Infinite Latent Features and Indian Buffet Process

Wenwen Ye, Kuazhuo Zhang

March 20, 2019

1 Introduction

In the area of unsupervised learning, recognizing features in images has received a lot of attention. In traditional studies of unsupervised learning, simple methods like clustering are vastly used. However, those methods require users to specify the number of features(clusters) to classified the data into. Other more advanced learning methods such as neural networks can take infinite latent features, but are difficult to apply and compute.

The paper from Griffiths and Ghahramani(2005)[1] proposed an interesting way of identifying latent features in the image by Indian Buffet Process, without the need to limit the number of features in advance. We will introduce their method, rebuild the algorithms, test them on simulated data, and optimize the algorithms.

2 Methods

The methods introduced by Griffiths and Ghahramani(2005)[1] are mainly consist of two parts: Indian Buffet Process, which is used to assign features to each object, and Gibbs sampling, which is used to sample hyperparameters of data.

2.1 Indian Buffet Process

Consider the case when we observe D features of N objects(images), which are stored in a $N \times D$ matrix X. Each object can have infinite amount of latent features. The latent features of X are represented by a $N \times K$ binary matrix Z, where K is the total number of all latent features and can be taken to infinity.

For the i^{th} object, suppose the current number of latent features that have been already identified is K, then the probability for i^{th} object to have those latent features is:

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

In the infinite model, take K to infinity:

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

where $m_{-i,k}$ is the number of objects that have k^{th} feature. After sampling for every already-existed latent feature, i^{th} object will then have $K^{(i)}$ additional features, where $K^{(i)}$ follows a $Poisson(\frac{\alpha}{N})$ distribution.

2.2 Linear-Gaussian Binary Latent Feature Model

2.2.1 Likelihood

In a linear-Gaussian binary latent feature model, we assume $\mathbf{X}$ follows a matrix Gaussian distribution: $\mathbf{X} \sim MN_{N,D}(\mathbf{ZA}, \sigma_X^2\mathbf{I})$, where $\mathbf{A}$ is a $K \times D$ weight matrix that has a Gaussian prior: $\mathbf{A} \sim N(0, \sigma_A^2\mathbf{I})$

Then the likelihood after $\mathbf{A}$ is integrated out will be:

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

2.2.2 Priors, posteriors and full conditionals

Given the likelihood above, the prior, posterior or full conditional distributions for $\mathbf{Z}, \sigma_A, \sigma_X$ and α is set as followed:

	prior	full condition/posterior for MH
α	Gamma(1,1)	Gamma(1+K,1+Hn)
σ_X	InvGamma(2,1/2)	Likelihood$\times$InvGamma(2,1/2)
σ_A	InvGamma(2,1/2)	Likelihood$\times$InvGamma(2,1/2)
$K^{(i)}$	Poisson($\frac{\alpha}{N}$)	Likelihood$\times$Poisson($\frac{\alpha}{N}$)
$\mathbf{Z}$	IBP	Likelihood$\times p(z_{ik} = 1\|\mathbf{z}_{-i,k})$

2.3 Gibbs Sampling Algorithm

The complete Gibbs sampling algorithm is therefore as followed:

1. Sample α from its prior Gamma(1,1)

2. Sample z_{ik}, where $i = 1, \ldots, N$ and $k = 1, \ldots, K$:

   i. Compute the full conditional probability for $z_{ik} = 1$ as

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

   ii. Sample z_{ik} from Bernoulli(p), where p is calculated above.

3. Sample the number of new features $K^{(i)}$ for object i:

   i Approximate the posterior distribution of $K^{(i)}$ by the truncated distribution (we truncate it at 5):

   $$\begin{aligned}
   p(K^{(i)} = 0|\mathbf{X}, \mathbf{Z}, \sigma_A, \sigma_X, \alpha) &= p(\mathbf{X}|\mathbf{Z}, \sigma_A, \sigma_X)p(K^{(i)} = 0)\\
   p(K^{(i)} = 1|\mathbf{X}, \mathbf{Z}, \sigma_A, \sigma_X, \alpha) &= p(\mathbf{X}|\mathbf{Z}, \sigma_A, \sigma_X)p(K^{(i)} = 1)\\
   &\vdots\\
   p(K^{(i)} = 5|\mathbf{X}, \mathbf{Z}, \sigma_A, \sigma_X, \alpha) &= p(\mathbf{X}|\mathbf{Z}, \sigma_A, \sigma_X)p(K^{(i)} = 5)
   \end{aligned}$$

   ii Assign a value from 0 to 5 to $K^{(i)}$ based on computed probabilities above.

4. Sample σ_X through Metropolis-Hasting:

   i Propose a new σ_X^* from InvGamma(3,2)

   ii Compute the posterior probability for σ_X^* and σ_X as follow:

   $$\begin{aligned}
   p(\sigma_X^*|\mathbf{X}, \mathbf{Z}, \sigma_A, \sigma_X) &= p(\mathbf{X}|\mathbf{Z}, \sigma_A, \sigma_X^*)p(\sigma_X^*)\\
   p(\sigma_X|\mathbf{X}, \mathbf{Z}, \sigma_A, \sigma_X) &= p(\mathbf{X}|\mathbf{Z}, \sigma_A, \sigma_X)p(\sigma_X)
   \end{aligned}$$

   iii Accept σ_X^* with probability $min\{1, \frac{p(\sigma_X^*|\mathbf{X},\mathbf{Z},\sigma_A,\sigma_X)}{p(\sigma_X|\mathbf{X},\mathbf{Z},\sigma_A,\sigma_X)}\}$

5. Sample a new α from its posterior Gamma(1+K,1+Hn), where Hn= $\sum_1^{h=i} \frac{1}{h}$

6. Repeat from step 2

3 Simulation and Realization

3.1 Simulated Data

3.1.1 Four features

Below is the simulated four features, represented by the 4×36 matrix $\mathbf{A}$ in our code:

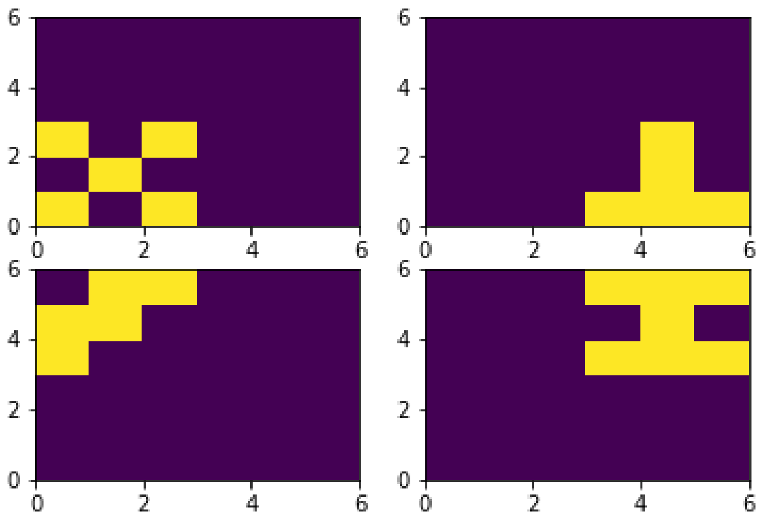

Figure 1: Four features used in our simulation

3.1.2 Simulated Z and X

Based on the linear-Gaussian latent feature model specified in section 2.2, we simulated **Z** and **X** to be as similar as possible to the dataset used by Griffiths and Ghahramani(2005)[1].

- **X** is $N \times D$, where $N = 100$ and $D = 36$
- **Z** is $D \times K$, where $K = 4$. z_{ik} is randomly chosen with probability 0.5

3.2 Realization of Gibbs Sampling

By comparing the **ZA** that we use to simulate data and the **ZA** generate by the algorithm, our code appears to largely catch the latent features.

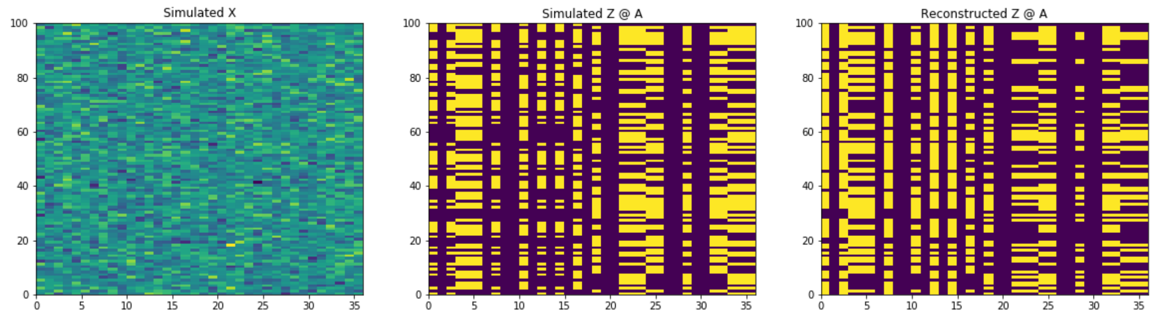

Figure 2: (1)The simulated X that we use to test our model, which is constructed by adding noise to simulated $\mathbf{ZA}$ (2)The simulated $\mathbf{Z}$ times $\mathbf{A}$ (3)After K converged, we multiply the current $\mathbf{Z}$ to matrix $\mathbf{A}$. Only the four most prominent features of $\mathbf{Z}$ are used.

By solving least square equations $\mathbf{ZA}_{recon} = \mathbf{X}$, with the $\mathbf{Z}$ obtained from sampler, and the simulated data $\mathbf{X}$, we obtain a reconstructed weight matrix $\mathbf{A}_{recon}$ that represents four most frequent features:

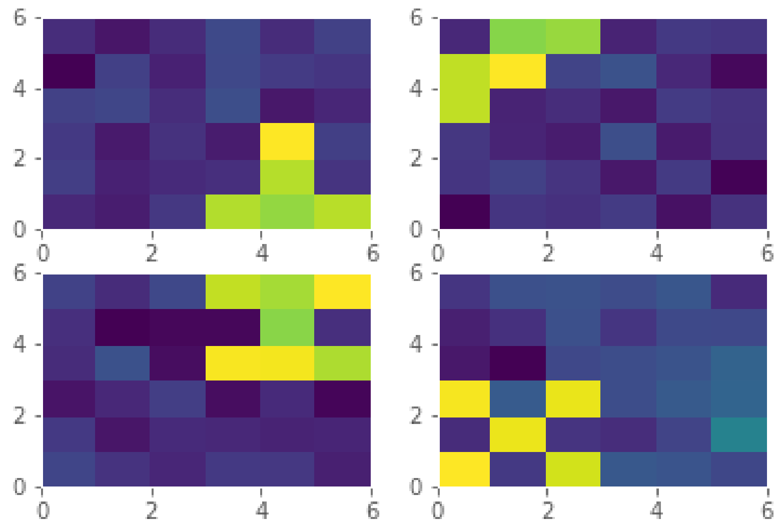

Figure 3: The four most frequent features captured by the sampler

The trace plots of the realization of our Gibbs sampler shows that the algorithm catches the features pretty well. After the first 50 loops, K quickly stabilizes to around 4. The variation was due to the Gaussian noise we added to the data, which may cause the model to confuse it with extra features. We also tried adjust σ_X, which is the variance of noise, the variation of K appears to be larger. The σ_X also quickly converges to its true value, 0.5, in the first 50 iterations.

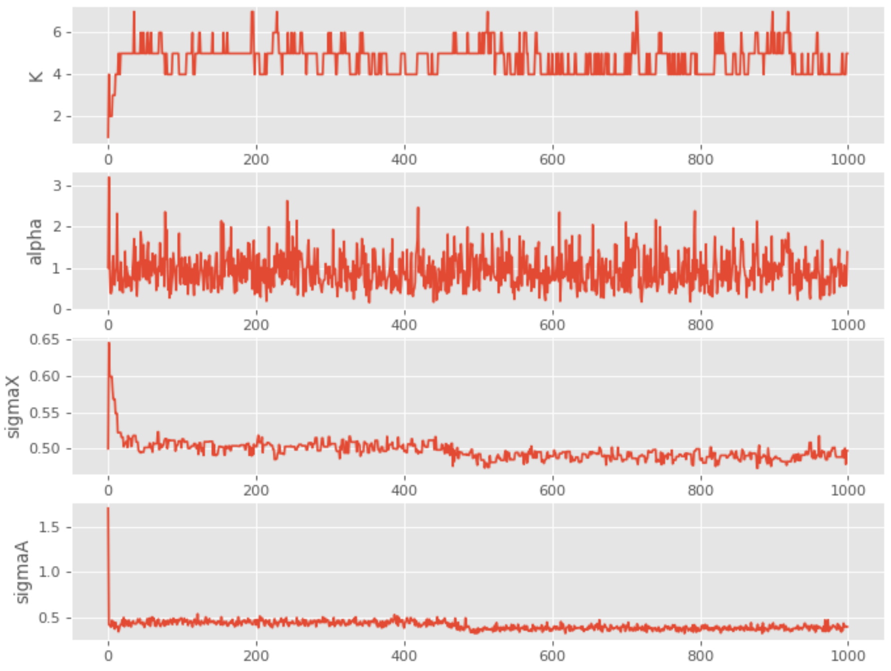

Figure 4: Trace plots for K, α, σ_X and σ_A

4 Optimization

The performance is measured through three functions:

Function name	Content
`new_K`	samples the number of new dishes for each object, uses `full_X`
`full_X`	computes the likelihood
`gibbs_sampler`	runs 1000 Gibbs sampling loops, use `new_K` and `full_X`

The biggest problem in optimizing the Gibbs sampler is that it cannot be parallelize directly. The naive single chain sampler keeps updating the parameters to their new states, and therefore each loop is dependent on the previous results.

Please note that, due to the different environments in which the code is running (on server or local), the total times of a complete 1000 iterations may vary a lot. Hence the performance of each optimization method should only be compared with the initial code run in the same environment together.

4.1 Profiling

We run the non-modified code for 100 loops and get the profile below (sorted by total time):

```
        16095564 function calls in 194.052 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
196686   51.293   0.000 168.508   0.001 :7(full_X)
393372   11.515   0.000  27.436   0.000 :197(diag)
1279117 10.415   0.000  10.415   0.000 :0(array)
196686    9.420   0.000  27.743   0.000 :534(cholesky)
196686    9.213   0.000  26.578   0.000 :464(inv)
590058    8.982   0.000  19.384   0.000 :138(_commonType)
196686    8.646   0.000  24.195   0.000 :1822(det)
1         7.361   7.361 194.049 194.049 :11(gibbs_sampler)
1180116   5.384   0.000   8.042   0.000 :110(isComplexType)
590058    5.275   0.000   8.776   0.000 :208(_assertNdSquareness)
196686    5.095   0.000   5.095   0.000 :0(trace)
1799874   4.686   0.000   4.686   0.000 :0(issubclass)
590058    4.388   0.000   4.388   0.000 :0(astype)
462677    4.106   0.000   4.106   0.000 :0(zeros)
689058    3.997   0.000  12.650   0.000 numeric.py:495(asanyarray)
393372    3.727   0.000   8.060   0.000 :105(_makearray)
9900      3.495   0.000  56.156   0.006 :10(new_K)
196686    3.355   0.000  11.261   0.000 fromnumeric.py:1364(trace)
590058    2.984   0.000   4.746   0.000 numeric.py:424(asarray)
590058    2.981   0.000   4.422   0.000 :123(_realType)
658701    2.965   0.000   2.965   0.000 :0(max)
590058    2.158   0.000   2.158   0.000 :197(_assertRankAtLeast2)
```

As we can see from the table, the self-defined function `full_X` uses almost a quarter of the total running time, and has the longest running time in all functions, as expected. Within the `full_X`, `np.linalg.inv` and `np.linalg.det` are the two most time costly functions.

4.2 Change functions

To shorten the running time as much as possible without changing the majority of the code, we aim to modify some of the mostly used function in it.

Our original code use `np.diag()` when creating new identity matrix when computing likelihood, which costs a lot of time. We replace `np.diag()` with `np.eye()`, a function more tailored for creating identity. Also, we remove the cholesky decomposition, and just use `np.linalg.inv()` directly. This costs slightly less time than the original version (18.6s to 10.1s for 100 loops).

```
        10588356 function calls in 145.830 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
196686   43.740    0.000  120.474    0.001 :7(full_X)
196686   10.119    0.000   28.736    0.000 :464(inv)
196686    8.656    0.000   24.325    0.000 :1822(det)
393372    7.875    0.000   12.430    0.000 :140(eye)
```

4.3 Cython and JIT

To further optimize the code, we cythonized the optimized functions. The table below shows the total running times of the 1000 iterations (complete algorithm) of the cythonized code and the original code:

code	run time in sec
Cython `new_K` and `full_X`	457.5609
Cythonized code	411.7675
JIT	154.8574
Initial code	463.2420

We see that Cython `new_K` and `full_X` and cythonizing only slighly improve the performance (in fact the run time for cython version is unstable, sometimes slightly higher than the optimimized version and sometimes lower). This is not too surprising because all our codes are already written using the numpy package which is inherently coded in C[2]. We also see that JIT improves the results significantly (approximately one third of the time taken by the initial code). Therefore, JIT is the most effective method among other choices for optimization.

4.4 Multiprocessing

We also tried to apply Multiprocessing to the function `new K`, which computes 5 independent Poisson probabilities . `new_K` is nested in the Gibbs sampler and therefore cannot be extracted out. However, the large number of Gibbs for loops make multiprocessing impossible to work, since it exceeds the limit of maximum processors we can have.

5 Discussion and Conclusion

We rebuilt the algorithm of detecting infinite latent features with Indian Buffet Process, and test it on a linear-Gaussian model. The results show that the algorithm works very well in our case, but noise and variation in the data may destabilize the posterior we got. Moreover, our attempts at optimization were restricted, due to the nature of Gibbs sampler. There are successful modifications that shorten the running time, such as our changes of functions(section 4.2) and `jit`(section 4.3). However, this model may requires a more complicated algorithm to be further optimized by parallel. For example, the paralleled Gibbs sampling by color grouping from Gonzalez,Low,Gretton and Guestrin[3].

6 Link to Github

`https://github.com/wy49/663FinalProject-YZ`

References

[1] TL. Griffiths and Z. Ghahramani. *Infinite latent feature models and the indian buffet process*, 2005.

[2] Cython Compilation. *Cython 0.28.2 Documentation*, 2018.

[3] JE. Gonzalez, Y. Low, A. Gretton and C. Guestrin. *Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees*, 2011.