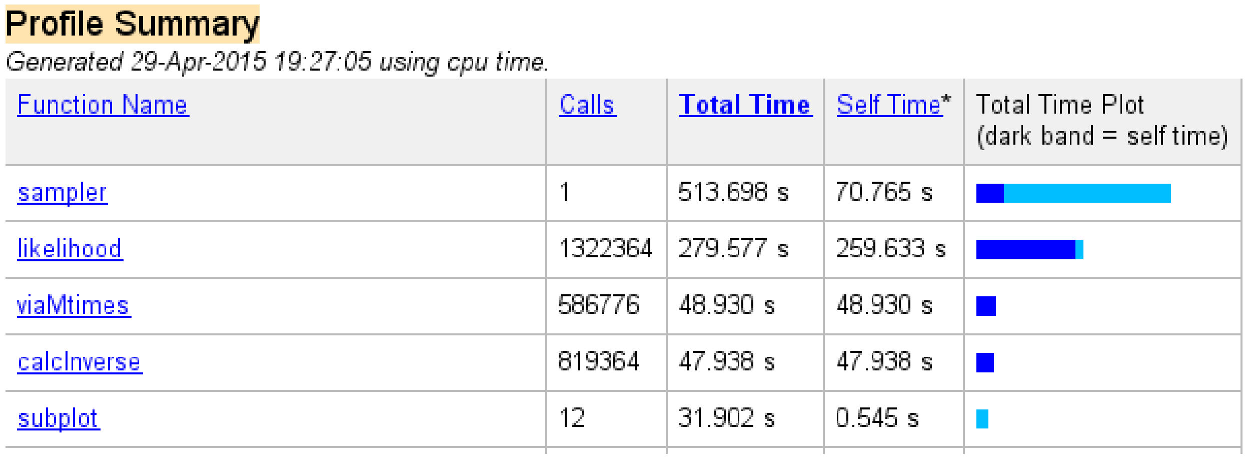
Chinese Restaurant Process (CRP) is a clustering algorithm which is based on objects previously in the cluster and some parameter α. It is explained in the setting of costumers selecting seats in a Chinese restaurant with infinite tables. The process starts with the first customer sitting on the first table. After that, each new customer either chooses an occupied table with probability proportional to number of customers already sitting on that table or chooses an empty table with a probability proportional to the parameter α. The process continues until every customer has sit down on a table. This allows for clustering of infinite number of objects into infinite classes.

As explained earlier, in IBP instead of assigning a customer to a cluster, they're assigned latent features(dishes). IBP allows for flexibility as the objects are not restricted to some class with other objects but are described by their own features. We can think of IBP as feature allocation problem whereas CRP is clustering problem. Even though both these processes solve similar problem of classifying objects, the flexibility in IBP makes it a better algorithm in many cases where objects cannot be partitioned into relatively homogeneous subsets, whereas CRP is favored when they can be partitioned into homogeneous subsets [1]. As an example, if we were clustering the objects from the Infinite latent features model described above, in an ideal situation we'd have 2^K clusters and this grows exponentially with increasing features and gets out of had really quickly.

7 Conclusions

We were able to implement the Indian Buffet Process and use it as a prior in the Infinite linear-Gaussian binary feature model as was shown by Griffiths and Ghahramani [1] and Yildirim [5]. Using the synthetic data, we were able to correctly predict the latent features in the simulated data as well as detect their presense in the objects accurately. For optimization of the code, we profiled the code and found some bottlenecks and to address the issues we removed redundant calculations, cythonized the code and looked at implementation of CuBLAS library from CUDA for some linear algebra operations. Some of these steps were successful and

Figure 5: Result of MATLAB profiler


Profile Summary

Generated 29-Apr-2015 19:27:05 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
sampler	1	513.698 s	70.765 s	
likelihood	1322364	279.577 s	259.633 s	
viaMtimes	586776	48.930 s	48.930 s	
calcInverse	819364	47.938 s	47.938 s	
subplot	12	31.902 s	0.545 s	

some were not. Another promising thing for code optimization was the rank one update matrix inversion technique described in by Griffiths and Ghahramani [2]. Although we were able to implement the matrix inversion and test the validity, we weren't able to incorporate it in the MCMC algorithm. More work can be done in the future to incorporate the method which will boost up the performance of the algorithm.

References

[1] Thomas Griffiths and Zoubin Ghahramani. Infinite latent feature models and the indian buffet process. 2005.

[2] Thomas Griffiths and Zoubin Ghahramani. Infinite latent feature models and the indian buffet process. technical report 2005-001. 2005.

[3] Carl Edward Rasmussen and Zoubin Ghahramani. Occam's razor. *Advances in neural information processing systems*, pages 294–300, 2001.

[4] Ilker Yildirim. http://www.mit.edu/ ilkery/.

[5] Ilker Yildirim. Bayesian statistics: Indian buffet process. 2012.