Claims Severity Modeling

by

Radhika Anand

Program in Statistical and Economic Modeling
Duke University

Date: ____________________________

Approved

______________________________________________

Sayan Mukherjee, Supervisor

______________________________________________

Surya Tapas Tokdar

______________________________________________

Kent P Kimbrough

Thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Science in the Program in Statistical and Economic Modeling
in the Graduate School
of Duke University

2015
ABSTRACT

Claims Severity Modeling

by

Radhika Anand

Program in Statistical and Economic Modeling
Duke University

Date: ________________________

Approved

_____________________________

Sayan Mukherjee, Supervisor

_____________________________

Surya Tapas Tokdar

_____________________________

Kent P Kimbrough

An abstract of a thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in the Program in Statistical and Economic Modeling in the Graduate School of Duke University

2015
Abstract

This study is presented as a portfolio of three projects, two of which were a part of my summer internship at CNA Insurance, Chicago and one was a part of the course STA 663: Statistical Computation.

Project 1, Text Mining Software, aimed at building an efficient text mining software for CNA Insurance, in the form of an R package, to mine sizable amounts of unstructured claim notes data to prepare structured input for claims severity modeling. This software decreased run-time 30 fold compared to the software used previously at CNA.

Project 2, Workers’ Compensation Panel Data Analysis, aimed at tracking workers’ compensation claims over time and pointing out variables that made a claim successful in the long run. It involved creating a parsimonious Mixed Effects model on a panel dataset of Workers’ Compensation claims at CNA Insurance.

Project 3, Infinite Latent Feature Models and the Indian Buffet Process (IBP), used IBP as a prior in models for unsupervised learning by deriving and testing an infinite Gaussian binary latent feature model. An image dataset was simulated (similar to Griffiths and Ghahramani (2005) [1]) to test the model.
## Contents

Abstract ................................................................................................................................. iv

List of Tables .......................................................................................................................... viii

List of Figures ........................................................................................................................ ix

Acknowledgements ............................................................................................................... x

1 Text Mining Software ........................................................................................................ 1

1.1 Introduction ..................................................................................................................... 1

1.1.1 Motivation .................................................................................................................. 2

1.1.2 Proposed Text Analytics Pipeline .............................................................................. 2

1.2 Software Functionality .................................................................................................... 4

1.3 Software Design, Methodology and Implementation ....................................................... 5

1.4 Breakdown of Speed Improvements .............................................................................. 6

1.5 Final R Package: ‘cta’ .................................................................................................... 7

1.6 Applications .................................................................................................................... 8

2 Workers’ Compensation Panel Data Analysis .................................................................. 9

2.1 Motivation ....................................................................................................................... 9

2.2 Dataset ........................................................................................................................... 9

2.3 Model Specifications ..................................................................................................... 11

2.4 Initial Insights ............................................................................................................... 12

2.5 Next Steps ..................................................................................................................... 13

3 Infinite Latent Feature Models and Indian Buffet Process ............................................... 14
### 3.1 Background

- **3.1.1 The Indian Buffet Process (IBP)**
- **3.1.2 Applications**

### 3.2 Implementation

- **3.2.1 Infinite Linear-Gaussian Binary Latent Feature Model**
- **3.2.2 Algorithm**
  - 3.2.2.1 Likelihood
  - 3.2.2.2 Gibbs Sampler
  - 3.2.2.3 Metropolis Hastings

### 3.3 Profiling and Optimization

- **3.3.1 Optimizing Matrix Inverse**
- **3.3.2 Optimizing Likelihood Function and the Sampler**
- **3.3.3 Cythonizing**

### 3.4 High Performance Computing

- **3.4.1 Multicore Programming**
- **3.4.2 GPU Programming**

### 3.5 Application and Results

- **3.5.1 Data Simulation**
- **3.5.2 Results**
  - 3.5.2.1 Detection of total number of latent features
  - 3.5.2.2 Detection of latent features present in each object and Object Reconstruction
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.2.3 Validation</td>
<td>28</td>
</tr>
<tr>
<td>3.6 Comparison</td>
<td>28</td>
</tr>
<tr>
<td>3.6.1 Comparison with MATLAB Implementation</td>
<td>28</td>
</tr>
<tr>
<td>3.6.2 Comparison with Chinese Restaurant Process</td>
<td>29</td>
</tr>
<tr>
<td>3.7 Conclusion</td>
<td>31</td>
</tr>
<tr>
<td>Appendix A</td>
<td>32</td>
</tr>
<tr>
<td>References</td>
<td>33</td>
</tr>
</tbody>
</table>
List of Tables

Table 1: Run-times and times speedup ................................................................. 6

Table 2: Runtimes for inverse functions (for 1000 loops) ........................................ 20

Table 3: Runtimes for likelihood function (for 1000 loops) ..................................... 21

Table 4: Total runtimes .......................................................................................... 21

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 in Figure 9 (in that order). .......... 28
List of Figures

Figure 1: Proposed text analytics pipeline .........................................................3
Figure 2: Text mining software functionality ......................................................4
Figure 3: Software design, methodology and implementation ............................6
Figure 4: Snapshot of R package: ‘cta’ .................................................................7
Figure 5: Panel data snapshot ............................................................................11
Figure 6: Univariate statistics by triage group ....................................................12
Figure 7: Variables that increase/decrease the odds of claim close .....................13
Figure 8: Profiler results ...................................................................................19
Figure 9: Features/basis images used to simulate data ........................................24
Figure 10: Simulated data (first four of 100 images) .........................................24
Figure 11: Trace plots for K, alpha, sigma X and sigma A .................................26
Figure 12: Histograms. (a) Posterior of K (b) Features (ordered by frequency of occurrence) ..............................................................26
Figure 13: Features detected by code ...............................................................27
Figure 14: Reconstructed images .....................................................................27
Figure 15: Profiling results of MATLAB code for IBP ......................................29
Figure 16: Chinese Restaurant Process ............................................................30
Figure 17: Indian Buffet Process .....................................................................31
Figure 18: Sample hash process .......................................................................32
Acknowledgements

I extend a sincere thanks to my wonderful professors at Duke University and amazing mentors at CNA Insurance, Chicago.
1 Text Mining Software

The goal of the project is to build a new text mining software for CNA Insurance, in the form of an R package, using an efficient design, to achieve significantly lesser run-times compared to their existing software.

1.1 Introduction

Claims severity modeling and subsequent loss reserving are two critical backbones of any insurance business. Thus, insurance companies invest a lot of time and effort in predicting how severe a claim would be in the future and the dollar reserves that would be needed to handle the claim.

Unfortunately, a major chunk of the initial data they gather about a claim is unstructured text data. Whenever a person files a claim, the insurance company receives text notes, written by an insurance ‘adjustor’. These notes contain suggestive information about the claim, off which some information is very relevant to model its future severity. However, since unstructured text data can’t be used as input to a severity model, they employ text mining softwares to convert the unstructured text data into a structured form.

Further, since the amount of text data is very huge, building the text mining software is not the only goal. An equally important goal is to build an efficient one, with minimum achievable time complexity.
1.1.1 Motivation

Prior to summer 2015, CNA had an existing text mining software built and licensed buy a third party company. There were a few issues with this software.

- It was slow and inefficient as it took almost a week to run on a complete set of around 16 million notes from past 5 years.
- It was written in SAS. And CNA, then, was in a phase of transitioning all its softwares to R.
- CNA gradually wanted become independent of the third party company and have all its softwares built in-house.

1.1.2 Proposed Text Analytics Pipeline

Figure 1 provides a high-level overview of the proposed text analytics pipeline at CNA. On the top left we see sample raw claim notes, which have a lot of potential information for claims modeling. These notes (unstructured data) then pass through the pre-processing phase (including case change, stemming, lemmatization, punctuation/stop-word removal, spell correction etc.). The pre-processed notes then pass through the mining (n-gram/flag creation) and natural language processing (parts of speech tagging, parsing, topic modeling, information retrieval) phase to get final structured variables as output, which can directly be used for several projects such as severity modeling, fraud detection, triaging, predicting duration etc.
Different interns worked on different parts of the pipeline. I particularly worked on the mining part shown in pink, above. I built the text mining software from scratch, in the form of an R package (using R and C), implementing a completely new and efficient design. Its functionality, design, methodology and implementation are described in the subsequent sections.
1.2 **Software Functionality**

Figure 2 describes the functionality of the text mining software. Inputs to the software include:

- Notes file, consisting of pre-processed claim notes (top-left)
- Term worksheet, a dictionary of n-gram* to flag association (bottom-left)

The output is as shown on top-right, which is a file with rows equal to the number of notes and columns equal to the number of flags. Whenever a note consists of an n-gram, its corresponding flag value (as matched from the term worksheet) is set to 1 and flag count is set to the number of times that n-gram appears in that note.

![Diagram of text mining software functionality](image)

**Figure 2: Text mining software functionality**

* n-gram is a contiguous sequence of n items in a text. We use 1-6 grams to model a context.
1.3 Software Design, Methodology and Implementation

The software is broken down into 3 major blocks, as seen in Figure 3, each of which is implemented in a way to produce the most efficient design.

The 3 blocks, in order of execution, are:

1) N-gram creation – This piece divides the notes into 1-6 N-grams, to model the context in the notes. This is done using library ‘RWeka’ which is inherently coded in Java. Each N-gram is stored in a temporary storage to minimize space complexity.

2) Dictionary matching – This piece matches n-grams in the notes to n-grams in the dictionary to set corresponding flags. This piece uses a novel architecture, which plays a major role in run-time improvement. The dictionary is implemented using a hash-table data structure (refer Appendix A), which has a constant O(1) average case lookup time. This is multiple orders faster compared to its naïve counterpart i.e. going through the entire dictionary each time, which has O(n) matching.

The entire hash process is implemented in C and then integrated with R. The C script uthash and its UT_hash_handle functionality is used for hash creation and lookup.

3) Flags’ storage – Final results of notes and their corresponding flags are stored in the form of sparse matrices since only very few flags are 1 for each note. This further yields in reducing both space and time complexity.
Both single and multicore/parallel versions of the software are implemented.

### 1.4 Breakdown of Speed Improvements

Table 1 shows a breakdown of how the speed improvements were attained, reported on a small sample of 2000 notes. The third party external software took about 26 seconds to run on 2000 notes and the new software took just about 1 second implying a speedup of approximately 26 times.

**Table 1: Run-times and times speedup**

<table>
<thead>
<tr>
<th>Technique</th>
<th>Total time (in sec.)</th>
<th>Times speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) External software (in SAS)</td>
<td>26</td>
<td>x1 (reference)</td>
</tr>
<tr>
<td>2) CNA: Hashing in R + Sparse Matrix in R</td>
<td>15</td>
<td>x1.74</td>
</tr>
<tr>
<td>3) CNA: Hashing in C + Sparse Matrix in R</td>
<td>1</td>
<td>x26</td>
</tr>
<tr>
<td>4) Parallel processing*</td>
<td>0.92</td>
<td>x28</td>
</tr>
</tbody>
</table>

*Higher the number of notes, more evident is the parallel processing speedup*
This implies that we, now, need only 6 hours to convert 16 million notes to structured input for severity modeling, compared to a week that was required previously using the third party software!

1.5 Final R Package: ‘cta’

Figure 4 shows a snapshot of the final R package called ‘cta’, which stands for CNA Text Analytics. The function `match_dict` in this package consists of majority of my R and C code in its backend.

![Figure 4: Snapshot of R package: ‘cta’](image-url)
1.6 **Applications**

- ‘cta’ is of common use to all lines of business at CNA, to prepare structured input, for modeling, triaging and other projects.

- Can easily be used for any task involving lookups in the future i.e. with any new dictionary.

- Flag counts, generated, can be used to model probability distributions.
2 Workers’ Compensation Panel Data Analysis

The goal of this project is to track the development of workers’ compensation (WC) claims over time and point out variables (particularly medical) that make a claim successful in the long run. A claim is termed as successful based on several factors such as faster claim close, favorable ultimate loss, faster return to work, faster maximum medical improvement etc.

2.1 Motivation

The existing 5 and 28-day workers’ compensation severity models at CNA provide a rich WC dataset, which together with the medical bill database can yield in an interesting time and claim varying panel dataset.

Furthermore, medical plans, diagnosis etc. are very dynamic and change almost every year and hence a time-varying analysis seems like an interesting avenue to explore. Such analysis is also expected to drive WC business decisions and help with case management, utilization review etc.

2.2 Dataset

Using the workers compensation dataset and medical bill dataset, we construct a panel dataset over claims \( n \) and time \( t \). For each of the \( N \) claims (around 16 million), we have data on 3544 predictors over time \( t \) at 28, 60, 90, …, 360 days. It is a micro panel with large \( N \) and small \( T \). Thus, units roots, structural breaks, co-integration etc. are not big concerns.
A snapshot of the panel dataset is shown in Figure 5. It shows data on 2 claims, where green signifies a less severe claim compared to the red claim.

The time point is shown in pink, where 28 refers to the value of each variable using the development of claim upto 28 days from when it is filed, 60 refers to value of each variable using the development of claim upto 60 days and so on.

The target variable, shown in purple, is a binary indicator based on time to close i.e. it is 0 till the time the claim is open and changes to 1 when it closes. Going forward we will add incurred loss, return to work, maximum medical improvement etc., to get a single composite indicator of a successful claim.

Then, as independent variables, we have both time-invariant and time-varying variables. We have time-invariant claimant details as shown in orange and time-varying medical bill-paid, medical visit and text variables (shown in green). For eg., in Figure 5, PDInaptientSurgery refers to the amount paid on inpatient surgery for this claimant upto each specified time point and VISInpatientSurgery is the number of visits that the claimant takes for inpatient surgery upto each time point. Similarly we have time-varying text variables (not shown in Figure 5) eg. AN_BACK, which is 0 until the keywords (n-grams) associated to the flag BACK appear in one of the adjustors notes for that claim.
2.3 Model Specifications

A Mixed-Effects Logistic Model is used to model this dataset. It is built off a random sample of 10,000 claims from 2007-14. Variable selection is done using a combination of Lasso and general intuition & testing. Random effects are used for intercept and slope on TIME_POINT, based on triage group. Diagonal covariance structure is used.

Triage groups refer to the severity groups where red is the most severe and green is the least severe. The random effects are based on triage group since claims in different triage
groups behave very differently. Some univariate statistics by triage group are shown in Figure 6.

![Univariate statistics by triage group and claim center](image)

**Figure 6: Univariate statistics by triage group**

Further, the data is right-censored at 365 days so that each claim gets the same time to develop. Several economic control variables are added such as for year and different states. Variables are transformed, logged and scaled where appropriate. Variance inflation factors are checked to prevent multi-collinearity.

### 2.4 Initial Insights

Figure 7 shows a few variables that increase/decrease the odds of claim close. Some initial insights are as follows:

- Presence of comorbidities decreases the odds of claim close
- Claims in IL close faster than those in NY and CA
- Claims for males close faster than for females
• Passive vs aggressive treatment: Inpatient surgery, Ambulance visits, ER visits, emergency care visits etc. increase odds of claim close unlike physical therapy visits, general drugs usage

• Prosthetic and psychiatric visits decrease odds of close

![Figure 7: Variables that increase/decrease the odds of claim close](image)

### 2.5 Next Steps

Some of the next steps to be undertaken by my colleague, in this project, are:

• Add Maximum Medical Improvement (MMI), Return to Work and Incurred Loss to the target variable

• Add more predictor variables

• Look at medical paid/visit variables more granularly

• Testing and Validation

• Model monitoring
3 Infinite Latent Feature Models and Indian Buffet Process

This project uses Indian Buffet Process as a prior in models for unsupervised learning by deriving and testing an infinite Gaussian binary latent feature model. An image dataset is simulated (similar to Griffiths and Ghahramani (2005) [1]) to test the model.

3.1 Background

The Indian Buffet process is very interesting in its approach to model objects using Bayesian Non-Parametrics, assuming the true dimensionality is unbounded. This concept is new to me and very intriguing at the same time. Statistical models exist, that provide latent structure in probabilistic models, but the critical question is the unknown dimensionality of the representation, i.e. how many features are required to express the latent structure. Bayesian Non-Parametrics is an answer to this question. One way is to use the Chinese Restaurant Process, which assigns each object to only one feature/class of the infinite array of features. The Indian Buffet Process extends this problem through its potential to assign an object (customer) to multiple features (dishes) [1]. As an example, we would prefer characterizing a person as married, atheist, female and democrat rather than simply assigning the person to one class.

3.1.1 The Indian Buffet Process (IBP)

The name Indian buffet process is derived from Indian restaurants in London that offer buffets with nearly infinite number of dishes. Formally, in the IBP, N customers enter a
restaurant one after the other. 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 Poisson(\(\alpha\)) number of dishes. The \(i^{th}\) customer moves along the buffet, sampling dishes in proportion to their popularity, picking dish \(k\) with probability \(m_k/i\), where \(m_k\) is the number of previous customers who have sampled that dish. Reaching the end of all previous sampled dishes, the \(i^{th}\) customer then samples a Poisson(\(\alpha/i\)) number of new dishes. We indicate which customers choose which dishes using a binary matrix \(Z\) with \(N\) rows and infinite columns. \(z_{ik}\) is 1 if the \(i^{th}\) customer samples the \(k^{th}\) dish [1].

Formally, \(Z \sim IBP(\alpha)\) as:

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

where, \(m_k\) is the number of objects with feature \(k\), \(K_{i}^{(i)}\) is the number of new dishes sampled by the \(i^{th}\) customer and \(H_N\) is the \(N^{th}\) harmonic number given by: \(H_N = \sum_{j=1}^{N} 1/j\)

In conditional probability terms (after taking the infinite limit), this can be expressed as:

\[
P(z_{ik} = 1|z_{-i,k}) = \frac{m_{-i,k}}{N}
\]

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 \(i\).
3.1.2 Applications

The Indian Buffet Process has a myriad of applications in Bayesian Non-Parametrics for latent feature allocation. It can be used to define a prior distribution in any setting where the latent structure in the data can be expressed as a binary matrix with a finite number of rows and infinite number of columns, such as the adjacency matrix of a bipartite graph where one class of nodes is of unknown size, or the adjacency matrix for a Markov process with an unbounded set of states [1].

One application is to use it as a prior in infinite latent feature models. An example is shown in our paper below where we model a noisy image dataset to detect its underlying features. Another example is proposed by Jacob and Yildirim [5], where they apply IBP to unsupervised multisensory perception.

Despite the far-reaching advantages of IBP, a technical issue arises in models where feature values have to be represented explicitly and the structure of the model does not permit the use of conjugate priors. Care has to be taken that the posterior distributions remain proper [1].

3.2 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].
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 $x_i \sim N(z_iA, \Sigma_X)$, 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.

3.2.2 Algorithm

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

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

3.2.2.1 Likelihood

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

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

3.2.2.2 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:

\[ P(z_{ik} = 1|z_{-i,k}) = \frac{m_{-i,k}}{N} \]

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 \( \text{Poisson} \left( \frac{\alpha}{N} \right) \)

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

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

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

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

\[ P(\alpha|Z) \sim \text{Gamma}(1 + K_+, 1 + H_N) \]

where, \( H_N \) is the \( N^{th} \) harmonic number and \( K_+ \) is the current number of features.

### 3.2.2.3 Metropolis Hastings

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

\[ \epsilon \sim \text{Uniform}(-.05,.05) \]

\[ \sigma_X^* = \sigma_X + \epsilon \]

\[ \sigma_A^* = \sigma_A + \epsilon \]

Accept this new \( \sigma_X^* \) with acceptance probability:
\[ AP = \min \left\{ 1, \frac{P(X|Z, \sigma_*^X, \sigma_A)}{P(X|Z, \sigma_*^X, \sigma_A)} \right\} \]

Similarly for \( \sigma_*^A \).

### 3.3 Profiling and Optimization

The basic code was written in Python using package `numpy`. 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 in Figure 8. We clearly see that the functions; sampler, likelihood and inverse take the maximum amount of time along with matrix multiplication (`np.dot`).

![Profiler Results](image)

2194313 function calls in 9.094 seconds

Ordered by: internal time

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:linenofunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>286528</td>
<td>3.351</td>
<td>0.000</td>
<td>3.351</td>
<td>0.000</td>
<td>{numpy.core._dotblas.dot}</td>
</tr>
<tr>
<td>35816</td>
<td>1.212</td>
<td>0.000</td>
<td>6.131</td>
<td>0.000</td>
<td>&lt;python-input-88-d1e8e415007&gt;:2:likelihood</td>
</tr>
<tr>
<td>1</td>
<td>0.988</td>
<td>0.988</td>
<td>9.094</td>
<td>9.094</td>
<td>&lt;python-input-19-bb64a3c9505b&gt;:3:sampler</td>
</tr>
<tr>
<td>35816</td>
<td>0.485</td>
<td>0.000</td>
<td>1.086</td>
<td>0.000</td>
<td>linalg.py:455inv</td>
</tr>
<tr>
<td>107448</td>
<td>0.412</td>
<td>0.000</td>
<td>0.758</td>
<td>0.000</td>
<td>twodim_base.py:190:eye</td>
</tr>
<tr>
<td>112652</td>
<td>0.355</td>
<td>0.000</td>
<td>0.355</td>
<td>0.000</td>
<td>{numpy.core.multiarray.zeros}</td>
</tr>
<tr>
<td>35816</td>
<td>0.333</td>
<td>0.000</td>
<td>0.778</td>
<td>0.000</td>
<td>linalg:py:1679:det</td>
</tr>
<tr>
<td>35816</td>
<td>0.314</td>
<td>0.000</td>
<td>0.314</td>
<td>0.000</td>
<td>{method 'trace' of 'numpy.ndarray' objects}</td>
</tr>
<tr>
<td>21332</td>
<td>0.255</td>
<td>0.000</td>
<td>0.255</td>
<td>0.000</td>
<td>{sum}</td>
</tr>
<tr>
<td>71632</td>
<td>0.189</td>
<td>0.000</td>
<td>0.329</td>
<td>0.000</td>
<td>linalg.py:139:commonType</td>
</tr>
<tr>
<td>112782</td>
<td>0.120</td>
<td>0.000</td>
<td>0.120</td>
<td>0.000</td>
<td>{numpy.core.multiarray.array}</td>
</tr>
<tr>
<td>71632</td>
<td>0.092</td>
<td>0.000</td>
<td>0.149</td>
<td>0.000</td>
<td>linalg.py:209:assertNdSquareness</td>
</tr>
<tr>
<td>107448</td>
<td>0.090</td>
<td>0.000</td>
<td>0.191</td>
<td>0.000</td>
<td>numeric.py:394:asarray</td>
</tr>
<tr>
<td>82361</td>
<td>0.090</td>
<td>0.000</td>
<td>0.090</td>
<td>0.000</td>
<td>{max}</td>
</tr>
<tr>
<td>35816</td>
<td>0.081</td>
<td>0.000</td>
<td>0.081</td>
<td>0.000</td>
<td>{method 'astype' of 'numpy.generic' objects}</td>
</tr>
<tr>
<td>10980</td>
<td>0.077</td>
<td>0.000</td>
<td>0.077</td>
<td>0.000</td>
<td>{method 'uniform' of 'mtrand.RandomState' objects}</td>
</tr>
<tr>
<td>71632</td>
<td>0.075</td>
<td>0.000</td>
<td>0.090</td>
<td>0.000</td>
<td>linalg.py:198:assertRankAtLeast2</td>
</tr>
</tbody>
</table>

**Figure 8: Profiler results**
3.3.1 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 2 that this `calcInverse` takes nearly half the time taken by the `np.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>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>linalg.inv</td>
</tr>
<tr>
<td>calcInverse</td>
</tr>
</tbody>
</table>

3.3.2 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 $Z^T Z + \frac{\sigma_z^2}{\sigma^2} I$ matrix in the sampler each time before sending it to the likelihood function. Then in the likelihood function we had determinant of this same matrix. To get rid of the redundancy, we removed all inverse calculations outside the likelihood function and instead just it calculated once in the likelihood
function and then took its inverse and determinant. This reduced the time taken by the likelihood function as can be seen in Table 3. The gain does not appear very significant here but is indeed high when seen together with the sampler.

We also 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 this optimized sampler (together with the optimized likelihood). We see that there is a significant decrease in the time taken by the optimized version compared to the naive one.

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

<table>
<thead>
<tr>
<th></th>
<th>Time (in secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old Likelihood</td>
<td>0.161591</td>
</tr>
<tr>
<td>New Likelihood</td>
<td>0.148518</td>
</tr>
</tbody>
</table>

**Table 4: Total runtimes**

<table>
<thead>
<tr>
<th></th>
<th>Time (in secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>343.536284</td>
</tr>
<tr>
<td>Optimized</td>
<td>296.065528</td>
</tr>
<tr>
<td>Cythonized</td>
<td>299.767935</td>
</tr>
</tbody>
</table>
3.3.3 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 (in fact the run-time for cython version is unstable, sometimes slightly higher than the optimized 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.

3.4 High Performance Computing

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

3.4.1 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.
3.4.2 GPU Programming

Next, we used 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 \texttt{np.dot} was found to be faster than CuBLAS matrix multiplication.

Thus, we use the optimized likelihood and sampler described in Section 3.3.2 as the final version. In the comparison section 3.6.1, we see how this code is faster and more efficient compared to available IBP codes online.

3.5 Application and Results

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

3.5.1 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, 6x6 images represented as a 100*36 matrix where each image/object has a D-dimensional vector of properties, $x_i$:

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

![Figure 9: Features/basis images used to simulate data](image)

![Figure 10: Simulated data (first four of 100 images)](image)

Figure 9 shows the 4 features (basis images) used to generate our simulated data and Figure 10 shows first four of the 100 simulated images which have one or more of the features and added noise.
3.5.2 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 11.

3.5.2.1 Detection of total number of latent features

In Figure 12 (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 or two objects (which came in the form of added noise). Figure 12 (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 mean 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.
3.5.2.2 Detection of latent features present in each object and Object Reconstruction

Figure 13 shows the four most frequent features detected 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 9. They are just re-ordered.
Next, we reconstruct the images using $X_i \sim N(Z_iA, 0)$, where the posterior mean of the feature weights matrix $A$, given $X$ and posterior means of $Z$, $\sigma_A$ and $\sigma_X$ is:

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

Figure 14 shows the posterior means of the reconstructions of the four original data images. The reconstructions provided by the model in Figure 14 clearly pick out the relevant features present in each image, despite the high level of noise as seen in Figure 10.

![Figure 13: Features detected by code](image1)

![Figure 14: Reconstructed images](image2)
3.5.2.3 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 9. We clearly see that the reconstructed images, as in Figure 14, pick exactly the same features. The first reconstructed image (refer Figure 14) 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 9). 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 in Figure 9 (in that order).

<table>
<thead>
<tr>
<th></th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st image</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2nd image</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3rd image</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4th image</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

3.6 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.

3.6.1 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 13) and the reconstructed images (Figure 14). We then profiled his MATLAB code and got results as shown in Figure 15. 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. about 300 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.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Calls</th>
<th>Total Time</th>
<th>Self Time*</th>
<th>Total Time Plot (dark band = self time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sampler</td>
<td>1</td>
<td>410.090 s</td>
<td>73.272 s</td>
<td></td>
</tr>
<tr>
<td>likelihood</td>
<td>1365838</td>
<td>201.738 s</td>
<td>148.125 s</td>
<td></td>
</tr>
<tr>
<td>viabtimes</td>
<td>603367</td>
<td>63.163 s</td>
<td>63.163 s</td>
<td></td>
</tr>
<tr>
<td>trace</td>
<td>1365838</td>
<td>53.613 s</td>
<td>53.613 s</td>
<td></td>
</tr>
<tr>
<td>calcinverse</td>
<td>862838</td>
<td>45.357 s</td>
<td>45.357 s</td>
<td></td>
</tr>
<tr>
<td>factorial</td>
<td>500000</td>
<td>24.943 s</td>
<td>24.943 s</td>
<td></td>
</tr>
</tbody>
</table>

Figure 15: Profiling results of MATLAB code for IBP

3.6.2 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. CRP solves the clustering problem and IBP solves feature allocation problem. IBP allows each customer to be assigned to multiple features (dishes), while CRP assigns each customer to a single feature (table). Figures 16 and 17, 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.

![Diagram of Chinese Restaurant Process](image)

**Figure 16: Chinese Restaurant Process**
3.7 Conclusion

We derived and tested an Infinite Linear-Gaussian Binary Latent Feature model, using IBP as the prior, to detect the underlying features in a noisy image dataset. We validated our results by detecting the same features that were used to simulate the images. The basic code was written in Python, which was then optimized by removing redundant calculations and code statements and vectorization. Cython, JIT and high performance computing tools were tested. Since the code is highly dependent on matrix calculations and our profiling showed that approximate matrix inversion proposed by Griffiths and Ghahramani [1] was faster than full rank matrix inverse, future work would involve obtaining a stable Markov Chain with this approximate inverse technique.
Figure 18: Sample hash process
References


