STA663 Final Report: Indian Buffet Process

Raghav Agrawal, Yiqian 'Peyton' Chen

4/26/2021

Github Link

https://github.com/peytonychen7/STA-663-Final-Project

Abstract

The dimensionality of representation is a critical question to solve in unsupervised learning when we try to represent objects with multiple latent features. One method to solve this question is to apply a Bayesian latent feature model where the prior is used to represent the number of latent features. The Indian buffet process is an efficient stochastic process that can generate a prior for a model to represent objects with an infinite number of latent features. In this paper, we implement the basic algorithm of a binary latent-feature model where the prior is generated by the Indian buffet process. We explore ways to optimize the algorithm and test the algorithm using simulated data sets and real data sets. We also compare the algorithm with two competing algorithms.

1 Introduction

Unsupervised learning techniques have gained attention in academia and the industry due to their application in a variety of fields, such as image and pattern recognition, cancer research, consumer research, etc (Bouguila et al, 2005; ISLR, 2013). These techniques can help us identify the properties of objects even in the absence of response variables. The properties of these objects can be better captured if we use multiple latent features to represent the objects (Griffiths & Ghahramani, 2005). As we work with latent features, one question we often need to consider is how many latent features do we need to use to represent the objects. The number of latent features can be finite or infinite. If we assume the dimension is infinite, one way to determine the latent structure is to use a Bayesian latent feature model, in which we use a prior distribution to represent the number of latent features needed, and use the likelihood function to analyze how these latent features are associated with the objects in the data set (Griffiths & Ghahramani, 2005). The Indian buffet process provides an efficient way to generate a prior distribution for a latent feature model over equivalence classes of binary matrices with a finite number of rows but potentially an infinite number of columns. Such a model can be used to represent each object with a large number of latent features.

One of the applications of the Indian buffet process is image processing. For example, Dang and Chainais applied the Indian buffet process as a prior and proposed a Bayesian nonparametric approach called Indian buffet process dictionary learning. The algorithm was applied to image inpainting and compressive sensing (Dang & Chainais, 2017).

In this paper, we implement the algorithm of the Indian buffet process described in the paper titled "Infinite Latent Feature Models and the Indian Buffet Process" by Thomas L. Griffiths and Zoubin Ghahramani. We use the Indian buffet process to generate a prior for a linear-Gaussian binary latent feature model described

in the paper, and implemented a Gibbs sampler to find the most frequent latent features. We explore ways to make the algorithm more efficient using computational techniques in Python and test the algorithm using simulated data and real-world data. We also compare the efficiency of our algorithm with two competing algorithms.

2 Description of Algorithm

2.1 Indian Buffet Process

We can use the Indian buffet process to generate a binary matrix $\mathbf{Z}$ that shows which latent features are associated with each of the objects. If kth feature is associated with the ith object, then we will set $z_{ik} = 1$; otherwise, we will set $z_{ik} = 0$. Suppose we have N objects, then the binary matrix $\mathbf{Z}$ would have a dimension of $N \times D$, where D is the number of the features that we would have at the end of the process.

The Indian buffet process draws an analogy with Indian buffet restaurants in London. The restaurants have a large number of dishes for customers to choose from. Suppose we have N customers entering the restaurant one after another. The first customer takes the first Possison(α) dishes. The dishes that were taken by this customer are "recorded" in the first row of the matrix $\mathbf{Z}$ by setting $z_{1k} = 1$ where k represents the location of the dishes taken by this customer. All dishes that are not taken by this customer remain as 0 in $\mathbf{Z}$. Then for each of the customers followed, the ith customer considers all the dishes that have been taken by all previous customers based on popularity, and takes each dish with a probability of $\frac{m_k}{i}$ where m_k is the number of customers who have tried the dish. Once this customer has considered all the dishes that have been tried by previous customers, s/he tries Poisson(α/i) dishes that have not been tried by any of the previous customers. All the dishes that are taken by each customer are "recorded" in $\mathbf{Z}$ in the same manner.

The probability of getting a matrix $\mathbf{Z}$ is as follows:

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

where K^+ represents the number of dishes that have been tried by at least one customer (i.e. $m_k > 0$), and $K_1^{(i)}$ represents the number of new dishes taken by the ith customer.

Once we know the distribution of $\mathbf{Z}$, we can define the full condtional for $z_{ik=1}$ in $\mathbf{Z}$ as $P(z_{ik} = 1 \mid \mathbf{Z}_{-ik})$. But we only need to condition on the elements in the same column $\mathbf{z}_{-i,k}$ since columns are independent. Therefore, we have the full conditional for z_{ik} as follows:

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

2.2 Applying the Indian buffet process to a linear-Gaussian binary latent feature model

We use the linear-Gaussian binary latent feature model derived in the paper, where the likelihood function is as follows:

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

To avoid potential underflow/overflow during the computation, we compute the log-likelihood function first and exponentiate it to get the likelihood. The log-likelihood function has the following form:

$$l(\mathbf{X} \mid \mathbf{Z}, \sigma_X, \sigma_A) = -\{\frac{ND}{2}log(2\pi) + \frac{N-K}{D}log(\sigma_X) + KDlog(\sigma_A) + \frac{D}{2}log(|\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}I|)\}$$
$$- \frac{1}{2\sigma_X^2}tr(\mathbf{X}^T(\mathbf{I} - \mathbf{Z}(\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}I)^{-1}\mathbf{Z}^T)\mathbf{X})$$

With the likelihood function, we can find the full conditional of z_{ik} by multiplying the likelihood function by the prior mentioned earlier:

$$P(z_{ik} \mid \mathbf{X}, \mathbf{Z}_{-ik}, \sigma_X, \sigma_A) \propto P(\mathbf{X} \mid \mathbf{Z}, \sigma_X, \sigma_A)P(z_{ik} \mid \mathbf{z}_{-i,k})$$

With each iteration, we will be checking to see if any new columns (K_{new}) may be added to the Z matrix. The K_{new} distribution looks as so.

$$\text{Prior} : P(K_{new} \mid \alpha) \propto Poisson(\frac{\alpha}{N})$$
$$\text{Posterior} : P(K_{new} \mid \mathbf{X}, \mathbf{Z}, \sigma_X, \sigma_A, \alpha) \propto P(K_{new} \mid \alpha)P(\mathbf{X} \mid \mathbf{Z^*}, \sigma_X, \sigma_A)$$
$$\mathbf{Z^*} = Z \text{ with } K_{new} \text{ columns appended}$$

We will also be considering the distributions of σ_X, σ_A, and α

For σ_X, σ_A we will be utilzing the Metropolis Hastings Algorithm by comparing values of $P(\mathbf{X} \mid \mathbf{Z}, \sigma_X, \sigma_A)$ under our current values of σ_X, σ_A against new values of σ_X, σ_A to see if they should be updated.

The Distribution of α is as so:

$$\text{Prior} : P(\alpha) \propto Gamma(a, b)$$
$$\text{Posterior} : P(\alpha \mid \mathbf{Z}) \propto Gamma(a + K, b + H_n)$$
where a and b are hyperparamaters

2.3 Gibbs Sampler Steps

1. Set X, number of iterations, number of potential new columns per iteration, inital α, α priors, ϵ, intial σ_X, σ_A

2. Initialize Z with Indian buffett process

3. For each iteration until max iterations:

a) For each Z_{ik}, sample to see if it is 0 or 1 using the fully conditional probability distribution laid out above

b) Sample to see if more columns should be added (increase k) or potentially be removed using the fully conditional probability distribution laid out above

c) Use MH and update σ_X as neccesary

d) Use MH and update σ_A as neccesary

e) Sample α from its posterior gamma distribution

3 Optimization for Performance

In this section, we explore a few optimization methods to make the sampler more efficient. The test results can be found in section 3.4. The testing detail can be founded in the `Optimization.ipynb` in the GitHub repository.

To test the performance, we used the example images used in Ilker Yildirim's paper as my simulation data set (Yildirim 2012). The detail of the numerical example can be found in Section 4.

3.1 Profile

We adopt the decorator from a post on Medium written by Farhad Malik (Malik 2020) to profile our original python code.

We generated an initial profile in Figure 1. From the profile, we can see that most of the runtime was spent on the `log_likelyhood` function. Therefore, we concentrated our energy to explore how to optimize the performance of our `log_likelyhood` function.

```
Mon Apr 26 14:10:50 2021    C:\Users\peyto\AppData\Local\Temp\tmp4az4i185

         60448050 function calls (59847646 primitive calls) in 176.500 seconds

   Ordered by: internal time
   List reduced from 65 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   979552   77.271    0.000  140.856    0.000 IBP_Sampler.py:43(log_likelyhood)
        1   15.018   15.018  176.500  176.500 <ipython-input-5-699e279c7ba3>:1(sampler)
   979552    8.610    0.000   17.410    0.000 linalg.py:482(inv)
   388480    7.793    0.000    7.793    0.000 {built-in method builtins.sum}
   979552    7.719    0.000   15.931    0.000 linalg.py:2063(det)
  1959104    7.677    0.000   13.886    0.000 twodim_base.py:154(eye)
   979552    7.031    0.000    7.031    0.000 {method 'trace' of 'numpy.ndarray' objects}
  2359311    6.798    0.000    6.798    0.000 {built-in method numpy.zeros}
4139726/3539322    4.722    0.000   53.791    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
  1959104    3.766    0.000    6.455    0.000 linalg.py:144(_commonType)
```

Figure 1: Profile of the Original Sampler

3.2 Optimization of Matrix Calculations

First, we explored ways to optimize the `log_likelyhood` function by calculating the determinant and inverse of matrix $\mathbf{M}$ more efficiently.

We define a `sampler_test` function that allows us to use different log likelyhood functions as input. We test each method using the same inputs with `niter = 100`.

3.2.1 Replacing `inv` with `solve`

In this approach, we consider replacing the calculation of inverse matrix with linear solve. Our intuition is linear solve may have a smaller complexity than calculating the inverse of matrix $\mathbf{M}$. We have to calculate $\mathbf{Z}\mathbf{M}^{-1}\mathbf{Z}^T$. So instead of calculating `Z @ np.linalg.inv(M) @ Z.T`, we tried `Z @ np.linalg.solve(M, Z.T)` in this case.

However, we found the performance is comparable. Using `np.linalg.solve` may not have a noticeable improvement for the function.

3.2.2 Singular Value Decomposition

We also explored a way of computing the determinant and the inverse matrix using singular value decomposition. Since $\mathbf{Z}^T\mathbf{Z} + \frac{\sigma_X^2}{\sigma_A^2}\mathbf{I}$ is a nonsingular matrix, we can calculate the determinant by calculating the products of the singular values s. Once we have U and V^T from the singular value decomposition, we can also calculate the inverse of matrix $\mathbf{M}$ as $VD^{-1}U^T$ where D^{-1} is a diagonal matrix with diagonal elements $1/s$.

But the performance is comparable to the original likelyhood function. It may be a bit slower than the original function. This may be due to the considerable amount of computational time required by finding the singular value decomposition.

3.2.3 Using functools

We also tried to use the `reduce` function from `functools` to see if we can improve the performance. Once again, we did not find any noticeable improvement in speed. Therefore, we decide to explore optimization methods using cython and numba with the original `log_likelyhood` function.

3.3 Cython and Numba

Next, we tried to optimize the `log_likelyhood` function using cython.

3.3.1 Comparing the performance of matrix multiplication

First, we compared the performance for matrix multiplication between the `@` operator in numpy and the `matrix_multiply` function we wrote in cython.

We randomly generated a matrix A_1 with size of 1000×36 and a matrix with A_2 with size of 36×1000 to compare the performance of matrix multiplication. We found that our `matrix_multiply` function written in cython is much slower than the `@` operator in numpy. Since matrix multiplications take a considerable amount of time in the `log_likelyhood` function, we decided not to use `matrix_multiply` function but wrote a cythonized `log_likelyhood` function using the `matmul` function in numpy.

3.3.2 Cythonize the `log_likelyhood` function

Then we tried to cynthonize the `log_likelyhood` function. However, it does not outperfrom the original likelyhood function. One of the reasons can be our cython code still depends heavily on functions in the numpy package to compute the determinant and the inverse function. Using numpy functions in cython may create unnecessary overhead. Due to the complex data structures in cython, we chose to stick with the original `likelyhood` function for now.

3.3.3 Numba

We also tried to use numba for matrix multiplication. But we found that the speed of matrix multiplication is much slower than using `@` operator in numpy. We can conclude that numba is not an optimal choice for our algorithm.

3.4 Conclusion on Optimization

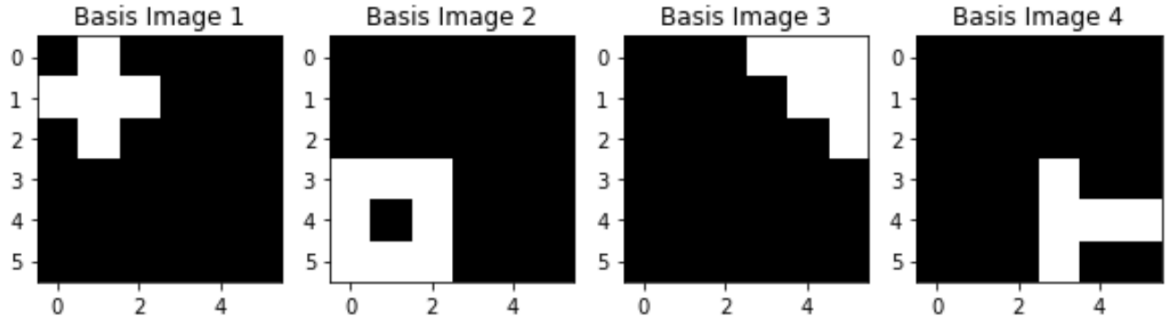
The table below shows a brief summary of the average speed of different methods when we tested them with `niter = 100`.

	Original	Linear_solve	SVD	Reduce	Cython
Speed (in seconds)	11.7	12.3	14.4	12.3	13.1

Based on the results, we did not find any method that provides a significant improvement for the speed of the algorithm. Therefore, we decided to use the original function written with numpy as final algorithm.

4 Applications to Simulated Data Sets

For this simulation, we followed the procedures laid out in Ilker Yildrim's paper. To begin, we created four basis images.

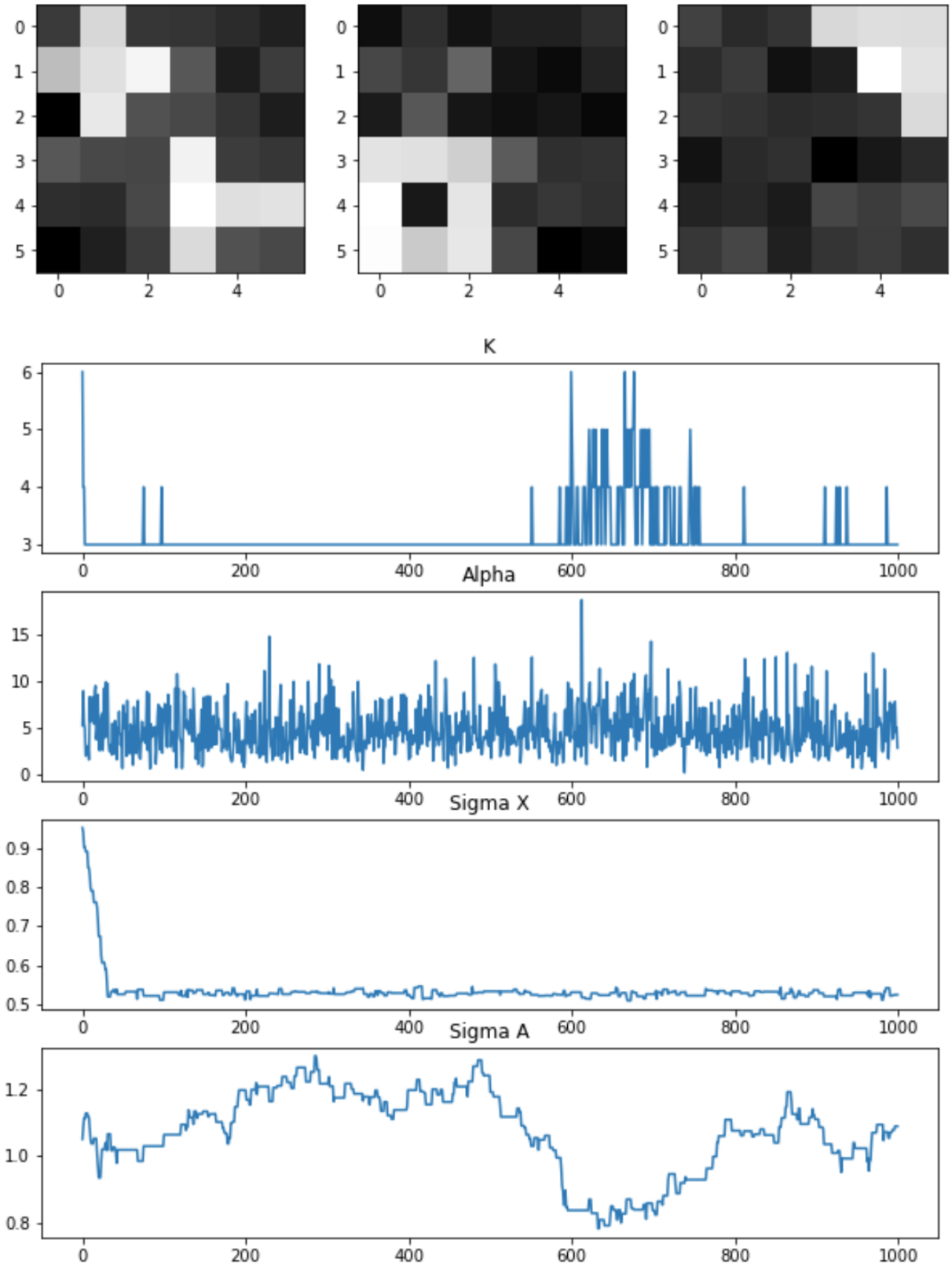


As we can see, each basis image has a different section and shape of the 6 by 6 grid. We used these basis images to construct our X with some noise added in ($\sigma_X = .5$). For this simulation, we ran it with 2 different seeds to see if/how the results would change as the code is quite dependent on random number generation. Both simulations ran for 1000 runs and had the same initial parameter settings.

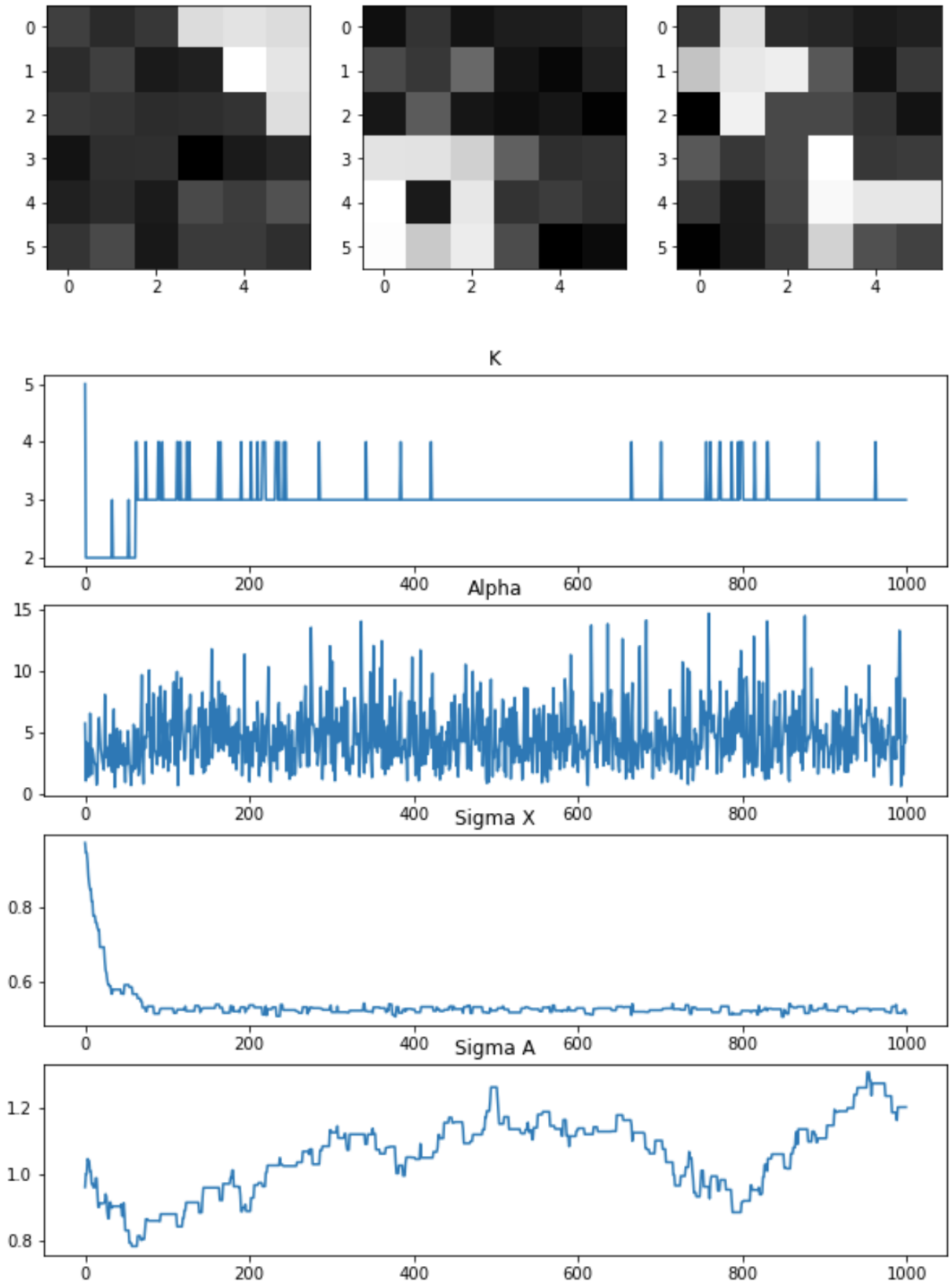
Intital Paramater Settings
-$\alpha = 1$
-$\epsilon = .05$
-α_a prior $= 1$
-α_b prior $= 1$
-$\sigma_X = 1$
-$\sigma_A = 1$
-max new ks (per iteration) $= 4$

Upon finishing the Gibbs sample, we found the Expected feature matrix which is calculated as so
E [A | Z, X] $= (Z^T Z + \sigma_x^2/\sigma_a^2 * I)^{-1} Z^T X$

Simulation 1



Simulation 2

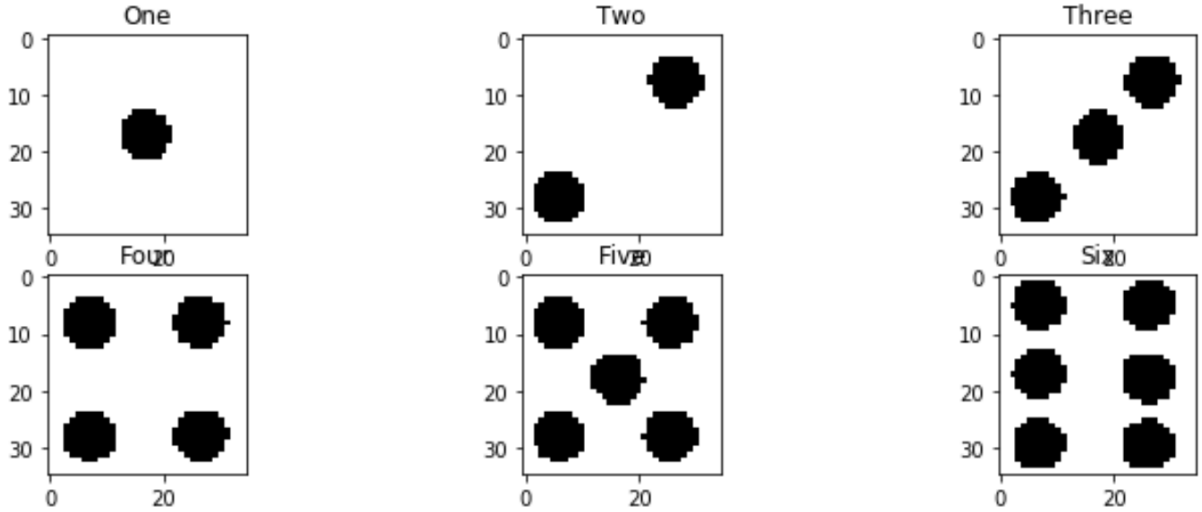


Both simulations returned very similar latent features not only to each other but also the original basis

images. One difference is that the latent features seemed to have combined two of the original basis images together but we are still getting the same features as what was originally coded in. The first simulation had a greater fluctuation in the value of k as it constantly moved between 3 and 6 and finally settled at 3 whereas the second simulation seemed to oscillate mainly between 3 and 4 with less frequent movement. Based on the latent images we have shown at the end, it is reasonable to assume that values with different ks are returning the same features just in a different form. In both, the σ_X also converged around the true value of .5.

5 Applications to Real Data Sets

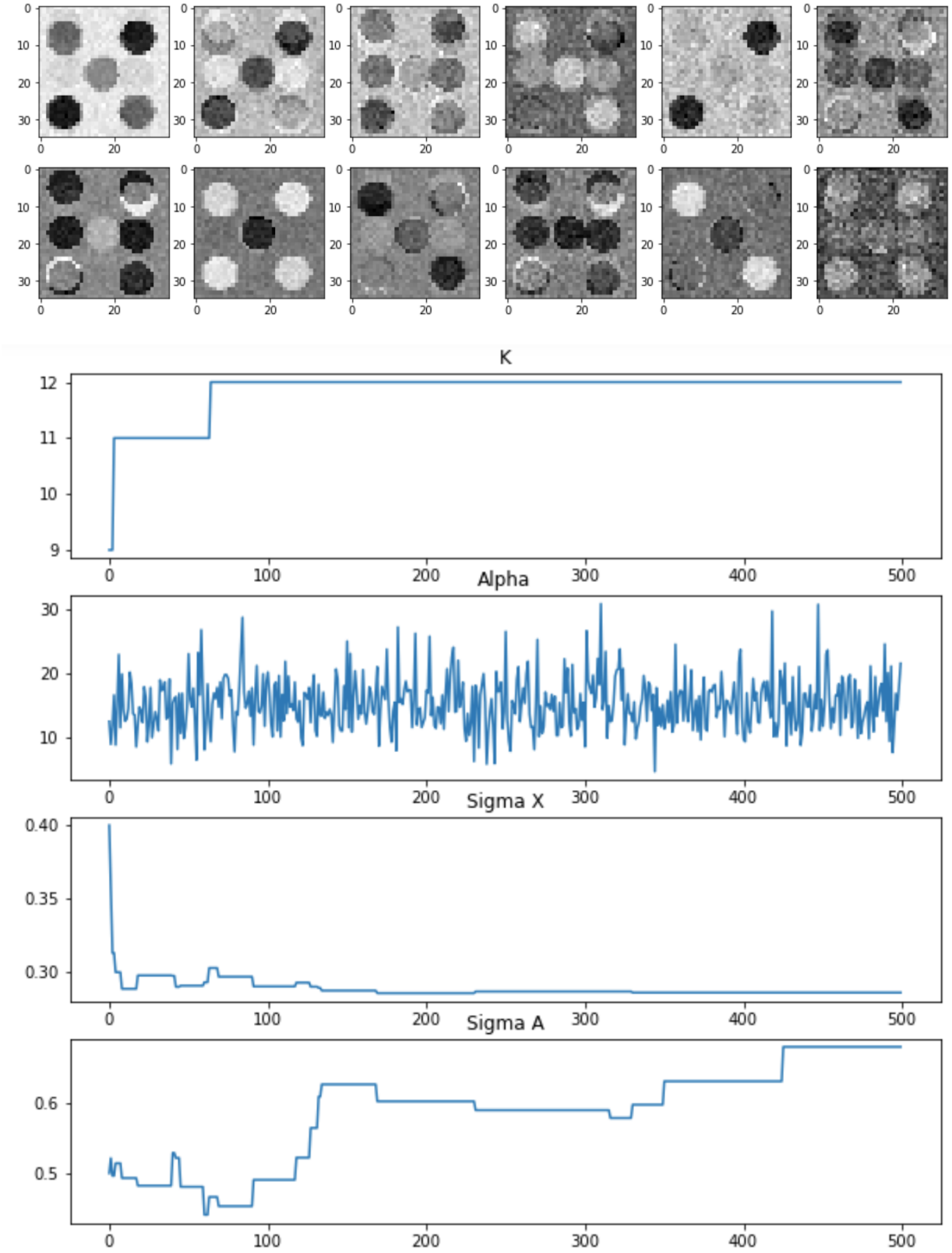
In addition to the simulation, we also ran our algorithm against a real world dataset to gain deeper insights into the application of IBP. For this real world example, we choose 6 images of a die with each image corresponding to a number on a die.



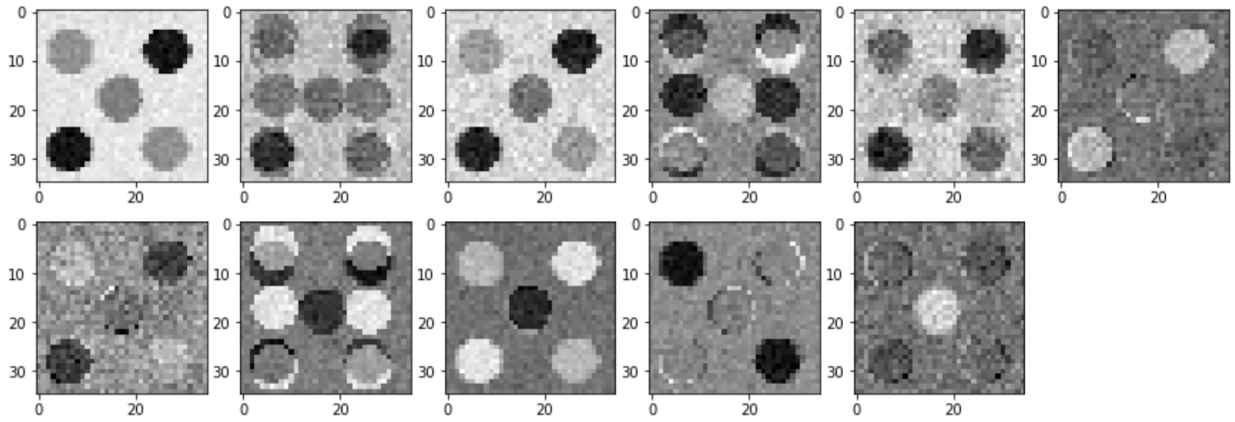
Once again, we ran two simulations with different seeds to help control for some of the randomness that occurs with this process. This time each run was for 500 iterations.

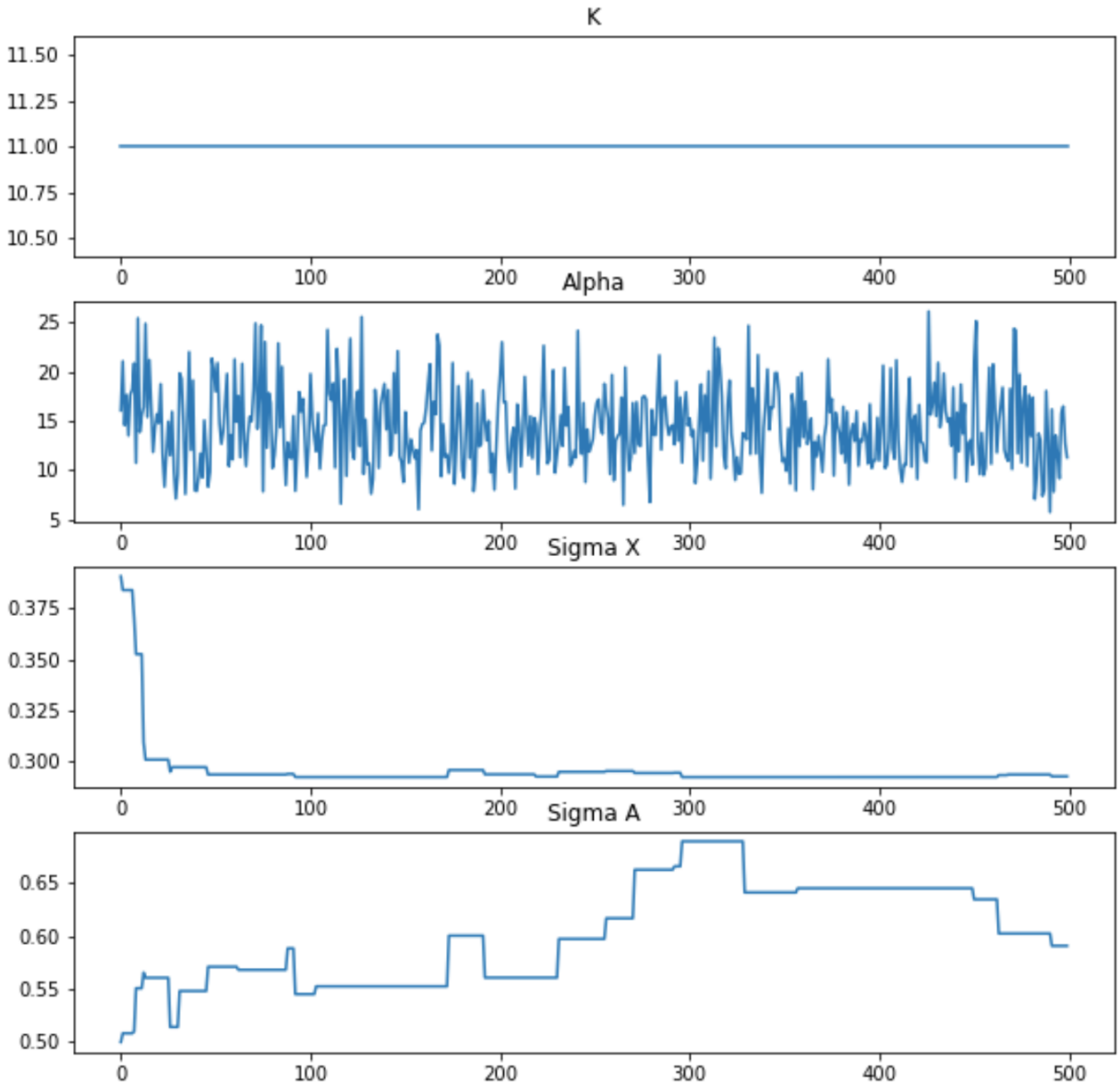
Intital Paramater Settings
-$\alpha = 1$
-$\epsilon = .05$
-α_a prior $= 1$
-α_b prior $= 1$
-$\sigma_X = .4$
-$\sigma_A = .5$
-max new ks (per iteration) $= 3$

Sample 1



Sample 2





The results from the 2 runs appear quite similar to each other with some minor differences. In the first run, 12 latent features are discovered instead of 11 like in the 2nd run. The first simulation most commonly

returns a pattern with 7 dots each lined up as to contain all possible numbers while the second simulation most commonly picks up a pattern like that in the number 5. However, both still do a good job at capturing the patterns of the dice which is highly encouraging. There is much less movement in K compared to the simulation example with the 2nd run not changing from its original value of 11. Both σ_X also are converging towards the true value of .25.

6 Comparative Analysis

6.1 Comparing Indian Buffet Process and Chinese Restaurant Process

The Chinese Restaurant Process (CRP) is another Bayesian method used to discover latent features and is often compared to the Indian Buffet Process. In the CRP, there are infinite customers and infinite tables. The first customer sits down at the first table. The next customer can either sit down at the same table or choose an unoccupied table. This then repeats for all of the customers where the probability of sitting at an already occupied table is proportional to the number of people sitting at the table and the probability of sitting at an unoccupied table is α.

While both involve finding latent features they do have some pretty stark differences. The Chinese Restaurant Process can be thought of as a traditional clustering algorithm where objects (customers) are put into a class (table). The Indian Buffet process, however, allocates features where each object (customer) takes on certain features (dishes). This allows for more flexibility and therefore may be more favorable of the two especially when the features get more complicated and it's harder to partition the data into reasonable sections.

6.2 Comparing with an MATLAB Code Online

In section 3 & 4, we used the example images used in Ilker Yildirim's paper as my simulation data set to test the performance and accuracy of our algorithm. In this section, we compare the speed of our algorithm with the written in MATLAB in this paper (Yildirim 2012). The sample code can be found in Ilker Yildirim's homepage in MIT (http://www.mit.edu/~ilkery/). We continue to use the same example images and ran the sampler with the same inputs for `niter=1000`. We profiled the the MATLAB code and found the total runtime to be about 255 seconds (Figure 2). This appears to be slower than our algorithm in Python. The simulation result (Figure 3) is similar to our result in section 4.

Profile Summary (Total time: 255.541 s)

▾ **Flame Graph**

Flame graph is not available because the function call history size of 5000000 was too small. Rerun the Profiler with a larger history size. For more information, see profile.

Generated 26-Apr-2021 14:17:09 using performance time.

Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (dark band = self time)
sampler	1	255.474	44.389	
likelihood	1322364	167.689	155.006	
mpower	586776	23.169	5.952	
mpower>integerMpower	586776	17.217	17.217	
trace	1322364	12.684	12.684	
calcInverse	819364	11.984	11.984	
factorial	500000	6.921	6.921	
subplot	12	0.432	0.208	
num2str	3001	0.341	0.079	
hist	1	0.171	0.037	
axis	12	0.144	0.117	
int2str	2002	0.131	0.131	

Figure 2: Profile of the Original Sampler

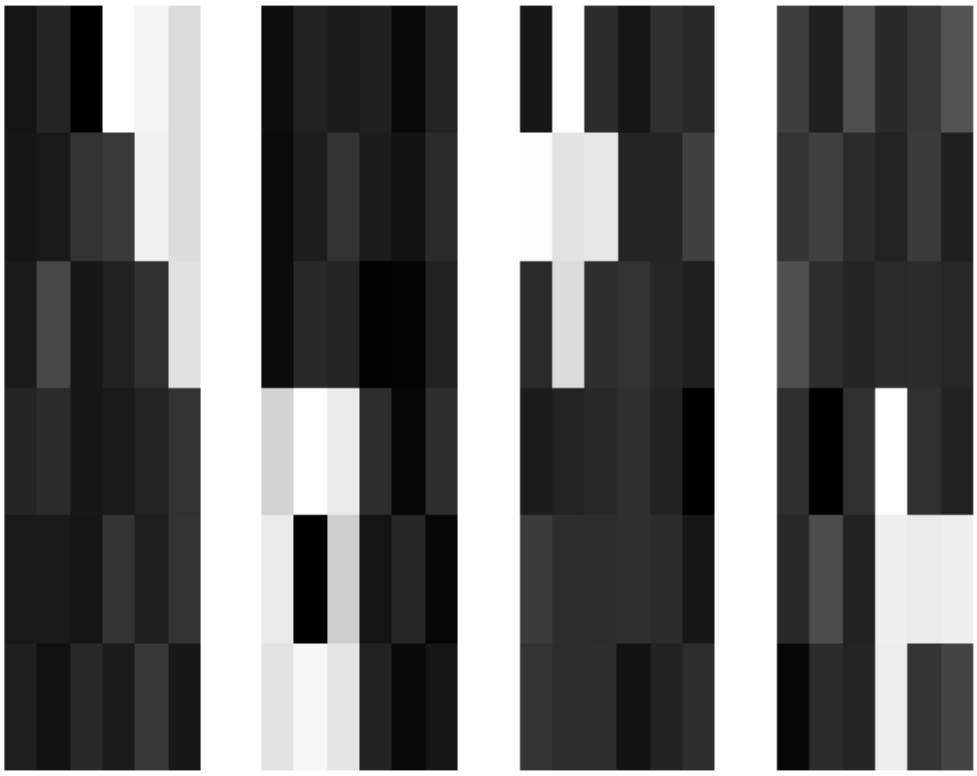


Figure 3: Simulation Results Using MATLAB

7 Conclusion

The Indian Buffet Process is a fairly popular tool used in Bayesian Unsupervised Learning. Through this paper, we have succesfully implemented the algorithm mentioned in the paper by Griffiths and Ghahraman and tested it in a simulated setting and a real world example. In addition, we did tests to discover that our algorithm was the most efficent and even compared our process to other popular techniques like the Chinese Restaurant Proces and other implementations of the Indian Buffet Process done in Matlab.

While the Indian Buffet Process is a powerful tool that can be used to discover latent variables, it does have some drawbacks. As much of the sampling is using random number generators like "np.random.random()" and "np.random.poisson" it can be dependent on what random numbers are being generated. Although our results were fairly consistent, more testing may need to be considered. In addition, the Gibbs Samplers require inputs of priors that may be difficult to have an initial inclination of their value.

For more information about the code, diagrams, or package installation regarding the material in this paper, please visit https://github.com/peytonychen7/STA-663-Final-Project.

8 Contributions

Raghav Agrawal, MSS Class of 2022 at Duke Iniversity

- Developed the inital algorithm
- Performed tests on Simulated Data as well as Dice Data
- Comparative Analysis to Chinese Buffet Process and Conclusion

Peyton Chen, MSS Class of 2022 at Duke Iniversity

- Performed Optimation tests to ensure algorithm was efficent
- Established Package Installation from PyPi
- Comparative Analysis to MatLab example and Introduction

9 References

Bouguila, Nizar, and Djemel Ziou. "Using Unsupervised Learning of a Finite Dirichlet Mixture Model to Improve Pattern Recognition Applications." Pattern Recognition Letters, vol. 26, no. 12, 2005, pp. 1916–1925., doi:10.1016/j.patrec.2005.03.016.

"Digital Math Resources." Media4Math, www.media4math.com/math-clip-art?field_la_display_title_value=%22Math+Clip
Dice+and+Number+Models%22.

Dang, Hong-Phuong, and Pierre Chainais. "Indian Buffet Process Dictionary Learning: Algorithms and Applications to Image Processing." International Journal of Approximate Reasoning, vol. 83, 2017, pp. 1–20., doi:10.1016/j.ijar.2016.12.010.

Griffiths, Thomas L., and Zoubin Ghahramani. "Infinite Latent Feature Models and the Indian Buffet Process." 2005.

An Introduction to Statistical Learning (ISLR): with Applications in R, by Gareth James et al., Springer, 2013, pp. 373–374.

Malik, Farhad. "Advanced Python: Learn How To Profile Python Code." Medium, FinTechExplained, 20 July 2020, medium.com/fintechexplained/advanced-python-learn-how-to-profile-python-code-1068055460f9.

Yildirim, Ilker. "Bayesian Statistics: Indian Buffet Process." August 2012, https://www2.bcs.rochester.edu/sites/jacobslab/cheat_sheet/IndianBuffetProcess.pdf