MONASH University

Automatic Code Generation for Statistical Models with Augmentation and Collapsing

Sachith Hasaranga Seneviratne
BScEng(Hons)

A thesis submitted for the degree of *Doctor of Philosophy* at
Monash University in 2020
Faculty of Information Technology

Copyright notice

Abstract

With the advent of data science, statistical modelling has become increasingly popular over the past decade. Probabilistic programming languages contrive to make such models available to a broader audience, including disciplines such as physics, where efficiency and scalability are crucial to the overall modelling process. The encapsulation of statistical constructs into simple syntactical structures allows for models to be defined declaratively without requiring any programming expertise. While traditional probabilistic languages allow for rapid prototyping of machine learning models, they do not handle discrete parameters in a scalable manner.

This research aims to (1) develop efficient data structures and algorithms to represent the graphical structure of statistical models while maintaining caches of intermediate results such as sufficient statistics to allow for efficient updates and (2) explore the extent to which statistical operations such as augmentation, collapsing and Gibbs sampling can be supported and (3) develop a scheme with symbolic support for automatically performing these operations.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

Print Name: Sachith Hasaranga Seneviratne

Date: 27/05/2020

Acknowledgement

As I reach the conclusion of my PhD, I would like to offer my most sincere appreciation to the many people who have supported me on this journey.

Firstly, my amazing PhD supervisors Prof. Wray Buntine and Dr. Lan Du without whom this journey would not have been possible. Wray has been with me through every step of this journey, from start to finish and I am truly grateful for all the support and motivation he has provided since the day I met him at a machine learning meetup during my honours year, 4 years ago. Wray is a great wellspring of advice, whether it be about machine learning or life in general. My meetings with him have changed me for the better in more ways than one. Lan has been a great source of support for me, especially with experimental work. Lan's ability to see things from multiple different perspectives has improved the quality of my research work and my paper writing, while his insight into the best way to present the work to others has also been of great assistance.

I offer my sincere gratitude to my thesis panel members at Monash University, Mario Boley, Daniel Schmidt and Christoph Bergmeir for their valuable feedback which allowed me to improve my work, and provided much appreciated perspective. I would like to offer special thanks to Arun Konagurthu, who was a previous panel member at seminars where I presented my work. Your assistance with reference material and discussions was a great help when I was getting started with this topic.

I would also like to thank the lecturers at Monash University whom I taught alongside and have helped me in various ways, including listening to my research woes and offering helpful advice and perspective - Arun Konagurthu, Aamir Cheema, John Betts and Mario Boley.

I would like to thank my fellow PhD students at Monash, many of whom have helped me at different stages of my PhD. I offer special thanks to He Zhao, for his help with both understanding his work and writing papers. I am grateful to the members of our reading group, including Bhagya, Caitlin, Dilini, Kasun, Nandini and Penny for their feedback on my work and many insightful discussions. A very heartfelt thank you to all my friends on both campuses for the many great times we've shared together, it has truly been a pleasure.

Finally, I'd like to thank my family for their continued love and support without which I would not be the person I am today. A special thank you to my wife, Akla, for her love, understanding and patience in helping me to get to this point.

Contents

Chapter 1

Introduction

The recent growth in data science as a discipline has prompted widespread development in many areas of artificial intelligence and machine learning. Advances in data storage and processing technologies have allowed large collections of raw data to accumulate. The sheer size of such data warehouses require the development of highly scalable machine learning techniques such as generative statistical modelling, in order to provide valuable key insights about the data. These insights can then be translated into future predictions and inferences to assist in key decision-making.

While considerable research interest in machine learning leads to the requirement for many novel algorithms, the need for scalability and efficiency also means programming complexity. While skilled programmers would have no issues navigating such novel algorithms, this inherent programming difficulty limits the scope of applicability of novel machine learning techniques to other fields such as physics and biology. Researchers in these other fields may not have the programming cabability. Probabilistic programming languages are being developed as a solution to this problem.

1.1 Probabilistic programming languages

Probabilistic programming languages support implicit statistical constructs that users can directly call and use, similar to using a code library, in order to rapidly develop and test machine learning algorithms. These languages intend to provide several key advantages over using standard programming languages for the implementation of machine learning models:

- The ability to declare statistical models using short, simple syntax translates directly into a drastic reduction in programming time and effort for users.

- Higher ease of use as much of the complexity is abstracted away by the programming syntax and statistical constructs
- Improved code modularity and maintainability as the model grows in complexity, due to the declarative nature of the model

1.1.1 Gibbs sampling with BUGS

Consider the following specification of a Latent Dirichlet Allocation [5, 7] model implemented in BUGS [40] (a probabilistic programming language):

```
model {
  for (k in 1:K) {
    phi[k,1:V] ~ ddirich(beta[])
  }
  for (d in 1:M) {
    theta[d,1:K] ~ ddirich(alpha[])
    for (n in 1:N) {
      z[d,n] ~ dcat(theta[d,])
      w[d,n] ~ dcat(phi[z[d,n],])
    }
  }
}
```

While this model definition bears striking similarities to a standard imperative program, it is not actually executed in a top to bottom, left to right ordering as a standard program would be. The BUGS parser first converts this into a graphical structure, which can then be processed further in order to exploit relationships such as statistical conjugacy present between distributions.

While existing probabilistic programming languages perform quite well on models requiring sampling from continuous distributions, they do not perform quite as well on models which require discrete sampling [62]. Additionally, due to the innate nature of sampling, it is possible to optimize many models by providing parallel versions of the model in question. Error reporting can also be problematic in models due to the inherent complexity associated with the number of possible interactions between different types of distributions. Since probabilistic programming languages aim to be as general as possible, providing proper debugging support is highly desirable. While existing languages sometimes provide workarounds for potential issues arising in such situations, for

example: trying different priors, this may not be particularly helpful in the context in question [56].

1.1.2 Compiling to GPUs with Theano

The ability to generate higher level code (for example: Java or C++ code) for statistical models is very valuable, as it allows for closer scrutiny of the model in question. This also allows for easier embedding of models in different contexts, as the higher level code would be compilable on many different interfaces. An obvious extension is to also generate code that is compatible with GPUs or other distributed systems. Given the clear parallelizable nature of many models and sampling techniques, this would allow models to scale into very large datasets. Many neural network architectures use a similar approach in order to scale the problem they are solving to much larger datasets.

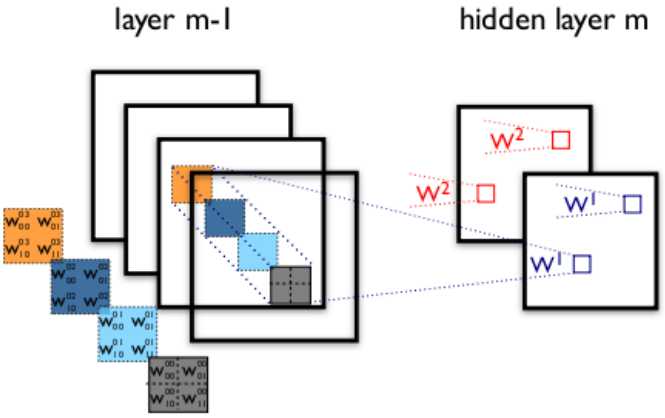


FIGURE 1.1: CNN diagram

Consider the following Theano [4] code for implementing neaural network based on a convolutional layer similar to the one of figure 1.1. The input consists of 3 features maps corresponding to an RGB color image of size 120x160. Two convolutional filters with 9x9 receptive fields are used. (Example sourced from [27]. Code non-essential to the model specification process has been omitted.) Of particular interest is the declarative nature of the model specification process and the relative ease with which complex architectural units can be defined with a few lines of code (for example, the convolutional layers).

```
x = T.matrix('x')   # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels

# Construct the first convolutional pooling layer:
```

```
layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 1, 28, 28),
    filter_shape=(nkerns[0], 1, 5, 5),
    poolsize=(2, 2)
)

# Construct the second convolutional pooling layer

layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 12, 12),
    filter_shape=(nkerns[1], nkerns[0], 5, 5),
    poolsize=(2, 2)
)

layer2_input = layer1.output.flatten(2)

# construct a fully-connected sigmoidal layer
layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 4 * 4,
    n_out=500,
    activation=T.tanh
)

# classify the values of the fully-connected sigmoidal layer
layer3 = LogisticRegression(input=layer2.output, n_in=500, n_out=10)

# the cost we minimize during training is the NLL of the model
cost = layer3.negative_log_likelihood(y)

# create a list of all model parameters to be fit by gradient descent
params = layer3.params + layer2.params + layer1.params + layer0.params

# create a list of gradients for all model parameters
grads = T.grad(cost, params)
```

In the case of deep neural networks, both the convolution process and the training of weights can be setup as matrix multiplication problems. Once this is done, the key complexity arises in efficiently parallelizing the matrix multiplication problem onto GPUs. There are distributed algorithms which perform these operations at high efficiency even for large datasets. This is automatically handled by most present deep learning systems. The result is that the complex processing which happens in the background is encapsulated from the user, who only needs to create a simple model specification making the entire process accessible to a much wider audience and thereby driving the overall popularity of the underlying techniques.

1.1.3 Programmable Inference with Gen

A key issue in probabilistic languages arises in providing flexibility in terms of supporting many different inference schemes. When different schemes are supported, it is important to provide users with some control of the overall inference process in order to ensure a robust system, that is able to meet the user's needs.

Gen [17] is a probabilistic programming language that allows users to gain a high degree of control over the inference process, by allowing them to write code for the inference process, while still leveraging other available inference libraries.

Consider the following model defined in Gen.

```
@gen function generate_datum(x::Float64, prob_outlier::Float64,
noise::Float64, @ad(slope::Float64), @ad(intercept::Float64))
    if @addr(bernoulli(prob_outlier), :is_outlier)
        (mu, std) = (0., 10.)
    else
        (mu, std) = (x * slope + intercept, noise)
    end
    return @addr(normal(mu, std), :y)
end

generate_data = MapCombinator(generate_datum)

@gen function model(xs::Vector{Float64})
    slope = @addr(normal(0, 2), :slope)
    intercept = @addr(normal(0, 2), :intercept)
```

```
    noise = @addr(gamma(1, 1), :noise)
    prob_outlier = @addr(uniform(0, 1), :prob_outlier)
    @diff begin
        addrs = [:prob_outlier, :slope, :intercept, :noise]
        diffs = [@choicediff(addr) for addr in addrs]
        argdiff = all(map(isnodiff,diffs)) ? noargdiff : unknownargdiff
    end
    n = length(xs)
    ys = @addr(generate_data(xs, fill(prob_outlier, n),
        fill(noise, n), fill(slope, n), fill(intercept,n)),
        :data, argdiff)
    return ys
end
```

Gen supports custom proposal distributions. In the code below, "is_outlier_proposal" is a custom proposal distribution which is used with Metropolis Hastings algorithm in "parameter_update"

```
@gen function is_outlier_proposal(previous_trace, i::Int)
    is_outlier = previous_trace[:data => i => :is_outlier]
    @addr(bernoulli(is_outlier ? 0.0 : 1.0), :data => i => :is_outlier)
end
```

```
and Julia functions to support inference.
```

```
function make_constraints(ys::Vector{Float64})
    constraints = Assignment()
    for i=1:length(ys)
        constraints[:data => i => :y] = ys[i]
    end
    return constraints
end
```

```
function parameter_update(trace, num_sweeps::Int)
    for j=1:num_sweeps
        trace = default_mh(model, select(:prob_outlier), trace)
        trace = default_mh(model, select(:noise), trace)
        trace = mala(model, select(:slope, :intercept), trace, 0.001)
    end
end
```

```
function is_outlier_update(trace, num_data::Int)
    for i=1:num_data
        trace = custom_mh(model, is_outlier_proposal, (i,), trace)
    end
    return trace
end
```

Users can then combine any of these aspects to perform inference. For example, in the following code, an inference program has been implemented using the previously defined make_constraints function. In this inference program, "default_mh" proposes new values for random choices based on a default proposal distribution, and uses the Metropolis-Hastings rule to accept or reject them. In situations where the default proposal function causes slow convergence, users may setup their own updates and initialization to attempt to improve performance. "is_outlier_update" updates the trace based on the custom proposal defined in "is_outlier_proposal" and assigns outlier status to a point based sampling from a bernoulli distribution. The model specification is defined in the "model" function, and this information is used in all future updates.

```
function inference_program_1(xs::Vector{Float64}, ys::Vector{Float64})
    constraints = make_constraints(ys)
    (trace, _) = initialize(model, (xs,), constraints)
    for iter=1:100
        selection = select(:prob_outlier, :noise, :slope, :intercept)
        trace = default_mh(model, selection, (), trace)
        trace = is_outlier_update(trace, length(xs))
    end
    return trace
end
```

The language provides access to some in-built helper functions for inference. In the example above, default_mh, select and initialize are examples of such functions.

Gen is built on top of Julia, and custom algorithms are incorporated using Julia functions. The inference function listed here uses default_mh, which proposes new assignments for random choices according to a default proposal distribution and then decides based on Metropolis-Hastings whether to accept or reject.

1.2 Research aims and evaluation

This section delves into the expected outcomes of this research as well as exploring avenues for evaluating the final framework created.

1.2.1 Research aims and expected outcomes

While many probabilistic programming languages have been proposed to automatically generate statistical models, and their general capabilities are approaching maturity in some areas, one area where we see they are not able to perform well is in sampling on discrete data where often times the probability models used do not support general schemes like naive Gibbs sampling [41] and Hamiltonian MC [10]. More generally we see some fundamental gaps in the capabilities of existing systems, as follows:

- There is no probabilistic programming language that takes full advantage of sufficient statistics and maintaining intermediate results along with statistical operations such as collapsing and augmentation in order to improve the efficiency of the sampling process (to the best of our knowledge).
- While some languages perform well in the case of continuous sampling, they perform much worse in the case of discrete sampling, due to the underlying sampling scheme. For example, this applies for the BUGS or Stan solutions for LDA, a characteristic problem in unsupervised learning.
- Most probabilistic languages are very general in the models they allow and in the functionality available as a programming language. However, doing so generally comes at the cost of lower efficiency than languages which allow a more restricted set of operations (Refer Section 3.1.1 and 3.1.3).

To overcome these issues, this research proposes a general architecture for a restricted class of models, which operates at a reasonable efficiency for supported models, and makes maximal use of sufficient statistics and intermediate results. This is developed as a capability that can be used to extend existing mature systems such as [10, 17], and is not intended as a full, independent standard alone probabilistic programming environment. For the purposes of experimentation, a basic probabilistic programming language was created and integrated with the system. While some initial exploration was done to integrate into Stan, this proved to be too difficult. The system developed will also aim to generate models which produce results with accuracy similar to those produced by the original models, as reported in literature. Additional support for statistical operations will be provided, leading to the possibility of easily extending simpler models in order to generate more complex models.

In conclusion, the expected research outcomes of this research include:

- Developing efficient data structures and algorithms to represent the intermediate graphical structure of statistical models while maintaining caches of intermediate results such as sufficient statistics
- Providing support for statistical operations such as augmentation and collapsing, using reusable components in order to simplify the process of constructing complex models as well as making the task of extending existing models more approachable.
- Developing a scheme with symbolic support for automatically performing these operations.

1.2.2 Worked Example

Consider evaluation in the case of LDA [5] for the model shown in figure 1.2:

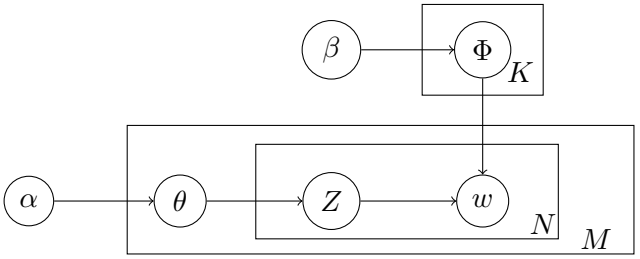


FIGURE 1.2: LDA Plate diagram

Here, M denotes the number of documents, N_m is number of words in document m. α is the parameter of the Dirichlet prior on the per-document topic distributions. β is the parameter of the Dirichlet prior on the per-topic word distribution. θ_m is the topic distribution for document m. Φ_k is the word distribution for topic k. $z_{m,n}$ is the topic for the n-th word in document m. $W_{m,n}$ is the n-th word in document m.

The generative process for LDA is as follows:

$$\begin{aligned}
\vec{\Phi}_k &\sim Dirichlet(\vec{\beta}) \\
\vec{\theta}_m &\sim Dirichlet(\vec{\alpha}) \\
\vec{z}_{m,n} &\sim Categorical(\vec{\theta_m}) \\
\vec{w}_{m,n} &\sim Categorical(\vec{\Phi}_{z_{m,n}})
\end{aligned}$$

From the model denoted above, the total probability can be defined as

$$P(W, Z, \theta, \Phi|\alpha, \beta) = \prod_{k=1}^{K} P(\Phi_k|\beta) \prod_{m=1}^{M} P(\theta|\alpha) \prod_{n=1}^{N_m} P(Z_{m,n}|\theta_m) P(W_{m,n}|\Phi Z_{m,n})$$

The standard analysis of LDA involves integrating out θ, Φ in order to do collapsed Gibbs sampling on Z. Using the independence of all θ_m and of all Φ_k, the integral can be simplified and separated to

$$\int_{\theta_m} P(\theta_m|\alpha) \cdot \prod_{n=1}^{N_m} P(Z_{m,n}|\theta_m) d\theta_m$$

Note that an equivalent formula exists for the Φ side as well for which the analysis is identical. The term from the θ side can be simplified to

$$\int_{\theta_m} \frac{\Gamma\left(\sum_{k=1}^{K} \alpha_k\right)}{\prod_{k=1}^{K} \Gamma(\alpha_k)} \prod_{k=1}^{K} \theta_{m,k}^{n_{m,k,(\cdot)}+\alpha_k-1} \, d\theta_j$$

Here $n_{m,k,v}$ denotes the number of times document m contains word v with the topic k assigned, and $n_{m,k,(\cdot)} = \sum_{v=1}^{V} n_{m,k,v}$ or in other words, $n_{m,k,(\cdot)}$ denotes the number of times topic k has been assigned in document m.

The integral above has the same form as the Dirichlet distribution and is finally simplified to

$$\int_{\theta_m} \frac{\Gamma\left(\sum_{k=1}^{K} \alpha_k\right)}{\prod_{k=1}^{K} \Gamma(\alpha_k)} \prod_{k=1}^{K} \theta_{m,k}^{n_{m,k,(\cdot)}+\alpha_k-1} \, d\theta_j = \frac{\Gamma\left(\sum_{k=1}^{K} \alpha_k\right)}{\prod_{k=1}^{K} \Gamma(\alpha_k)} \frac{\prod_{k=1}^{K} \Gamma(n_{m,k,(\cdot)} + \alpha_k)}{\Gamma(\sum_{k=1}^{K} n_{m,k,(\cdot)} + \alpha_k)}$$

This corresponds to integrating out θ on the document side. This does not require symbolic integration as the product term is an unnormalized Dirichlet distribution whose integral matches the normalizer of the distribution. On the word side, integrating out Φ is a similar process and leaves us with the following term:

$$\prod_{k=1}^{K} \frac{\Gamma\left(\sum_{v=1}^{V} \beta_v\right)}{\prod_{v=1}^{V} \Gamma(\beta_v)} \frac{\prod_{v=1}^{V} \Gamma(n_{(\cdot),k,v} + \beta_v)}{\Gamma\left(\sum_{v=1}^{V} n_{(\cdot),k,v} + \beta_v\right)}$$

The full conditional for a word indexed by (m,n) is required for Gibbs sampling as the update equation from which the sampler draws the hidden variable. This can be obtained from the joint distribution (formed by the two terms above). Hence, the update equation for the Gibbs sampler is given by (derived as per page 22 of [31] [1] and with v corresponding to $W_{m,n}$:

$$p(Z_{m,n} = k|Z_{-(m,n)}, w) \propto (n_{m,k,(\cdot)} + \alpha_k) \frac{n_{(\cdot),k,v} + \beta_v}{\sum_{v=1}^{V} n_{(\cdot),k,v} + \beta_v} \tag{1.1}$$

[1] http://www.arbylon.net/publications/text-est2.pdf

A key takeaway from this analysis is the need for symbolic reasoning in the context of optimizing efficiency. The counts $n_{m,k,v}$ are 3-dimensional in nature. However, by setting up the computation of the counts in the form of $n_{m,k,(\cdot)} = \sum_n 1_{Z_{m,n}=k}$, it is possible to perform the sampling using several 2-dimensional data structures, efficiently. In order to set up these data structures automatically, the compiler needs to be able to perform symbolic reasoning, which is one of the key challenges of this research.

Thus, the Gibbs sampling algorithm for Latent Dirichlet Allocation is as follows:

Initialization
$n_m^k = 0$ document-topic count
$n_m = 0$ document-topic sum
$n_k^v = 0$ topic-term count
$n_k = 0$ topic-term sum
for *all documents* $m \in [1, M]$ **do**
  for *all words* $n \in [1, N_m]$ **do**
    sample topic $Z_{m,n} = k$ from the multinomial distribution and increment counts
    n_m^k++; n_m++; n_k^v++; n_k++;
  end
end
Gibbs sampling over burn-in and sampling periods
while *not finished* **do**
  for *all documents* $m \in [1, M]$ **do**
    for *all words* $n \in [1, N_m]$ **do**
      given that word $w_{m,n}$ having term v is currently assigned to k
      $n_m^k - -$; $n_m - -$; $n_k^v - -$; $n_k - -$;
      Multinomial sampling according to update equation 1.1
      sample topic $k \sim$ equation above and use new assignment of topic to increment counts
      n_m^k++; n_m++; n_k^v++; n_k++;
    end
  end
  Check for convergence
end

Algorithm 1: Gibbs Sampling algorithm for LDA

Presented below is the code generated for LDA by the system. Note that the algorithm uses c0 for n_m^k, c0_1 for n_m, c1 for n_k^v and c1_1 for n_k

```
//Initialization
for (int m = 0; m < M; m++){
    for (int n = 0; n < N; n++){
```

```
            z[m][n]=Math.Random()*K;
            c0[m][z[m][n]]++;
            c0_1[m]++;
            c1[z[m][n]][w[m][n]]++;
            c1_1[z[m][n]]++;
        }
    }
    //For each iteration of the Markov Chain run the following:
    for (int m = 0; m < M; m++){
        for (int n = 0; n < N; n++){
            c0[m][z[m][n]]--;
            c0_1[m]--;
            c1[z[m][n]][w[m][n]]--;
            c1_1[z[m][n]]--;
            //Sample from full conditional
            double[] p = new double[K];
            for (int k = 0; k < K; k++){
                p[k]=(Math.pow(a_1+c0_1[m],-1))*(Math.pow(b_1+c1_1[k],-1))
                      *(c0[m][k]+a[k])*(c1[k][v]+b[v]);
            }
            //cumulate values
            for (int k = 1; k < K; k++){
                p[k]+=p[k-1];
            }
            int k;
            double val = Math.random()*p[K-1];
            for (k = 0; k < K; k++){
                if (p[k]>val)break;
            }
            z[m][n]=k;
            c0[m][z[m][n]]++;
            c0_1[m]++;
            c1[z[m][n]][w[m][n]]++;
            c1_1[z[m][n]]++;
        }
    }
```

In this case, it is apparent that the final resulting algorithm of the analysis process is quite straightforward to understand, while the analysis required to reach that point is not quite as simple. The generated algorithm can be checked for time and space complexity

by simple inspection. The similarity of any generated code to the human-generated version for this model could also similarly be tested. In general the human generated versions of the algorithm will be available in the publications related to that work and can be used as a baseline for comparison to the code/algorithm generated by this system. In addition to this, model-specific evaluation (such as perplexity for the model above) can be carried out in a similar fashion by running the evaluation metric on both the standard version and generated version of the algorithm followed by a comparison of results.

1.2.3 Evaluation

Evaluation of this work will primarily be based on the following criteria:

Efficiency - The efficiency of the final algorithm generated by the platform would be evaluated for both time and space complexity. Where possible time and space locality will be exploited in caching in order to optimize these complexities.

Accuracy - The evaluation of accuracy or predictive performance will vary from model to model. The aim here is to achieve a level of performance comparable to that achieved by existing algorithms/techniques. Where applicable metrics such area under receiver operating characteristic curve (AUC-ROC), mean square error and mean absolute error will be used to quantify accuracy.

Models - A set of models that cannot be automatically generated by existing probabilistic programming languages has been identified. These models will be used for testing evaluation. There are many different ways of developing code for a given algorithm so automatic inspections or comparison of code is not feasible. Therefore we can do human inspection of code/algorithm for comparison, or runtime performance comparisons. The results of these experiements can be found in Chapter 7.

Evaluating Variants - Different variants for a particular model can be developed by the system. In general, however, there is no theory to easily say which variant my be superior in a particular context: multi-core, GPU, distributed computing, streaming data, small samples, etc. Therefore we demonstrate how variants can be automatically generated, by providing generated examples of the code for different variants. Generated code for different variants is provided in Chapter 7. This code can then be used for complexity analysis, which will inform users about the best situation to use a particular variant in.

Chapter 2

Probability and Bayesian Modelling

This chapter provides a brief review of basic probability theory to support the thesis. First consider Bayesian modelling. This is a basic review of notation and terms, and more detail can be seen in text books such as [24]. This chapter briefly covers the Bayes Theorem, discrete distributions, conjugate priors, augmentation methods and Gibbs sampling.

2.1 Bayesian Modelling

For probabilistic and Bayesian modelling, one has a data sample to analyse $\vec{x}$, and a parameterised probability model $p(\vec{x}|\vec{\theta})$ which is referred to as the *data distribution* or *data likelihood*, and multiple data items are assumed to be independently and identically distributed. Thus for a sequence of data $\vec{x}_1, ..., \vec{x}_I$ represented as $\mathbf{X}$, the full data likelihood becomes $p(\mathbf{X}|\vec{\theta}) = \prod_{i-1}^{I} p(\vec{x}_i|\vec{\theta})$.

The task of Bayesian analysis is to estimate the unknown parameter $\vec{\theta}$, more precisely infer its *posterior distribution*, $p(\vec{\theta}|\mathbf{X}, \vec{\alpha})$ which is constructed from the *prior distribution* $p(\vec{\theta}|\vec{\alpha})$ and the *data likelihood* using the Bayes Theorem. The parameters of the prior distribution, $\vec{\alpha}$, are usually treated as a hyper-parameter of the model.

With Bayes' theorem, the posterior distribution can be derived as:

$$p(\vec{\theta}|\mathbf{X}, \vec{\alpha}) = \frac{p\left(\mathbf{X}|\vec{\theta}\right) p(\vec{\theta}|\vec{\alpha})}{p(\mathbf{X}|\vec{\alpha})} = \frac{p(\mathbf{X}|\vec{\theta})p(\vec{\theta}|\vec{\alpha})}{\int p(\mathbf{X}|\vec{\theta'})p(\vec{\theta'}|\vec{\alpha})\mathrm{d}\vec{\theta'}}, \tag{2.1}$$

where $p(\mathbf{X}|\vec{\alpha})$ is the *marginal distribution* with the parameter $\vec{\theta}$ marginalised/integrated out.

After the model parameters are estimated, the standard use of a Bayesian model is to apply it to a new sample $\vec{x}^*$ to make predictions:

$$p(\vec{x}^*|\mathbf{X}, \vec{\alpha}) = \int p(\vec{x}^*|\vec{\theta})p(\vec{\theta}|\mathbf{X}, \vec{\alpha})\mathrm{d}\theta,$$

where the integral is often hard to compute so instead approximations are used. Selecting data and prior distributions, and constructing the connections of those distributions are the main tasks of *Bayesian modelling*. After a model is built, the training of the model is about the inference of the parameters by inferring their posterior distributions, referred to as *Bayesian inference*. The primary task of more complex Bayesian modelling is to approximate the unknown data likelihood $p(\vec{x}|\vec{\theta})$ by performing approximate Bayesian inference. The technique used in this thesis is Gibbs sampling, introduced later in this chapter.

2.2 Probability Distributions for Discrete Data

This thesis focuses on Bayesian inference for discrete data, so data distributions for discrete data are reviewed here. Observations of a sample is V dimensional vector, which can either be binary $\vec{x} \in \{0, 1\}^V$ or count-valued $\vec{x} \in \mathbb{N}^V$, where $\mathbb{N} = \{0, 1, 2, \cdots\}$. The data collection consists of N iid samples, denoted as $\mathbf{X} = [\vec{x}_1, \cdots, \vec{x}_N]$, meaning that $\mathbf{X}$ is a V by N discrete matrix and $\vec{x}_i$ is its i^{th} column.

If an entry x_j of data vector $\vec{x}$ is a binary variable, a natural model is the Bernoulli distribution.

$$x_{i,j} \sim \text{Bern}(\theta_j) \ ,$$

where usually the scalar θ_j would be different per dimension j. One may also use a vector version

$$\vec{x}_i \sim \text{Bern}(\vec{\theta}) \ ,$$

and these are used below where the distribution is not naturally in a vector form.

If $\vec{x}$ is a count-valued vector, one can use the Poisson, negative-binomial, or multinomial distributions as the data distribution.

$$\begin{aligned}
\vec{x} &\sim \text{Pois}(\vec{\theta}), \\
\vec{x} &\sim \text{NB}(\vec{r}, \vec{p}), \\
\vec{x} &\sim \text{Multi}(x_\cdot, \vec{\theta}),
\end{aligned}$$

TABLE 2.1: Conjugate priors

Data distribution	Prior distribution	Posterior distribution
$x \sim \text{Bern}(\theta)$	$\theta \sim \text{Beta}(\alpha, 1/\beta)$	$\theta \sim \text{Beta}(\alpha + x, \beta + 1 - x)$
$x \sim \text{Pois}(\theta)$	$\theta \sim \text{Gamma}(\alpha, 1/\beta)$[1]	$\theta \sim \text{Gamma}(\alpha + x, 1/(\beta + 1))$
$x \sim \text{NB}(r, p)$	$p \sim \text{Beta}(\alpha, \beta)$	$p \sim \text{Beta}(\alpha + x, \beta + r)$
$\vec{x} \sim \text{Cat}(\vec{\theta})$	$\vec{\theta} \sim \text{Dir}(\vec{\alpha})$	$\vec{\theta} \sim \text{Dir}(\vec{\alpha} + \vec{x})$
$\vec{x} \sim \text{Multi}(x., \vec{\theta})$	$\vec{\theta} \sim \text{Dir}(\vec{\alpha})$	$\vec{\theta} \sim \text{Dir}(\vec{\alpha} + \vec{x})$
$x \sim \text{Gamma}(\alpha, 1/\beta)$	$\beta \sim \text{Gamma}(\alpha_0, 1/\beta_0)$	$\beta \sim \text{Gamma}(\alpha_0 + \alpha, 1/(\beta_0 + x))$

where for Poisson, the rate $\vec{\theta} \in \mathbb{R}_+^V$ and $\mathbb{R}_+ = \{x : x \geq 0\}$; for negative-binomial, $\vec{r} \in \mathbb{R}_+^V$ and $\vec{p} \in (0, 1)^V$; for multinomial, $\vec{\theta}$ is a probability vector; and total count $x. = \sum_{v=1}^{V} x_v$ is assumed given.

The comparisons between the above three choices can be summarised as follows:

1. The Poisson and negative-binomial distributions are scalar distributions so vectorised in the above presentation.
2. When the total count of a data vector $x.$ is known, $\vec{x} \sim \text{Pois}(\vec{\theta})$ is equivalent to $\vec{x} \sim \text{Multi}(x., \vec{\theta}/\theta.)$.
3. The negative binomial is an extension of the Poisson. The Poisson distribution does not allow the variance to be adjusted independently from the mean, while the negative-binomial distribution consist of one more parameter to model the data variance independently.

2.2.1 Conjugate Prior Distributions

Prior distributions are needed for the above discrete distributions. In practice, *conjugate priors* are frequently used. These enable the posterior to have the same algebraic form as the prior, which significantly reduces the complexity of inference by bypassing the computation of the integral in Eq. (2.1). This means ease of forming a posterior, often times ease in sampling, and also a fairly straight forward interpretation of inference. Standard conjugate priors used in this thesis are given in Table 2.1

Consider the Bernoulli and beta as an example. The prior distribution is the beta, $\theta \sim \text{Beta}(\alpha, \beta)$. After observing one sample x, the posterior becomes $\theta \sim \text{Beta}(\alpha + x, \beta + 1 - x)$. A data sample updates the model parameter with Bayes' theorem, but does so with a simple change to the parameters of the prior. Where n positive data are observed, and m negative, then the posterior would be $\theta \sim \text{Beta}(\alpha + n, \beta + m)$. If fewer samples are observed or the data space is sparse, the prior would have a stronger influence on the posterior, while the uncertainty of the posterior is reduced when more samples are observed.

Note that not all parameters in these conjugate priors have simple distributions. The beta and Dirichlet distributions and the first argument to the gamma distribution do not have natural conjugate priors. Moreover, they are critical for the basic discrete distributions, especially when doing hierarchical modelling. To address this problem, the method of data augmentation is introduced next.

2.2.2 Non-parametric distributions

Non-parametric models have been increasingly used in recent machine learning approaches. While there is an extensive literature on non-parametric models and their computation in the statistics community [34, 37] and coming out of the machine learning community [53, 54], for the purposes of this thesis we can use a finite approximation that avoids the involvement of non-parametric theory.

So, for example, the Dirichlet process can be approximated by a Dirichlet as follows. Let $H()$ be the base distribution for the Dirichlet process, and let α be the concentration parameter, and let K be an integer. We use a K-dimensional Dirichlet to approximate the Dirichlet process.

$$\begin{aligned} \mu_k &\sim H() && \text{for } k \text{ in } 1, ..., K \\ \vec{\pi} &\sim \text{Dirichlet}_K\left(\frac{\alpha}{K}, \ldots, \frac{\alpha}{K}\right) \\ \mu \sim G &= \sum_{i=1}^{K} \pi_k \delta_{\mu=\mu_k} \end{aligned}$$

Here, G is approximately a sample from the Dirichlet process as $K \to \infty$. This technique is used extensively in the research community.

Similar approximations exist for the Gamma process. Thus in this thesis, non-parametric methods can be avoided. Arguably, this is not the most efficient or effective way to do non-parametric computation, but it is adequate.

2.3 Data Augmentation

Conjugacy makes inference simple but it also limits the flexibility of building Bayesian models. The technique of *data augmentation* can be used with the prior distributions above to convert a non-conjugate distribution into a simple form, which may be conjugate.

A full worked example is given with the Poisson-Gamma-Gamma model in the next subsection, but consider the basic form. A simple hierarchical model might take the

form $p(x|\theta)$ and $p(\theta|\beta)$, where x is the data, θ is a model parameter and β is a hyper-parameter. If the likelihood $p(x|\theta)$ as a function of θ is not conjugate to the prior $p(\theta|\beta)$, then complications can arise in estimation. Augmentation can sometimes address this. A new variable is introduced, t, and assume it is discrete for now, with a joint distribution $p(x,t|\theta)$, which must have the following properties:

- it marginalises out back to the original model, $p(x|\theta) = \sum_t p(x,t|\theta)$,
- the joint likelihood $p(x,t|\theta)$ has a simpler functional form in θ, and is now (hopefully) conjugate to $p(\theta|\beta)$, and
- the conditional $p(t|x,\theta,\beta)$ can be sampled efficiently.

In this case, we can sample t in an MCMC step and then go on and estimate or sample θ.

An example of augmentation is given below. The Student t-Distribution is used to model the marginal distribution of the mean x of a Gaussian with unknown variance. Augmenting the model with variable λ with strategic choice of parameters and distribution sets up the 2 terms in red to get cancelled out, leading to a simplification of the model.

$$p(x|\nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}\left(1+\frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

Introduce $\lambda \sim \text{gamma}\left(\frac{\nu+1}{2}, 1+\frac{x^2}{\nu}\right)$:

$$\begin{aligned} p(\lambda|\nu,x) &= \frac{\left(1+\frac{x^2}{\nu}\right)^{\frac{\nu+1}{2}}}{\Gamma\left(\frac{\nu+1}{2}\right)}\lambda^{\frac{\nu-1}{2}}e^{-\left(1+\frac{x^2}{\nu}\right)\lambda} \\ p(\lambda,x|\nu) &= \frac{1}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}\lambda^{\frac{\nu-1}{2}}e^{-\left(1+\frac{x^2}{\nu}\right)\lambda} \\ x|\lambda,\nu &\sim \text{Gaussian}\left(0,\frac{\nu}{2\lambda}\right) \end{aligned}$$

Note, however, this is not data augmentation but parameter augmentation, but the behavior is similar.

2.3.1 The Poisson-Gamma-Gamma Model

Now we consider the following hierarchical Bayesian model:

$$\begin{aligned}\alpha &\sim \text{Gamma}(a_0, 1/b_0),\\ \beta &\sim \text{Gamma}(c_0, 1/d_0),\\ \theta &\sim \text{Gamma}(\alpha, 1/\beta),\\ x &\sim \text{Pois}(\theta).\end{aligned}$$

Given the Poisson-Gamma conjugacy in Table 2.1, consider the posterior for θ, α, β given the single data x. The joint probability illustrates conjugacy:

$$p\left(x, \theta | \alpha, \beta\right) = \frac{\theta^x}{x!} e^{-\theta} \frac{\beta^\alpha}{\Gamma(\alpha)} \theta^{\alpha-1} e^{-\beta\theta}$$

We can show that θ's posterior given x is also a gamma: $\theta | x, \alpha, \beta \sim \text{Gamma}\left(\alpha + x, 1/(\beta+1)\right)$. This gamma posterior is conjugate to the gamma prior of β, which is the scale parameter. The posterior of β given x, θ is a gamma distribution as well: $\beta | x, \theta \sim \text{Gamma}(c_0 + \alpha, 1/(d_0 + \theta))$. But there is no ready form for the posterior of α due to the term $\Gamma(\alpha)$.

Marginalising out θ yields a probability for x given α and β:

$$p\left(x | \alpha, \beta\right) \propto \frac{\Gamma(\alpha + x)}{\Gamma(\alpha)} \frac{\beta^\alpha}{(\beta+1)^{\alpha+x}} \ . \tag{2.2}$$

This is also not conjugate to the gamma prior of α because of the ratio $\frac{\Gamma(\alpha+x)}{\Gamma(\alpha)}$. This ratio is called the *Pochhammer symbol* and sometimes a rising factorial, and denoted by $(\alpha)^x$ [64].

So sampling for β and θ given x can be done efficiently. But there is no efficient way to sample for α given x, regardless of whether θ is marginalised out or not, and general purpose but slow schemes need to be used such as Metropolis-Hastings.

Fortunately, the Pochhammer symbol can be augmented with an auxiliary variable t: $\frac{\Gamma(\alpha+x)}{\Gamma(\alpha)} = \sum_{t=0}^{x} S_t^x \alpha^t$ where S_t^x indicates an unsigned Stirling number of the first kind [13, 55]. With t, Eq. (2.2) can be augmented as follows. First reexpress the initial probability.

$$\begin{aligned} p\left(x | \alpha, \beta\right) &\propto \sum_{t=0}^{x} S_t^x \alpha^t \frac{\beta^\alpha}{(\beta+1)^{\alpha+x}} \\ &= \sum_{t=0}^{\infty} \delta_{t \le x} \, S_t^x \, \alpha^t \frac{\beta^\alpha}{(\beta+1)^{\alpha+x}} \\ &\propto \sum_{t=0}^{\infty} p\left(x, t | \alpha, \beta\right) \end{aligned}$$

Now match the terms:

$$p\left(t|x,\alpha,\beta\right) \;\propto\; \delta_{t\leq x}\, S_t^x \alpha^t \;.$$

Moreover, the likelihood $p\left(x,t|\alpha,\beta\right)$ as a function of α is now conjugate to the prior distribution for α because it is in a gamma form.

Therefore, the posterior of α given x,t can be written as:

$$\alpha|x,t,\beta \sim \text{Gamma}\left(a_0+t, 1/\left(b_0+\log\frac{\beta}{\beta+1}\right)\right). \tag{2.3}$$

Now to work with this neat gamma distribution for α we first need to sample t from $t\sim p\left(t|x,\alpha\right)$, where

$$p\left(t|x,\alpha\right) = \frac{S_t^x \alpha^t}{\sum_{t=0}^{x} S_t^x \alpha^t}$$

Fortunately, the above Pochhammer symbol is the normalisation term of the posterior of another distribution called the Chinese Restaurant Process [55] with α as its concentration parameter, x as the number of customers, and t as the number of tables assigned for those customers. For the case above only the number of tables t is needed, and the distribution of t is called the Chinese Restaurant Table (CRT) distribution by [70].

$$t|x,\alpha \sim \sum_{i=1}^{x} \text{Bern}\left(\frac{\alpha}{\alpha+i-1}\right), \tag{2.4}$$

where $\frac{\alpha}{\alpha+i-1}$ is the probability of opening a new table for the i^{th} customer. Thus $p\left(t|x,\alpha\right)$ can be sampled efficiently using this sum of Bernoulli variables.

Refer to this augmentation as the *CRT augmentation*. These various formula then get pieced together to obtain a sampler for α given x and β as follows:

1. sample $t|x,\alpha$ using the CRT distribution of Equation (2.4),
2. sample $\alpha|x,t,\beta$ using the gamma distribution of Equation (2.3)

This CRT augmentation has been used extensively to build hierarchical models with gamma distributions for Poisson-distributed data [33, 64, 66, 68, 71, 72].

2.3.2 The Multinomial-Dirichlet-Gamma Model

Consider the following model:

$$\begin{aligned}
\alpha_v &\sim \text{Gamma}(a_0, 1/b_0),\\
\vec{\theta} &\sim \text{Dir}(\vec{\alpha}),\\
\vec{x} &\sim \text{Multi}(x_\cdot, \vec{\theta}),
\end{aligned}$$

where $\vec{x} \in \mathbb{N}^V$, $\vec{\theta}$ is a probability vector, and $\vec{\alpha} \in \mathbb{R}_+^V$. Given the Multinomial-Dirichlet conjugacy in Table 2.1, θ's posterior is also Dirichlet:

$$\vec{\theta} \sim \text{Dir}(\alpha + \vec{x}).$$

Marginalising out $\vec{\theta}$ yields:

$$p\left(\vec{x}|\vec{\alpha}\right) \propto \frac{\Gamma(\alpha_\cdot)}{\Gamma(\alpha_\cdot + x_\cdot)} \prod_{v=1}^{V} \frac{\Gamma(\alpha_v + x_v)}{\Gamma(\alpha_v)}.$$

The CRT augmentation deals with the product of right-hand side gamma ratios. The left-hand side ratio can be manipulated with a beta distribution.

For the left-hand side gamma ratio, introduce a beta distributed auxiliary variable $p|x, \alpha_\cdot \sim \text{Beta}(\alpha_\cdot, x_\cdot)$ and augment the ratio like so:

$$\frac{\Gamma(\alpha_v)}{\Gamma(\alpha_\cdot + x_\cdot)} \propto \int_p p^{\alpha_\cdot - 1}(1-p)^{x_\cdot - 1}. \tag{2.5}$$

For the right-hand side gamma ratios, for each v introduce an auxiliary CRT variable t_v given x_v, α_v using Equation (2.4).

This becomes:

$$p\left(\vec{x}, \vec{t}, p|\vec{\alpha}\right) \propto \prod_{v=1}^{V} p^{\alpha_v} \alpha_v^{t_v} = \prod_{v=1}^{V} \alpha_v^{t_v} e^{-\log \frac{1}{p} \alpha_v},$$

which as a likelihood for $\vec{\alpha}$ is conjugate to the gamma prior of each α_v. The posterior of α_v given t_v, p is:

$$\alpha_v | t_v, p \sim \text{Gamma}\left(a_0 + t_v, 1/\left(b_0 + \log \frac{1}{p}\right)\right). \tag{2.6}$$

Refer to this technique as the beta-CRT augmentation. In summary, the steps are:

1. for each v, sample $t_v|x_v, \alpha_v$ using the CRT distribution of Equation (2.4),
2. sample $p \sim \text{Beta}(\alpha_\cdot, x_\cdot)$.
3. sample $\alpha|\vec{t}, p$ using the gamma distribution of Equation (2.6)

TABLE 2.2: Auxiliary variable samplers for terms in β.

FORM	AUXILIARY VARIABLES	NEW FORM
$(I+\beta)^{-\alpha}$	$q \sim \Gamma(\alpha, I+\beta),\ \alpha, I > 0$	$e^{-q\beta}$
$\Gamma(\beta+n)/\Gamma(\beta)$	$t \sim \text{CRT}(\beta, n),\ n \in \mathcal{Z}^+$	β^t
$\Gamma(\beta)/\Gamma(\beta+\alpha)$	$q \sim \text{be}(\beta, \alpha),\ \alpha, \beta > 0$	q^β
$\beta(\beta+\alpha)...(\beta+(n-1)\alpha)$	$t \sim \text{Poch}(\alpha, \beta, n),\ n \in \mathcal{Z}^+$	β^t
$\alpha(\alpha+\beta)...(\alpha+(n-1)\beta)$	$t \sim \text{Poch}(\beta, \alpha, n),\ n \in \mathcal{Z}^+$	β^{n-t}

Note that in addition to beta-CRT, the CRT augmentation can also be applied to model a vector of count-valued data:

$$\begin{aligned} \alpha_v &\sim \text{Gamma}(a_0, 1/b_0), \\ \beta &\sim \text{Gamma}(c_0, 1/d_0), \\ \vec{\theta} &\sim \text{Gamma}(\vec{\alpha}, 1/\beta), \\ \vec{x} &\sim \text{Pois}(\vec{\theta}), \end{aligned}$$

where $\vec{x} \in \mathbb{N}^V$.

The major difference between CRT and beta-CRT for modelling a data vector is that in the former one, $\vec{\theta}$ is unnormalised, while it is normalised probability vector in the latter one. The beta-CRT augmentation has also been heavily-used in building hierarchical models based on the multinomial-Dirichlet [64, 66, 67].

2.3.3 Different Augmentations

Examples of some known augmentations used when Bayesian modelling of discrete data, as collected from previously mentioned Gibbs sampling papers, are given in Table 2.2. To see how this would be used consider the following scenario: Suppose you wanted to sample

$$p(\beta|n) \propto \beta^{(n)} e^{-\beta} ,$$

for $n \in \mathcal{Z}^+$. Now this is close to a gamma distribution, but the term $\beta^{(n)}$ is the problem. It matches the second row in the table under FORM. To sample according to this, sample as described in the second column, $t \sim \text{CRT}(\beta, n)$, and now the probability is transformed by replacing FORM with NEW FORM.

$$p(\beta|n, t) \propto \beta^t e^{-\beta} ,$$

which is now easy to sample. Likewise, in any functional form we wish to sample from, the augmentations listed in column 2 let us transform, FORM into NEW FORM.

An algorithm for sampling from CRT(β, n) simulates sequential table sampling from a Chinese restaurant process, and is given in Algorithm 2. This cleans up the algorithm in Equation (2.4).

CRT-sample(β, n)**:**
// $n \in \mathcal{Z}^+$, $\beta > 0$
$t = 1$ // first data always opens table
for $i = 1$ *to* $n - 1$ **do**
  $t = t +$ **Bern** $\left(\frac{\beta}{\beta+i}\right)$ // does new data open a table?
end
return t //distributed proportional to $\beta^t S_t^n$

Algorithm 2: Simple sampling tables for a CRT(β, n)

An algorithm for sampling a Pochhammer symbol, Poch(α, β, n), is similar to the CRT algorithm, and given in Algorithm 3.

Pochhammer(α, β, n)**:**
// $n \in \mathcal{Z}^+$, $\alpha, \beta > 0$
$t = 0$
for $i = 0$ *to* $n - 1$ **do**
  $t = t +$ **Bern** $\left(\frac{\beta}{\beta+\alpha i}\right)$
end
return t

Algorithm 3: Sampling a Pochhammer symbol

2.4 Bayesian Inference

Bayesian inference is used to estimate the parameters of a Bayesian model using their posterior distributions. The posterior of the model parameter $\vec{\theta}$ can be obtained by Bayes' theorem, Eq. (2.1). But to exactly compute the posterior, we need to calculate the integral in the marginal distribution, $p(\mathbf{X}|\vec{\alpha})$ which is usually intractable.

To deal with this, approximate Bayesian inference algorithms are used. Two of the most commonly-used approximate Bayesian inference techniques are *Markov Chain Monte Carlo* (MCMC) sampling and *variational inference*. The former approach obtains (approximate) samples from the posterior. The latter approach approximates the posterior with a variational distribution. These two are a broad family of methods with much recent research.

This thesis works with a specific type of MCMC sampling called Gibbs sampling. It is often a simple method to implement and also a sufficiently general method to allow substantial application, for instance in recent machine learning algorithms [33, 64, 66,

68, 71, 72], in one of the first probabilistic programming approaches BUGS, and also supported by some distributed processing methods [8, 35].

2.4.1 Gibbs Sampling

Gibbs sampling [25], which forms the basis for BUGS, is a special case of a very general MCMC method called Metropolis-Hastings sampling. For this the conditional posterior distribution of each dimension of θ, θ_v, is used to resample θ_v. So

$$\theta_v^* \sim p\left(\theta_v^* | \vec{\theta}_{\neg v}, \mathbf{X}\right).$$

where $\vec{\theta}_{\neg v}$ is the vector $\vec{\theta}$ with the scalar value θ_v^* removed. Gibbs sampling works best when this conditional posterior has a simple form, which usually happens in conjugate models. Some models maybe partially conjugate, as is the case for the Multinomial-Dirichlet-Gamma model above. So for these, data augmentation can work well.

Using Gibbs sampling (or other MCMC sampling algorithms) one collects samples from the posterior, and these can be used to characterise the posterior. So a number of Gibbs samples may be run to "burn-in" the sampler and thereafter samples are retained. We then use the samples to compute the following properties of the posterior. Let $\vec{\theta}^{(s)}$ be the s^{th} collected sample. Then

$$\vec{\theta} \approx \sum_{s=1}^{S} \vec{\theta}^{(s)}/S \qquad \text{(posterior mean)}$$

$$p\left(\vec{x}^* | \mathbf{X}\right) \approx \sum_{s=1}^{S} p\left(\vec{x}^* | \vec{\theta}^{(s)}\right)/S \qquad \text{(predictive distribution)}$$

2.4.2 Operations in Gibbs Sampling

The two main classes of operations used in developing efficient Gibbs samplers are collapsing and augmentation, covered next.

2.4.2.1 Collapsing Operations

Collapsing (or marginalization) is the process of integrating over some or all parameters of the model and may be associated with a reduction of the data to sufficient statistics of the marginal model. If the marginalized distribution is tractable, it can then be directly used in inference. Figure 2.1 denotes an example of marginalization as applied to a graphical structure, where the new node 's' has been marginalized out of the graph leading to a new connection between nodes 'lc' and 'b'. In the discrete case,

marginalizing involves summing over the possible states of the marginalized variable and in the continuous case, the variable would be integrated out instead.

For the example in Figure 2.1, s is being marginalised out. The new set of variables thus becomes $X/\{s\}$, and the following formula achieves the marginalisation:

$$p(X/\{s\}) = \sum_{s \in values(s)} p(X)$$

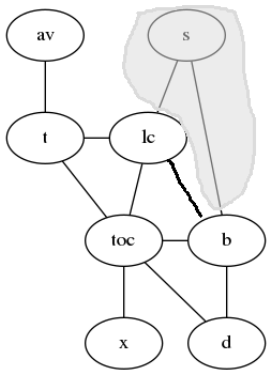


FIGURE 2.1: Collapsing

2.4.2.2 Augmentation Operations

Data augmentation is the inverse operation of marginalization. Data augmentation involves adding parameters to the model to make it more tractable. This is particularly helpful when the complexity of adding parameters is less significant than the benefits of simplifying the distributions of the model. Figure 2.2 denotes an example of augmentation as applied to a graph, where the new node 'u' has been introduced into the graph and has been connected with 't','lc' and 'toc', due to being conditioned on them.

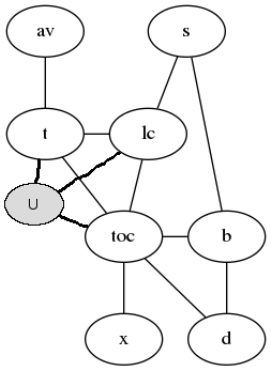


FIGURE 2.2: Augmentation

In general, augmentation is done strategically in order to simplify the model. In this case, let X denote the nodes in the graph except u. The new joint distribution $p(u, X)$

must satisfy

$$p(X) = \sum_{u \in values(u)} p(u, X) \ .$$

Moreover, we know from the shape of the undirected graph that

$$\frac{p(u, X)}{p(X)} = p(u|X) = p(u|t, lc, toc) \ ,$$

and thus

$$p(u, X) = p(X)p(u|t, lc, toc) \ .$$

Examples of some known augmentations were given in Table 2.2.

2.5 Machine Learning with Gibbs Sampling

Researchers in the machine learning community are using Gibbs sampling particularly for unsupervised learning in discrete domains, for which the method is particularly suited [12, 13, 18, 19, 33, 38, 55, 64, 64, 66–68, 70–72]. The majority of these methods do augmentation and collapsing to achieve efficiency. In the area of non-parametric topic models, for instance, superior results [6] with similar speed to sophisticated distributed variational methods were obtained by using a technique called hogwild Gibbs [35] that allows the use of multi-core processing. For all of these algorithms, existing methods in the probabilistic programming community are unable to support development of corresponding efficient algorithms.

Chapter 3

Literature Review

This section provides an overview of the prior work relevant to this thesis.

The main body of literature is present under probabilistic programming, which relates to work presenting abstractions in the area of statistics and machine learning, to enable users to write code that builds models and performs training or inference with these probabilistic constructs. A number of influential probabilistic programming languages have been reviewed, with particular attention paid to aspects such as language design, statistical abstractions, symbolic support and robustness. The review covers key aspects of classic probabilistic programming languages and also provides a brief overview of recent developments.

Symbolic processing is another area that influences this thesis, as the developed system is very reliant on the symbolic primitives present in this area of computer science. The relevant work from this area can be divided into symbolic systems and pattern matching algorithms. Symbolic systems provide general symbolic support for use in a plethora of use cases. Pattern matching algorithms are primarily focused on performing symbolic search based on symbolic structures.

3.1 Probabilistic Programming

Probabilistic programming [29, 47] is a programming paradigm introduced with the objective of enabling the description of probabilistic models using simple syntax and then performing inference on those models. While most probabilistic programming languages extend other basic programming languages (such as Java), several others (such as BUGS and Stan) offer their own declarative formulation for program syntax. Declarative programming languages focus on what needs to happen, rather than how it happens. This is distinct from imperative languages which focus on instructing the

compiler step by step how program execution should occur. Imperative languages offer more control to the programmer while declarative languages are more user friendly.

At present, a wide variety of probabilistic programming languages are available. Nearly all probabilistic languages are Bayesian, simply due to the difficulty of creating other frameworks for automated reasoning about general models. Among the Bayesian probabilistic programming languages, some (BUGS [41], STAN [10], AutoBayes [22], JAGS [48] and Infer.NET [44]) allow only a restrictive subset of models to be represented compared to frameworks based on more general programming languages such as IBAL [45], BLOG [43], Figaro [46], and Church [28]. This is a trade-off between flexibility and efficiency, as the restricted languages are much faster, in general, compared to the more flexible languages [26]. In this review, we will be primarily evaluating the more restrictive class of languages, as scalability is a primary requirement of the system under development.

From a probabilistic programming perspective, declarative languages have the distinct benefit of allowing users less versed in programming to execute their own statistical models with relative ease, while allowing expert users to rapidly prototype their models using the innate flexibility provided by not needing to specify standard distribution semantics and axioms related to probability and statistics.

This section introduces several probabilistic programming languages that have seen great use over the past decade and offers a critical analysis of various aspects of the language design and its impact on usability, efficiency and extensibility.

3.1.1 BUGS

BUGS is a probabilistic programming language for performing Bayesian inference using Gibbs sampling. BUGS is written in Component Pascal, a lesser known programming language [40]. While Bayesian analysis is commonplace at present, BUGS was introduced in an era where techniques based on simulation were rare. While Bayesian methodologies were still applied, they were mostly viable in situations where closed form solutions could be derived using statistical conjugacy, or when numerical integration techniques could be applied to the problem at hand. The introduction of BUGS led to the implementation of Bayesian methods in a wide range of fields including ecology, actuarial science, genetics and sports modelling. BUGS currently exists as 2 projects WinBUGS[41] and OpenBUGS[50].

A key aspect of BUGS is the flexibility it provides as a consequence of graphical modelling. BUGS uses a directed acyclic graph (DAG) to represent a model, with every vertex in the DAG corresponding to a variable in the model and each directed link representing the relation of direct dependence.

As an example, the code below denotes a linear regression example in BUGS. Note that the $\sim$ symbol denotes sampling from a distribution. The corresponding DAG is given in Figure 3.1.

```
model {
    for (i in 1:N) {
        y[i] ~ dnorm(mu[i], tau)
        mu[i] <- alpha + beta * x[i]
    }
    alpha ~ dnorm(m.alpha, p.alpha)
    beta ~ dnorm(m.beta, p.beta)
    log.sigma ~ dunif(a, b)
    sigma <- exp(log.sigma)
    sigma.sq <- pow(sigma, 2)
    tau <- 1 / sigma.sq
    p.alpha <- 1 / v.alpha
    p.beta <- 1 / v.beta
}
```

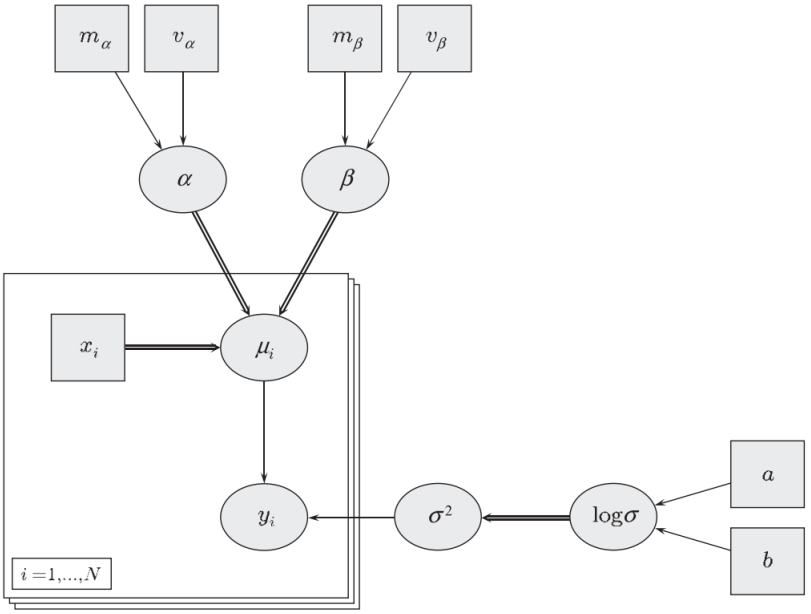


FIGURE 3.1: BUGS Linear Regression example

The joint probability distribution for all vertices in a DAG can be given by:

$$p(V) = \prod_{v \in V} p(v \mid P_v)$$

where P_v denotes the parents of vertex v.

In the context of Gibbs sampling, we require the full conditional distribution of each vertex conditional on the values of all other vertices. Let u be any node in the graph. then:

$$p(u \mid V \setminus u) \propto p(u \mid P_u) \times \prod_{v \in C_u} p(v \mid P_v)$$

where P_v and C_v denote the parents and children of vertex v respectively. This follows from the Markov blanket property for a Bayesian network, which states that the Markov blanket for a node A in a Bayesian network is the set of nodes composed of A's parents, its children, and its children's other parents (Figure 3.2).

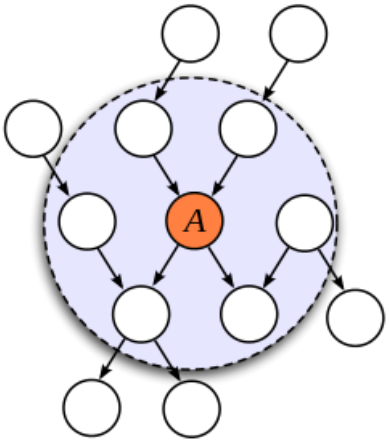


FIGURE 3.2: Markov Blanket

This factorization has 2 implications:

1. Computation of the joint posterior for a DAG requires only the relationship between each node and its parents.
2. The full conditional distribution for any vertex or set of vertices depends only on its Markov blanket and is thereby a local property of the graph.

In combination, this means that each vertex only needs to store information about how it is related to its Markov blanket and any computations involving a node are also local computations which do not require knowledge of the structure of the rest of the graph.

The BUGS Language

BUGS is a declarative language for representing graphical models using code. This means that the user simply defines the relationships between nodes and provides the structure and properties of the graph, but does not have as much control (compared to a regular programming language) over the actual execution of the generated code. This is highlighted by the language's lack of an (if,then,else) construct which is a standard structure in most traditional programming languages.

While the declarative nature of BUGS allows for rapid prototyping of complex statistical models, it is important to note the limitations of doing so. The design philosophy of BUGS is based on the software engineering concepts of abstraction and encapsulation. When considering the operation of the generated graphical structure from the perspective of Gibbs sampling, it is sufficient to abstract out the key details. As each vertex is visited, its relevant contribution to the full conditional is updated, and the process continues by navigating to its child vertices. This allows for decoupling of the distributions which the nodes represent. BUGS uses this property to allow for a component oriented approach. Statistical distributions, functions and even updating algorithms can be added or removed with very high flexibility, allowing for an easy-to-use interface users to generate models.

A major disadvantage of this approach of abstraction becomes apparent when debugging and error-reporting is considered. Because of the modular way in which BUGS is set up, it is possible for a very large range of models to be defined within it. This allows for the possibility of many different vertex-vertex context-dependent interactions. Additionally, modelling components with no direct dependence might conflict with each other. While each component performs basic testing on itself and its Markov blanket, these checks are necessarily localized and are also context independent.

As models get more complex, this increases the likelihood that a very complex error will occur. Additionally, due to the complexity of the interactions, debugging information cannot generally be derived about the situation, leading to ambiguous error statements [16]. Especially when these occur during situations such as Markov Chain Monte Carlo simulations, it can be very difficult to pinpoint the source of the error. The BUGS user manual lists several work-arounds for such scenarios, including selecting different priors, changing initial values and re-parameterizing the model. However, this trial and approach system becomes inapplicable when the model is very complex and the number of parameters is high [56].

The design methodology behind BUGS should allow for a high degree of parallelism to be incorporated into its system. Multi-chain Gibbs sampling uses independent chains, with each chain requiring no communication with other chains and represents an easily parallelizable component. Additionally, performing updates may also be done in parallel to some extent, but would require proper handling of locks on nodes and race conditions. However, BUGS does not appear to exploit much of this parallelism [39].

3.1.2 AutoBayes

AutoBayes [22] is a framework for generating automated data analysis programs from statistical models. AutoBayes uses a schema-based system to perform program synthesis. Schema encode generic code in the form of a template and additionally specify applicability constraints on them. These are checked against the model using theorem proving techniques. AutoBayes supports symbolic and algebraic computations and attempts to derive closed-form solutions for most of its applications. The entire system has been developed in SWI-Prolog [60] but generates C/C++ code.

A crucial aspect of AutoBayes is that it incorporates symbolic reasoning into its automated workflow [49]. The choice of Prolog has been motivated by the system's requirement of symbolic reasoning along with reasoning over graphs and general purpose programming operations such as input/output. AutoBayes maintains information regarding all assumptions (for example, an expression being non-zero) and either discharges them during synthesis or generates code-level assertions which then have to be checked during run-time. A symbolic algebra system has not been used due to the lack of transparency, as operations done by such engines are not always visible in terms of the assumptions they use. During symbolic calculations, such simplifications can lead to possible unsoundness and hence incorrect programs since all assumptions may not be explicitly made available. This arises from the fact that a symbolic algebra system will have many different methods for simplifying symbolic expressions and may represent the same expression in different ways. For example, a probabilistic language built on a symbolic algebra system may always expect (a + b + c) to be represented as ((a + b) + c), but if the system represents it as (a + (b + c)), it may cause issues which can be considerably difficult to debug.

The architecture of AutoBayes is composed of several key components.

The **Synthesis kernel** is tasked with generating the initial Bayesian network based on the input model specification. The DAG is generated in a manner very similar to BUGS except for how loops and nested variables are handled. AutoBayes uses the technique of flattening, which removes nested variables and introduces new variables as replacements. This has the added implication of enlargening the DAG by adding more edges and vertices. For example, the declaration

```
x(i) ~ gauss(a(c(i)), b(c(i)))
```

introduces the three edges:

```
a(j) -> x(i)
```

```
b(j) -> x(i)
c(i) -> x(i)
```

In this situation note that although x and c indexed by i, each x(i) now depends on all a(:) and b(:), which portrays the unknown nature of the values of the indices c(i) at model construction time and additionally requires a new indexing variable j to be introduced.

Once the initial network is constructed, synthesis proceeds by applying schemas in an exhaustive manner using depth-first search with backtracking. This allows variants of programs to be generated - any search that terminates by correctly matching all required criteria is a valid program, and DFS with backtracking is guaranteed to find all such variants.

The schemas used predominantly fall into one of four categories:

Network decomposition schemas are composed of suitable encodings of independence theorems for Bayesian networks. These schema define the decomposition of a probabilistic inference task over a complex network into simpler tasks over less complex networks. This additionally informs the decomposition of the data analysis program into simpler components as well. The preconditions for these schema are generally checked by using graphical reasoning.

Formula decomposition schemas work on complex formulae in a similar manner to Network decomposition schemas but work on formulae instead.

Statistical algorithm schemas are graph schemas which are more complex than Network decomposition schemas. They tend to involve more moving parts, for example, storing the results of certain intermediate calculations. These require considerable symbolic reasoning during the initialization process.

Numerical algorithm schemas take over once graph based schema have been exhausted of conditional probabilities, and these schemas provide the means of setting up the appropriate distributions to be converted into an optimization problem. AutoBayes then attempts to solve this optimization problem by initially attempting to find a closed-form solution (using partial differentiation followed by solving simultaneous equations using gaussian variable elemination), followed by the alternative of using numerical techniques (such as Newton-Raphson) to obtain solutions.

3.1.3 JAGS

JAGS [48] stands for Just Another Gibbs Sampler. JAGS is a probabilistic language which uses Gibbs Sampling as its primary sampling technique. An R port of JAGS (R2JAGS [52]) is also available. JAGS was developed with 2 primary aims in mind.

- Creating a BUGS clone which could be modified.
- Developing a platform for exploring tools for criticizing statistical models.

The architecture of JAGS contains several key components. Graphs are characterized by different types of **nodes**.

1. Stochastic nodes - characterized by a distribution with parameters defined by its parents.
2. Logical nodes - directly calculated as a deterministic function of its parents.
3. Constant nodes - constant valued nodes.
4. Array Nodes - containers for a set of nodes.
5. Subset Nodes - generated using subscripting from other nodes. The size is fixed, but the subscripts may be stochastic.

Samplers define techniques for updating graphs. For example, Gibbs sampling acts on a single node. **Monitors** track sampled values and summarize them. **Models** are used to associate a particular graph with relevant samplers and monitors.

JAGS removes the concept of "for loops" used in BUGS, replacing it with **named dimensions** instead. As BUGS is a declarative language, using for loops does not lead to actual iteration but rather denotes to the parser that tiling (repetitive structure) is present for a certain model block. JAGS uses array nodes and requires users to define the size of an array node in terms of a previously defined dimension.

3.1.4 Stan

Stan [10] is another probabilistic programming language for the specification of statistical models. Stan programs define a log probability function over parameters conditioned on the provided data. Stan supports Bayesian inference for continuous-variable models using Markov chain Monte Carlo methods. Sampling is primarily done by the No-U-Turn sampler [32], which is a variant of Hamiltonian Monte Carlo (HMC) [20].

Stan was primarily developed to overcome run-time issues present in JAGS and BUGS, due to the slow convergence performance of Gibbs sampling in situations such as time-series modelling and hierarchical modelling with interacted predictors, which contain parameters with high correlation in the posterior.

Stan programs comprise of several specialized **blocks**:

- The **data block** is used for declaring the data for fitting the model. These declarations are used to compile the Stan program more efficiently, which is different from the approach taken by BUGS and JAGS, which use the technique of assigning variables as data or parameters at run-time.
- The **transformed data block** contains new variables which are defined based on transforming variables from the data block. The execution of this block is scheduled immediately after data input occurs, with constraint validation and associated error reporting.
- The **parameter block** similarly defines the parameters and types used in the model, but do not denote information about priors. This block will be executed each time the log probability is updated. The **transformed parameter block** defines new variables indicating transformations on defined parameters, and these variables are updated with each execution of the parameter block.
- The **model block** defines the log probability function by setting priors for the previously declared variables. Priors may also be defined over vectors, allowing values to be drawn independently of each other based on the distribution. The model block is run with each evaluation of the log probability function, immediately following the transformed parameters block.

The basic statements supported by Stan are equivalent to those supported by BUGS, but while BUGS defines a directed acyclic graph, Stan defines a log probability function, and computation of this is all that is required to the HMC algorithm. This is critical to understanding how Stan differs from BUGS. This log probability function is contained in an implicitly defined log probability accumulator, and a sampling step simply corresponds to an increment in this accumulator. This distinction from BUGS explains why variables need to be defined prior to use of sampling statements in Stan.

The use of the log probability representation has several key advantages:

- Speed - Multiplication is a more expensive operation than addition. Therefore, when multiplying many probabilities together (as is required in many cases where

independence is present), it is less computationally expensive to when the probabilities are represented in logarithmic format. While the initial conversion to log form requires some computation, this cost is only incurred once.

- Representation - Many probabilities are derived from distributions taking on exponential form. Passing these through a log function removes the exponential function and provides a simpler representation.
- Accuracy - The floating point representations used in computers are inexact by nature. Taking the log transform helps to improve numberical stability and improves the overall precision of the representation.

Stan is imperative in that it allows for sequencing of statements, unlike BUGS. This creates the possibility for the user to define an order for the statements to be executed and crucially allows for control structures such as "if-then-else" and "while".

3.1.5 Edward

Edward [57, 58] is a probabilistic programming language based in Python. The aim of the Edward library is to be enable rapid experimentation and the use of probabilistic models ranging over a variety of model complexities including hierarchical models and deep probabilistic models.

Edward supports the following types of models:

- Neural networks
- Generative models
- Directed graphical models

Edward aims to combine three fields: Bayesian machine learning, deep learning, and probabilistic programming. It provides support for a wide variety of inference techniques including:

- Variational Inference: including Black box variational inference, Stochastic variational inference and generative adversarial networks;
- black box MCMC methods running off the log probability (like Stan) including Hamiltonian Monte Carlo and Stochastic gradient Langevin dynamics;
- Gibbs sampling;
- compositional inference such as the message passing algorithm.

Edward is currently available as Edward2 [59] with support for TensorFlow or Numpy as the backend. Edward2 is also available through TensorFlow Probability[1].

3.1.6 Greta

Greta[2] is a probabilistic programming language aiming to achieve scalable statistical modelling in R. Greta models are written in R itself which makes it more accessible to users who are already familiar with R syntax. Greta uses TensorFlow Probability allowing models to scale using CPU clusters and GPU.

3.1.7 TensorFlow Probability

TensorFlow Probability[3] is a Python library aiming to provide the capability of easily combining machine learning models on GPUs. This library features:

- Support for a wide variety of probability distributions
- support for deep probabilistic models
- inference using variational methods and Markov chain Monte Carlo techniques.

3.1.8 Gen

Gen [17] is a probabilistic programming language that focuses on incorporating programmable inference. Gen's architecture incorporates a generative function interface which attempts to create decoupling between probabilistic languages' domain specific languages (DSLs) and their corresponding inference algorithm implementations. Here, DSLs refer to the language definitions used by probabilistic languages in order to abstract functionality provided by their system. Generally, probabilistic languages will also provide inference DSLs, allowing users to define the inference process using syntactical structures from these DSLs which will usually be more restrictive than a general purpose programming langauge.

Gen provides support for:

- Combining different domain specific languages within the same model
- Custom inference algorithm implementation by the user, while optionally relying on a provided standard inference library.

[1] https://blog.tensorflow.org/2018/04/introducing-tensorflow-probability.html
[2] https://cran.r-project.org/web/packages/greta/
[3] https://www.tensorflow.org/probability/

- The ability for users to extend the language (for example: by defining new domain specific language to plug-in).
- Support for the definition of specialized proposal distributions, which can incorporate problem specific knowledge into the inference process. These proposals can be processed automatically by the system with minimal user interaction.

Gen provides support for a number of inference techniques via its standard inference library, including Hamiltonian Monte Carlo and Metropolis-Hastings MCMC with support for custom proposals as well.

3.1.9 Shuffle

Shuffle [1] is a language for generating robust inference schemes. While many probabilistic programming languages support either automatic inference or manual inference algorithm creation (by providing a standard library, or other such abstractions), most do not validate the final outcome to ensure the correctness of the final algorithm or scheme.

Shuffle attempts to address this by creating a language for manual inference algorithm creation based on several key rules:

- The basic rules of probability must be respected.
- The statistical dependencies of the defined probabilistic model must be maintained in the inference process.
- The generated scheme must be optimized.

Shuffle provides support for inference by allowing users to specify models, where the underlying encapsulated implementation is one of the following:

- A probability density function
- A Monte Carlo sampler
- a Monte-Carlo Markov Chain Transition Kernel

Underlying support is present for developers to incorporate inference using Gibbs sampling and Metropolis-Hastings, among others.

3.1.10 AugurV2

AugurV2 is a compiler that operates on probabilistic models defined using a restricted language. It can support Bayesian networks where the graphical structure is fixed. This includes models such as topic models, mixture models and deep generative models, but does not support non-parametric distributions and models with undirected edges (e.g. Markov random fields). A key aspect of AugurV2 is the use of intermediate languages, which operates on the declarative specification and updates it with the aim of generating an executable inference algorithm for use on CPU or GPU.

3.1.11 Summary of probabilistic programming

BUGS uses a directed acyclic graph representation which we have adopted in our system along with a similar notation system for model specification. It uses Gibbs sampling as the statistical engine.

AutoBayes is a framework which incorporated symbolic reasoning into its workflow and used a schema based system for performing modifications of the model. A similar strategy was adopted in this system with some schema being explicitly available as operations and others being processed symbolically through modification of the likelihood or related representations therein. AutoBayes analyses the model DAG for simplifications and then matches on general schema like the EM algorithm of simple MAP computation.

JAGS, a BUGS extension, introduces a special system for "tiling" or "named dimensions" which incorporates the possibility of symbolically denoting the dimensions of a variable. We have adopted a similar approach and have extended the possibility for array indices to be sampled as well and have provided support for the corresponding complex updates which need to propagate throughout the model graph (see section 5.4.2.5).

Where BUGS defines a directed acyclic graph, Stan defines a log probability function which is contained in an implicit log probability accumulator, with a sampling step, using HMC, simply corresponding to an increment in this accumulator. Stan makes use of blocking within the model specification including separate data blocks, parameter blocks and model blocks, a design which has inspired our use of a similar system, although we have not used a seperate block for defining transformations of data, instead opting to identify the necessity of such transformations via symbolic reasoning (refer to section 5.3.3) and automatically computing and memoizing(caching) such quantities as needed (refer to section 5.4.2.7).

Edward and Greta use some schema approaches-based, like AutoBayes, but also better integrate with modern GPU computations. They have inspired further exploration into some of the future directions of our system such as parallelization and the possibility of

generating GPU-friendly versions of code. Gen, another recent addition, has introduced the concept of programmable inference, giving users more flexibility in term of the inference process. Shuffle introduces the concept of embedding probabilistic semantics into the algorithm generation context, while AugurV2 uses intermediate "languages" to refine declarative specifications.

Other recent approaches are similar, providing more appropriate programming language capability to the general probabilistic programming agenda, as well as supporting a broader variety of black-box inference readily supported from log probability and simple DAG/network computation and manipulation.

3.2 Symbolic Processing

Symbolic processing or symbolic computation is the discipline related to the development of algorithms enabling the manipulation of symbolic mathematical expressions and other symbolic structures. In general, Symbolic processing systems are used in software dealing with numerical programs and scientific computing. In order to support the more sophisticated operations (augmenting and collapsing) required for our system, Symbolic processing is a necessary capability and therefore a review of it is done here. However, note, we need open source code to properly interface with our system.

A general symbolic processing system requires the definition of two main capabilities:

Representation: The symbolic representation of quantities in the system determines the state space of expressions that can be represented in the system.

Simplification: A symbolic processing system contains symbolic operations that can be used to convert expressions into other expressions.

A number of symbolic processing systems are available at present, varying widely based on the representations they support as well as in the operations they allow. While some systems such as Mathematica [61] and Maple[11] are proprietary with proprietary libraries, others such as SymPy [42] and Sage [21] are freely available.

3.2.1 SymPy

SymPy [42] is a Python library for symbolic processing. It provides symbolic capabilities and can be used either in a standalone fashion, or in other applications as a library.

SymPy supports numerous symbolic capabilities including the following:

- Basic arithmetic: *, /, +, -, ** (exponentation)

- Simplification
- Expansion
- Matrix arithmetic
- Pretty printing
- Discrete summations and products

3.2.2 Sage

Sage (System for Algebra and Geometry Experimentation) is a system developed as an open source alternative to Magma, Maple, Mathematica, and MATLAB [51].

Sage supports the following operations (among others):

- Support for parallel processing using multi-core processors, multiple processors, or distributed computing.
- 2D and 3D graphs of symbolic functions and numerical data.
- Matrix manipulation

Both SymPy and Sage were considered for providing symbolic support for the system, SymPy was picked ahead of Sage due to having pattern matching operations which were a better match for the symbolic representations used in our system. The pattern matching systems in these was not found to be sufficient for the capabilities required by the system being developed. For example, support for symbolic pattern matching with commutativity, associativity and matching with multiple multiplicative terms was quite limited.

3.2.3 Pattern Matching

Pattern Matching is a powerful tool in symbolic systems as it provides a means for term rewriting and simplification. Pattern matching attempts to find matches within a given symbolic representation for a formula for "wildcards" or placeholder terms to be matched [2]. The generated substitution or the matches can then be used to replace the wildcard terms in the original pattern. For example, the substitution z=a is a valid match to the pattern f(z) when run on the symbolic representation for "f(a)".

An issue present with pattern matching is that pattern matching with either associativity or commutativity of terms has been shown to be NP-complete [3]. However, for a

likelihood which is the product of many different terms, it is not viable to exhaustively search through different combinations in order to find a match to a complex pattern.

3.3 Summary and Discussion

This chapter presents the literature review conducted in relation to the design of this work. A brief summary covering the key points is presented below.

Probabilistic programming is a paradigm related to using simple syntax to define model specifications followed by automated complex operations carried out to enable processing based on model properties. The general benefit of this is that it simplifies things for the user and enables users with less expertise access to a wider variety of operations to conduct experiments with. This was apparent from the onset of BUGS which enabled scientists from various disciplines to experiment with BUGS in order to generate meaningful results. This is confirmed by the publications made using BUGS in fields such as pharmacometrics, ecology, health-economics, genetics, archaeology, psychometrics, coastal engineering, educational performance, behavioural studies, econometrics, automated music transcription, sports modelling, fisheries stock assessment, and actuarial science [40].

Symbolic processing is the process of enabling the manipulation of mathematical expressions. Systems providing symbolic capabilities are mainly limited by their representations and simplification of the expressions they support. Systems chosen for evaluation were based on their support for a variety of simplification operations primarily matrix arithmetic and pattern matching capabilities. SymPy was picked for integration into the system due to the innate synergies present between its internal representations and the representations used in our system.

Chapter 4

System Design

This chapter presents an overview of the overall system design. Particular attention is paid to the design decisions undertaken while building the system, and how these decisions deliver in terms of robustness, efficiency and other preferable attributes of the system. An introduction to the schema based system is also provided in this chapter, along with the overall philosophy behind it. A brief discussion of the system workflow is also provided in order to link the design of the system with the desired outcomes of the system. Finally, the chapter concludes with a worked example, presented to illustrate the key components and behaviours of the system.

4.1 System Design

4.1.1 Design Factors

As the system was built with Gibbs sampling in mind, BUGS and JAGS provided an initial baseline in terms of language development, due to the fact that they both rely heavily on Gibbs sampling. However, some inspiration was also drawn from Stan in this regard. Based on the literature review, several key desirable aspects of probabilistic languages were discovered:

- **Language Simplicity:** is an important feature as it determines the ease with which a user will be able to use the system. As such, the syntax for model specification was derived to closely follow existing languages and design concepts. This is discussed in more detail in sections 3.3 and 5.2.1.
- **Robustness:** is an important quality as being able to debug the probabilistic language is of the utmost importance. This is discussed in more detail in section 3.1.1.

- **Efficiency:** of the final generated code is crucial because it determines how effective the overall system is.
- **Ease of debugging:** This is an important quality for development, as users will rely on the feedback from the system to correct any mistakes. Providing details of the generated variants, and performing simulation before conducting code generation are useful in this regard.

4.1.2 Further Design Factors

In addition to the factors in Section 4.1.1, due to the innate nature of this work, several other design aspects were considered:

- **Variant coverage:** As one of the novelties of this system arises from the support provided for statistical operations, it is imperative that a variety of different variants be generated for testing.
- **Component Generality:** As the modules developed need to perform similar operations on different models, it is important that the components implemented should be as generalized as possible. In order to achieve this, a schema-based system inspired by AutoBayes [22] was implemented. Schema are discussed in more detail in section 3.1.2.
- **Flexibility:** The code generation process was designed so that it would be flexible enough to export code in any major programming language.
- **Value Addition:** Due to the system holding considerable information about the code being generated, it is possible to create additional value by enabling tasks like generating LaTeX for equations, and parallelizing the final code by reasoning about the dependencies in the abstract code structure.
- **Code Simplicity:** The major operations used in the generated languages were constricted to simple operations such as - for loops, arrays, vectors, simple arithmetic and simple function calls. This has the added benefit of ensuring the final code can be understood by most users, regardless of coding proficiency.

4.1.3 Schema-based Design

Candidate Gibbs samplers that the system works with are represented using a pair consisting of a graph (the model structure) and a likelihood (the component model distributions). A graph and likelihood pair are referred to as a **variant**. Also introduced into the system are **schema** which generalize symbolic operations like collapsing and

augmenting into a common flexible form. Variants then are a way of maintaining alternative intermediate and final candidates for Gibbs sampling. It was found schema could be extended to a general framework for supporting several other operations that can be applied to a given intermediate variant. Candidates are called variants because the system might generate several functioning variants from a single initial model. Unlike BUGS and Stan, where a single global sampler and model is used, once collapsing and auugmenting are introduced into probabilistic programming environment, the generation of variants is unavoidable.

Thus schemas are used as a means of providing a blueprint for operations in the variant generation process. The four types of schemas used are:

Collapsing schema: corresponding to collapsing operations and containing applicability constraints and information regarding nodes to remove.

Augmenting schema: corresponding to augmentation operations and containing applicability constraints and information regarding nodes to introduce into the graph.

Simplification schema: corresponding to symbolic or graphical simplifications.

Custom schema: for providing users with flexibility in making changes to the model generation process.

4.2 System Overview

Figure 4.1 shows the overall workflow of the system with an emphasis on variant generation. The numbers in the figure correspond to the 8 steps below.

1. Initially, the parser reads in a model specification and generates a graph based on the dependencies created by the sampling statements.
2. The graph generated in step 1 is annotated with the relevant distributional information, such as probability density functions, based on the distributions supported by the statistical system.
3. Symbolic processing is carried out on the graph to generate the symbolic likelihood of each individual node, as well as other symbolic quantities such as the indicator functions relevant to the statistics of each node.
4. The graph is passed onto the variant generator, which generates possible statistical operations which may be carried out on the graph in the future.

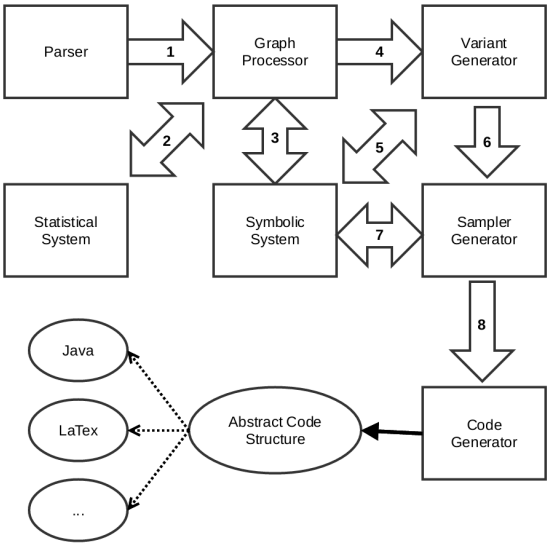


FIGURE 4.1: Simplified workflow diagram

5. The applicability criteria of statistical/graphical operations is computed by the symbolic processor based on symbolic pattern matching and simplification. These execute collapsing or augmentation steps. The symbolic likelihood, distributional information regarding the nodes and connectivity information is processed in order to extract applicable statistical operations. Identified operations are applied in a depth-first manner on the generated graph. This step is repeated until all possible graph variants have been explored (recursive depth first search with backtracking).

6. For each explored model variant, a sampler is generated for each parameter that is still present in the graph. The sampler generation process is not directly bound to the variant generation process and therefore can be adapted to accommodate many sampling strategies. Currently, Gibbs sampling is used.

7. In order to generate a sampler for the current variant, during this generation process, symbolic processing is used to extract only terms relevant to the parameter under consideration and to convert the likelihood terms into formats conducive to sampling. Additionally, analysis is conducted to identify statistics or terms that should be computed and maintained or recomputed at each iteration of the Markov chain. This corresponds to the use of caching for statistics and their relevant marginals, and is necessary for efficient code.

8. The code generation process is carried out for each variant and for each parameter. The code is initially generated as an abstract code structure, categorized as initiation or Markov chain code. Once abstract code generation is complete, the code is specialized into the target programming language. It can also be converted into other formats, such as an algorithm in LaTex suitable for embedding into a document.

4.3 Extended Example

This section provides a detailed explanation of the behaviour of the system when run on the MetaLDA model [63], which provides opportunities to demonstrate many of the different aspects of the system (collapsing, augmentation and pattern matching) in a single example. The likelihoods were generated using the LaTeX syntax generated by the system (after slight modifications to allow multiline printing). Details of the user input specification can be found in section 5.2.1. The model specification provided by the user as input into the system is as follows:

```
parameters {lambda[LxK],a[DxK],b[KxV],phi[KxV],theta[DxK],delta[TxK],z[DxI]}

data {
    b[k,v] = "Product(delta[t,k]**g[v,t],(t,1,T))";
    a[d,k] = "Product(lambda[l,k]**f[d,l],(l,1,L))";
    w[Dx?] = file(W.txt,int);
    g[VxT] = file(g.txt,boolean,sparse);
    f[DxL] = file(f.txt,boolean,sparse);
}

model {
    for (l in 1:L){
      for (k in 1:K) {
        lambda[l,k] ~ dgamma(mu,mu);
      }
    }

    for (k in 1:K) {
      for (t in 1:T){
        delta[t,k] ~ dgamma(nu,nu);
      }
      phi[k,1:V] ~ ddirich(b[k]);
    }

    for (d in 1:D){
        theta[d,1:K] ~ ddirich(a[d]);
        for (i in 1:len(w[d,:])) {
          z[d,i] ~ dcat(theta[d]);
          w[d,i] ~ dcat(phi[z[d,i]]);
        }
```

```
    }
}
```

Note the undefined hyperparameters mu and nu in the specification. These are assigned default values (usually 1.0) during code generation and the user is free to change them as they wish. The diagram corresponding to this model specification is given in 4.2.

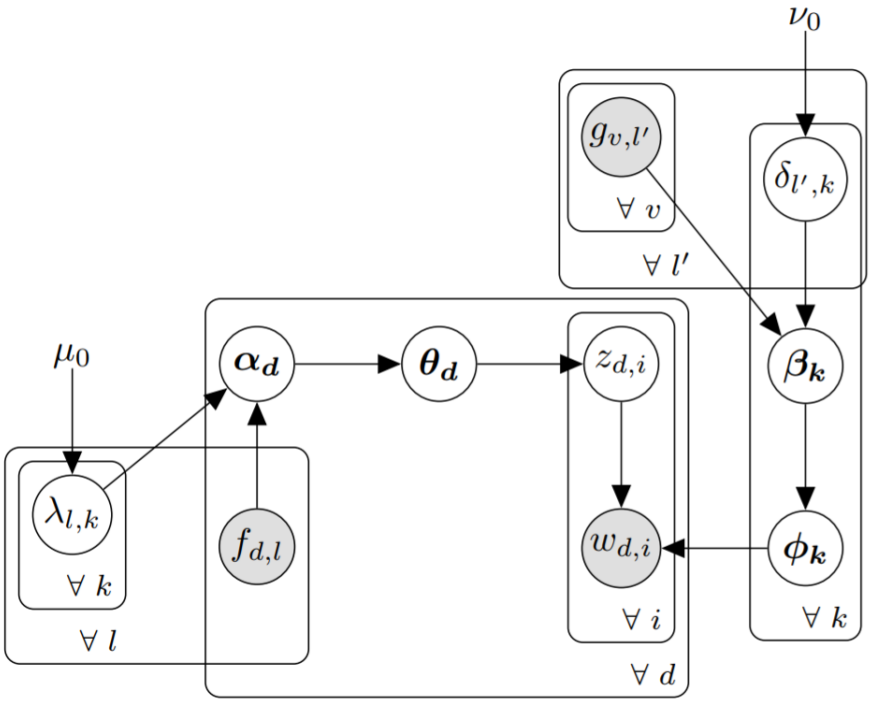


FIGURE 4.2: MetaLDA diagram

After parsing the specification and performing the necessary symbolic processing, the intermediate graphical structure denoted by figure 4.3 is generated. Note that there are some slight differences between the diagram and the formulae (This is because the formulae in this section have been automatically generated by printing the latex generated by the system, and thus corresponds to the initial model specification at the start of section 4.3).

The corresponding likelihood for this graph is given by equation 4.1. In this state, the graph processor identifies that nodes θ is conjugate to node α as indicated in figure 4.4. This conjugacy is confirmed by checking the corresponding terms in the likelihood.

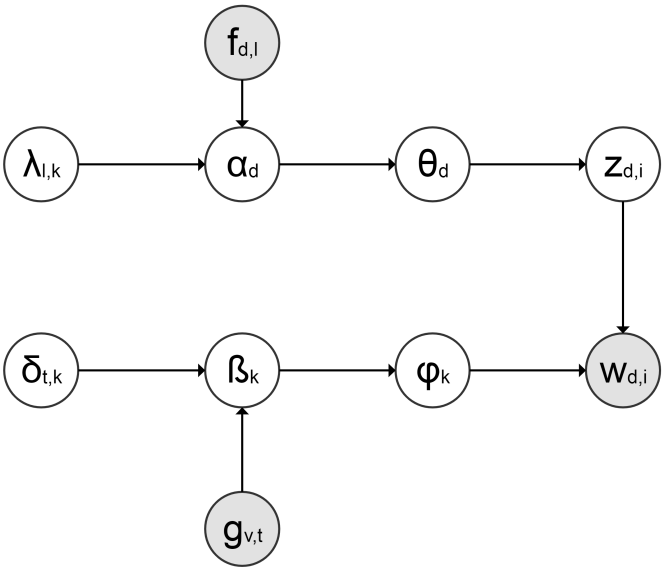


FIGURE 4.3: MetaLDA generated graph

$$\frac{\left(\prod_{d=1}^{D}\Gamma\left(\sum_{k=1}^{K}a(d,k)\right)\right)\left(\prod_{k=1}^{K}\Gamma\left(\sum_{v=1}^{V}b(k,v)\right)\right)\left(\prod_{\substack{1\le v\le V\\1\le k\le K}}\phi^{c_1}(k,v)\right)\left(\prod_{\substack{1\le v\le V\\1\le k\le K}}\phi^{b(k,v)-1}(k,v)\right)\left(\prod_{\substack{1\le k\le K\\1\le d\le D}}\theta^{c_0}(d,k)\right)\prod_{\substack{1\le k\le K\\1\le d\le D}}\theta^{a(d,k)-1}(d,k)}{\left(\prod_{\substack{1\le k\le K\\1\le d\le D}}\Gamma(a(d,k))\right)\prod_{\substack{1\le v\le V\\1\le k\le K}}\Gamma(b(k,v))} \tag{4.1}$$

Note that z and w do not appear in equation 4.1 as they are accounted for in the counts c_0, c_1 which correspond to topic and term counts respectively. Further explanation of this step is present in section 1.2.2. For example, c_0 is a count of the form $c(d,k)$ which tracks how many topics(k) coincide with term v. It is denoted simply by c_0 as the system represents internal counts specifically in this form and maintains additional information about them separately. Accordingly, node θ is collapsed out by performing a collapse operation. The node theta is deactivated, and a new conjugate node is introduced to provide the likelihood introduced by the collapsing operation. The resultant graph is given in figure 4.5.

The new likelihood is given by equation 4.2. Note that the deactivated node θ no longer contributes to the likelihood. However, the presence of this node in the graph is significant, as it provides a simple way of undoing the collapse operation (important for maintaining state during the depth-first-search). This is indicated in figure 4.5 by the dotted lines.

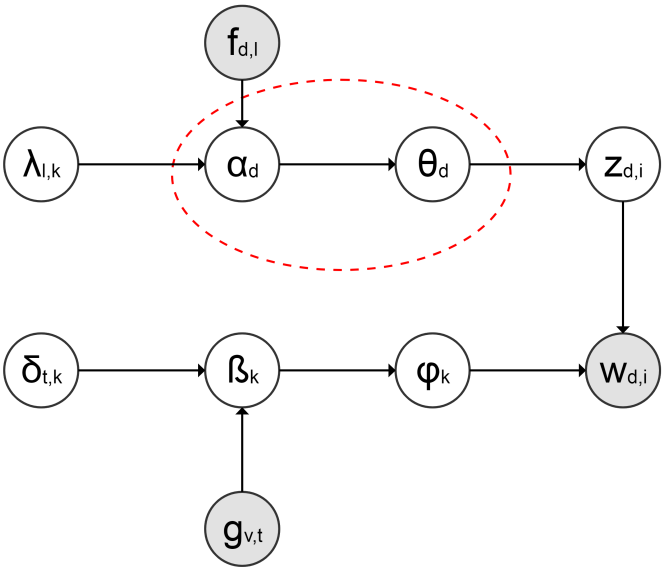


FIGURE 4.4: MetaLDA generated graph 2

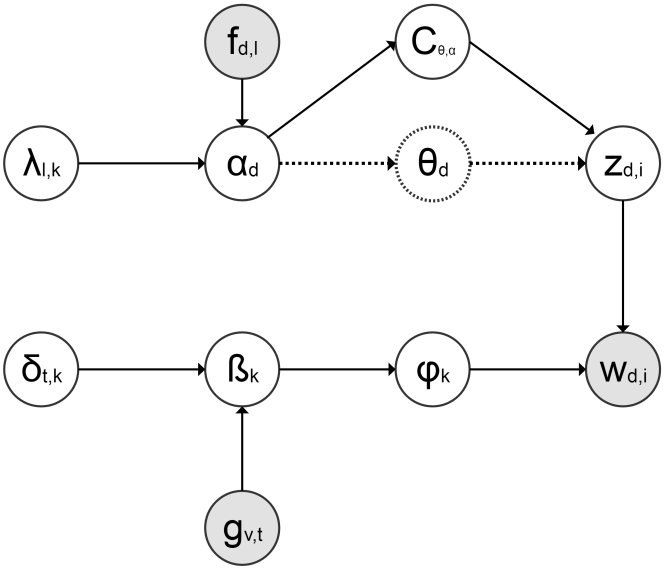


FIGURE 4.5: MetaLDA generated graph 3

$$\frac{(\prod_{d=1}^{D}\Gamma(\sum_{k=1}^{K}a(d,k)))(\prod_{k=1}^{K}\Gamma(\sum_{v=1}^{V}b(k,v)))(\prod_{\substack{1\le k\le K\\1\le d\le D}}\Gamma(c_0+a(d,k)))(\prod_{\substack{1\le v\le V\\1\le k\le K}}\phi^{c_1}(k,v))\prod_{\substack{1\le v\le V\\1\le k\le K}}\phi^{b(k,v)-1}(k,v)}{(\prod_{d=1}^{D}\Gamma(\sum_{k=1}^{K}(c_0+a(d,k))))(\prod_{\substack{1\le k\le K\\1\le d\le D}}\Gamma(a(d,k)))\prod_{\substack{1\le v\le V\\1\le k\le K}}\Gamma(b(k,v))} \tag{4.2}$$

At this stage, the symbolic system identifies a match for the augmenting pattern of the form $\Gamma(W1+W2)/\Gamma(W1)$, where $W1$ and $W2$ indicate wildcards for pattern matching. The matching terms are given in equation 4.3. Here c_1 is a count of the form $c(k,v)$ which tracks how many topics(k) coincide with term v.

$$\frac{(\prod_{\substack{1\leq k\leq K\\1\leq d\leq D}}\Gamma(c_0+a(d,k)))}{(\prod_{\substack{1\leq k\leq K\\1\leq d\leq D}}\Gamma(a(d,k)))} \tag{4.3}$$

The terms in equation 4.3 form a rising factorial, which can be augmented with an auxiliary variable $t_{d,k}$ as shown in equation 4.4, where S indicates an unsigned Sterling number of the first kind.

$$\frac{(\Gamma(c_0+a(d,k)))}{(\Gamma(a(d,k)))} \propto \sum_{t_{d,k}=0}^{c0} S_{t_{d,k}}^{c0} a(d,k)^{t_{d,k}} \tag{4.4}$$

Here $t_{d,k}$ can be sampled as shown in section 2.3.3 for the CRT sampler. Performing the augmentation operation leads to the graph shown in figure 4.6.

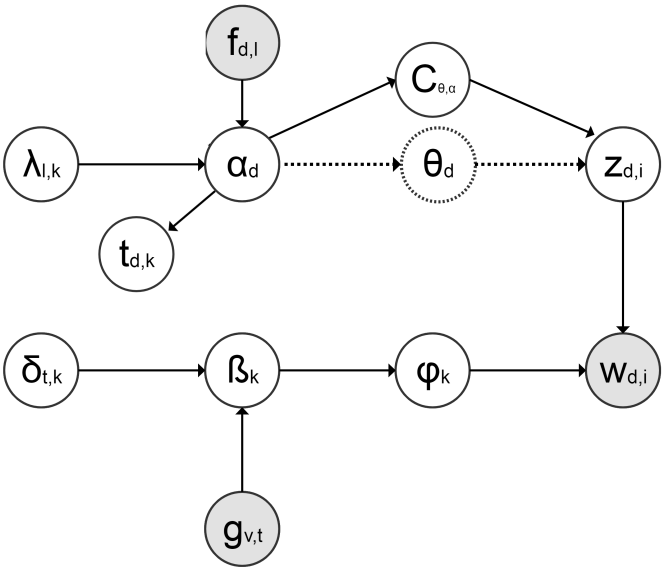


FIGURE 4.6: MetaLDA generated graph 4

This graph has the likelihood shown in equation 4.5.

$$\frac{\begin{array}{r}(\prod_{d=1}^{D}\Gamma(\sum_{k=1}^{K}a(d,k)))(\prod_{k=1}^{K}\Gamma(\sum_{v=1}^{V}b(k,v)))(\prod_{\substack{1\leq k\leq K\\1\leq d\leq D}}a^{\mathrm{t}\,(d,k)}(d,k))\\(\prod_{\substack{1\leq v\leq V\\1\leq k\leq K}}\phi^{c_1}(k,v))\prod_{\substack{1\leq v\leq V\\1\leq k\leq K}}\phi^{b(k,v)-1}(k,v)\end{array}}{(\prod_{d=1}^{D}\Gamma(\sum_{k=1}^{K}(c_0+a(d,k))))\prod_{\substack{1\leq v\leq V\\1\leq k\leq K}}\Gamma(b(k,v))} \tag{4.5}$$

At this stage, the symbolic system identifies a match for the augmenting pattern of the form $\Gamma(W1)/\Gamma(W1+W2)$. The matching terms are given in equation 4.6.

$$\frac{(\prod_{d=1}^{D}\Gamma(\sum_{k=1}^{K}a(d,k)))}{(\prod_{d=1}^{D}\Gamma(\sum_{k=1}^{K}(c_0+a(d,k))))} \tag{4.6}$$

The terms in equation 4.6 can be augmented by a set of Beta random variables $q_{1:D}$ as shown in equation 4.7. Here $q_d \sim Beta\left(\sum_{k=1}^{K}a(d,k), \sum_{k=1}^{K}c_0\right)$

$$\frac{(\Gamma(\sum_{k=1}^{K}a(d,k)))}{(\Gamma(\sum_{k=1}^{K}(c_0+a(d,k))))} \propto \int_{q_d} q_d{}^{(\sum_{k=1}^{K}a(d,k))-1}(1-q_d)^{(\sum_{k=1}^{K}(c_0))-1} \tag{4.7}$$

Performing the augmentation operation leads to the graph shown in figure 4.7. The corresponding likelihood is given in equation 4.8. Simplification details can be found in section 2.3.2.

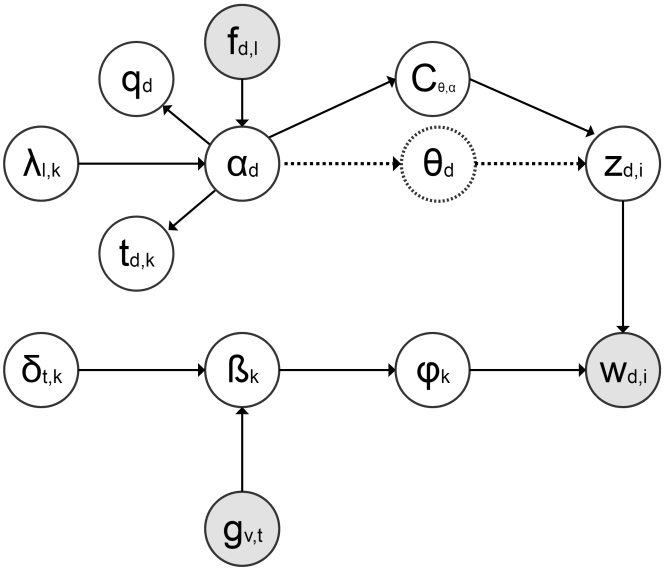


FIGURE 4.7: MetaLDA generated graph 5

$$\frac{\begin{aligned}(\prod_{k=1}^{K}\Gamma(\textstyle\sum_{v=1}^{V}b(k,v)))(\prod_{\substack{1\le k\le K\\1\le d\le D}}a^{\mathrm{t}\,(d,k)}(d,k))(\prod_{\substack{1\le v\le V\\1\le k\le K}}\phi^{c_1}(k,v))\\(\prod_{\substack{1\le v\le V\\1\le k\le K}}\phi^{b(k,v)-1}(k,v))\prod_{\substack{1\le k\le K\\1\le d\le D}}\mathrm{q}^{a(d,k)}\,(d)\end{aligned}}{\prod_{\substack{1\le v\le V\\1\le k\le K}}\Gamma(b(k,v))} \tag{4.8}$$

Isolating the terms relevant to $a(d,k)$ gives equation 4.9.

$$\prod_{\substack{1\leq k\leq K\\ 1\leq d\leq D}} a^{\mathrm{t}\,(d,k)}(d,k) \prod_{\substack{1\leq k\leq K\\ 1\leq d\leq D}} \mathrm{q}^{a(d,k)}\,(d) \tag{4.9}$$

Substituting for $a(d,k)$ with $\lambda(l,k)^{a(d,k)}$ and simplifying to isolate only relevant terms gives the contributions to the marginal posterior for $\lambda(l,k)$ from equation 4.9 as shown in equation 4.10.

$$e^{-\lambda(l,k)\sum_{d=1;f(d,l)=1}^{D} a(d,k)log(1/q(d))/\lambda(l,k)} \lambda(l,k)^{\sum_{d=1;f(d,l)=1}^{D} t(d,k)} \tag{4.10}$$

Which allows $\lambda(l,k)$ to be sampled according to equation 4.11.

$$\lambda(l,k) \sim Ga\left(\mu + \sum_{d=1;f(d,l)=1}^{D} t(d,k),\ 1/\mu - \sum_{d=1;f(d,l)=1}^{D} a(d,k)log(q(d))/\lambda(l,k)\right) \tag{4.11}$$

Since the model is symmetric, a similar strategy can be adopted for sampling $\delta(t,k)$ and a similar set of states will occur involving only the nodes in the bottom of the figure (β_k , ϕ_k, $g_{v,t}$). With this, all the nodes can be sampled/computed and code can be generated for the model. The generated code has been added as Appendix A.1.

Chapter 5

System Architecture

While Chapter 4 gives a high-level overview of the system, a more detailed presentation is required to fully understand the workings of the final system. This chapter presents the detailed design and covers all of the components at a more granular level.

5.1 System Architecture

A more detailed version of the system architecture is presented in Figure 5.1 with some modules of lower priority omitted for the sake of brevity. Again, numbers in the figure correspond to numbers listed below.

1. The parser relays the processed model specification as an adjacency list to generate an annotated graph.
2. The graphical processor updates the annotated graph and processes initial information such as potential candidates for collapsing.

Symbolic processing for variant generation

3. The symbolic processor processes the annotations of the graph to derive density functions for parameter nodes.
4. The derived functions are generated as symbolic representations (indicator functions as well as standard symbolic representations).
5. Symbolic reasoning is applied on these functions to identify which may lead to potentially complex updates and the results are encoded in the graph.

Variant generation operation identification

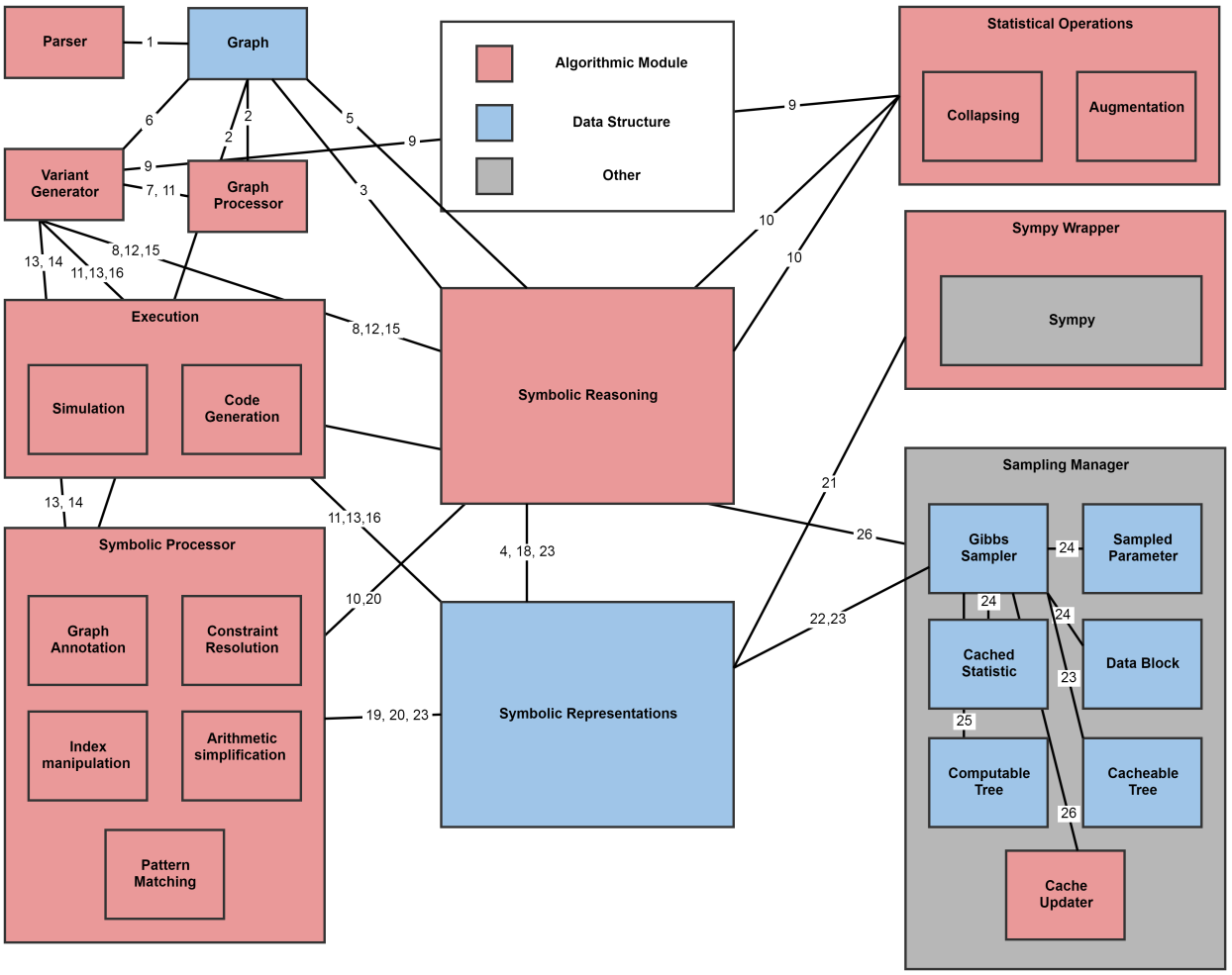


FIGURE 5.1: Architectural diagram

6. The initial model is passed to the variant generator to derive variants.
7. The variant generator generates a list of operations for the current state of the graph based on graphical patterns.
8. The variant generator generates possible symbolic operations based on the current state of the likelihood.
9. Any collapsing/augmentation operations are applied.
10. Depending on the operation the required change is generated processed using Symbolic reasoning and the Symbolic processor to generate the required update.
11. The representation is updated along with the graph based on the operation conducted.
12. Possible Symbolic operations are identified.
13. Symbolic operations are carried out causing changes to the representation.

14. Steps 7 to 13 are repeated until all model variants have been explored. For each model variant the following steps are carried out.

Symbolic processing for Sampler Generation

15. Symbolic reasoning is carried out to identify the structure of the likelihood and this is processed symbolically to generate the overall model likelihood.
16. The corresponding symbolic representation is generated.
17. For each parameter remaining in the model, the steps 18-24 are carried out.
18. Symbolic reasoning is carried out to identify the terms in the likelihood relevant to the parameter.
19. The relevant terms are extracted from the overall likelihood. The resultant expression is represented as an Expression Tree.
20. Symbolic reasoning is carried out to identify the form of the full conditional probability for the parameter. This is then generated in standard form.
21. The result is logged/visualized using SymPy.

Sampler Generation

22. Based on the full conditional, the corresponding Gibbs Sampler is generated.
23. The full conditional is converted to a cacheable tree format. Using this with pattern matching and symbolic reasoning, relevant sufficient statistics are identified.
24. Data Structures relevant to the parameter are generated, if already available symbolic links are created to the existing structure instead.

Code Generation/Execution

25. The Gibbs sampler full conditional is updated to a computable tree representation to prepare it for the sampling process.
26. Execution is initiated. Initialization is carried out with the Cache updater performing message passing duties to ensure all parameters and statistics remain consistent. Code Generation is carried out in a similar manner.

A rough estimate of the lines of code of each of the major modules is given below to provide an overview of the scope of each system. As the representation of distributions is decoupled from the other systems, the lines of code is an accurate representation of that individual system. The size of the systems corresponds primarily to functionality focused around that system and modularity is maintained in an object oriented fashion.

Lines of code estimate	
Module	Lines of Code
Symbolic Reasoning	2000
Symbolic Representation	1500
Sampling Manager	2500
Statistical Operations	2000
Execution	1500
SymPy Wrapper	1000
Parser	1000
Graph	500
Variant Generator	1000
Graph Processor	500
Symbolic Processor	3000

5.2 Graph Generation

The model analysis process starts off with the generation of an initial annotated graph. This section details the modules involved in this process.

5.2.1 Parser

The language used for defining models was inspired by several existing systems. The sampling statements were inspired by BUGS. The presence of iteration or tiling was inspired by JAGS. Explicitly denoting useful properties such as parameter nodes was inspired by Stan. The language currently consists of the following components:

- The **model** keyword specifies the model block, which is used to define the model. The model can contain tiling and sampling statements.
- The **data** keyword is used for declaring the data for fitting the model. Data can be in a sparse format, and could be located in a separate file. The data keyword can also be used to define constants.
- The **parameter** keyword similarly defines parameters and types used in the model and also specifies the dimensions of parameters used in the model.

- **Distributions** are prefixed by the letter 'd' and must only appear in a sampling statement. For example, (phi $\sim$ ddirich(a)) denotes that ϕ is distributed according to the Dirichlet distribution.
- **Formulae** are supported using the assignment operator. Additionally, operators such as Sum and Prod (short for product) are supported, allowing for further flexibility.
- The **for** keyword is used to indicate tiling, and additionally specifies the dimensions of the tiling as well. For example "for (k in 1:K)" denotes tiling for all variables containing k, with a dimension of 1 to K.
- **Annotations** are defined using square brackets, and can be used to provide additional information to the compiler.
- **Line comments** are enabled using the # symbol #.

A sample model specifications for LDA is provided below.

```
#LDA
parameters {z[MxN],phi[KxV],theta[MxK]}

data{
    w[MxN] = file(W.txt);
    a[K] = file(a.txt);
    b[V] = file(b.txt);
}

model {
  for (k in 1:K) {
    phi[k,1:V] ~ ddirich(b);
  }
  for (m in 1:M) {
    theta[m,1:K] ~ ddirich(a);
    for (n in 1:N) {
      z[m,n] ~ dcat(theta[m]);
      w[m,n] ~ dcat(phi[z[m,n]]);
    }
  }
}
```

The parser processes the provided model and creates the corresponding graph for further processing.

5.2.2 Graph

All information regarding the model is encapsulated into a directed acyclic graph (DAG), with the following structure.

- The **nodes** correspond to parameters or data variables within the model.
- The **edges** indicate direct dependence between nodes.

Nodes contain information regarding the sufficient statistics for a particular parameter. These are generated by the Symbolic Processor (discussed in detail under 5.3.3).

5.3 Variant Generation

Once the graph has been generated by the parser, and the necessary annotations have been completed by the symbolic processor, the variant generation process can take place. This section describes this process in detail.

5.3.1 Variant generation workflow

A variant can be characterised by a graph G and a likelihood L. Each variant of the model $V(G, L)$ is derived in a reversible manner from the original model $V_0(G_0, L_0)$. In general, given a sequence of statistical operations $(S_1, S_2...S_n)$. $V(G, L) = S_n S_{n-1}...S_1 V_0(G_0, L_0)$ can be reversed by the sequence of inverse statistical operations $(S_n^{-1}, S_{n-1}^{-1}...S_1^{-1})$. A more thorough definition, along with relevant proofs can be found in Chapter 6.

Variants are generated using a depth-first search algorithm, with state being maintained by an intermediate graphical structure which encapsulates information about the model necessary for the sampling process.

5.3.2 Statistical Operations

The main technique used in the system for changing the underlying algorithm is the operations of collapsing and augmentation, covered in Section 2.4.2. As will be made clear by example in sections 5.3.5.1 and 4.3, pattern matching is required in order to identify the applicability of augmentation and collapsing to an intermediate graphical state. In collapsing, this is in the form of an initial graphical pattern matching operation followed by an arithmetic simplification operation on the likelihood. In augmentation, an initial pattern matching operation on the likelihood followed by an update of the

intermediate graphical structure is carried out. This is then followed by an update of the likelihood of the model.

5.3.2.1 Statistical operation generation

The statistical operations of collapsing and augmentation are the primary means by which the variant generation process introduces changes into the graph. Statistical operations are embedded within schemas and their applicability is queried using symbolic processing. The statistical operations encoded in our system are the augmentations given in Table 2.2 and the collapsing is enabled using the conjugate forms of the exponential family distributions we allow. A simple overview of these is provided in Algorithms 4 and 5. As the collapsing depends on conjugacy of exponential family distributions, the schema provides the symbolic format of the normalizer for the conjugate distribution as a baseline for the likelihood to be matched against. conjugate(N, N.prior, prior, node) simply checks that the distributions of N and N.prior match those of prior and node.

```
CollapseIdentify(Graph G, CollapseSchema S):
prior, node = S.getNodeTypes()
for all nodes N ∈ G do
    if conjugate(N, N.prior, prior, node) then
        jointLH = N.likelihood*N.prior.likelihood
        G.addOperation(Collapser(N,N.prior))
    end
end
```

Algorithm 4: Collapse operation generation

```
AugmentIdentify(Graph G, AugmentSchema S):
pattern = S.getPattern()
likelihood = G.GenerateLikelihood()
if likelihood.matches(pattern) then
    matches = MatchPattern(likelihood, pattern)
    for MatchedPattern M ∈ matches do
        aug = pattern
        for Wildcard W ∈ pattern do
            aug.substitute(W , M.getMatch(W))
        end
        N = S.generateNode(aug)
        N.updateConnectivity(G)
        G.addOperation(Augmentor(N , aug))
    end
end
```

Algorithm 5: Augment operation generation

In the case of augmentation, because the operation arises as a result of the state of the likelihood, it is possible to have the same augmentation triggered multiple times.

In order to avoid duplicating the same augmented variable under a different name, a signature (hash) is used: the pattern used in the augmentation and concatenated with the substitutions from the pattern matching operation.

5.3.3 Symbolic Processor

This section provides details on the symbolic reasoning functionality of the system. An overarching example is provided at the end of the section.

5.3.3.1 Graph Annotation

While the parser provides context information regarding the connectivity of the graph, in order to proceed with the analysis of the model, it is necessary to annotate each node with formulae corresponding to its sufficient statistics. This involves symbolic reasoning over indicator variables, which are derived based on the node and its distribution type as well as the dimensions of the parameter represented by the node.

5.3.4 Constraint Resolution

Some models have inherent constraints applicable to them, such as the identifiability constraint in GaP [9] which is denoted by equation 5.1 and explained in more detail in section 5.3.5.1. Generally, such constraints lead to considerable simplification of the model. The symbolic processor manages the symbolic representation of the constraint as well as decides on the context and result of its application in the simplification process.

5.3.4.1 Index manipulation

Due to the system providing support for models with multiple dimensions, the computation of the model likelihood requires index manipulation in order to identify potential opportunities for further simplification. This is especially relevant to constraint resolution as in general, the context in question will only apply to a single index (or single dimension) of a particular parameter. This is explained with examples in section 5.3.5.1.

5.3.4.2 Algebraic simplification

As the graph undergoes graphical operations such as collapsing and augmentation, the format of the likelihood changes depending on the operation performed. However, in order to apply these operations, the likelihood must match a particular format. Hence,

arithmetic simplification is required to bridge the gap between a single graph operation and the graph operation immediately following it, on the computation path to generating a likelihood which is conducive to having a full conditional likelihood extracted for the sampling process. An example is found in section 5.3.5.1.

5.3.4.3 Equivalent Symbolic Likelihoods

In some situations, the likelihood generation process can create mathematically equivalent, yet symbolically different likelihoods that have implications on how the sampling and model analysis process is carried out. Consider LDA as an example. The fully collapsed variant of LDA relies on the symbolic form of $\theta(d,k)^{c1(d,k)}$ where $c1(d,k)$ indicates the number of times topic k appears in document d. However, the base version (no collapsing) of LDA relies on the symbolic form $\prod_{d,i}\theta(d,k(i))$ (where i corresponds to the word index) to represent the probability of θ and relies on isolating the probability for a particular $\theta(d,k)$. Support for this form of switching can be supported in 3 different ways:

1. Heuristic based switching - Decide between any such forms based on a heuristic, given the current state of the graph and/or likelihood.
2. Evaluation based switching - Decide between any such forms based on introducing each term into the likelihood followed by performing a sampler generation step to verify which of the forms fits the likelihood currently.
3. Explore all possibilities in the variant generation process (introduce as a level into the depth first search)

Of these methods the first method was used during initial stages of the system where collapsing was the only operation supported, however evaluation based switching provides a more stable process for deciding which form the likelihood should take and is currently used. Introducing this issue into the variant generation process is the most effective in terms of overall model coverage as it guarantees that all possibilities are explored, however the issue arises in the fact that introduces another level to the search, causing a lengthier search process. Incorporating into the variant search process would also involve introducing a new type of schema corresponding to this issue.

5.3.4.4 Pattern Matching

Pattern matching is a key operation required throughout the work-flow of the system and is used in multiple stages of the model evaluation process. The following operations rely heavily on pattern matching:

- Constraint resolution: checking for the applicability of a constraint
- Cacheability analysis of full conditional formulae
- Index manipulation: identifying the form of an index and relation to parameters being sampled
- Arithmetic simplification: checking for applicability of simplification rules

Pattern matching is primarily carried out in 2 main forms. Regular expressions are used to resolve simple pattern matching. While complex pattern matching uses a custom implementation of a tree based representation of the likelihood function. This is discussed in further detail under the section on symbolic representations 5.4.1. Simple tree based matching is carried out on such representations by performing a simple modified depth first search over the labels on such a tree, by first converting both query and string into tree representation.

Pattern matching is also used extensively in the statistical operations of collapsing and augmenting nodes. This is discussed with examples under section 5.3.2.

Complex pattern matching

Complex pattern matching builds on the simple tree based matching explained previously.

The overall steps are as follows:

- First a transform is applied on both likelihood and pattern to split it up into terms that are more conducive to matching.
- A round of matching is conducted between the terms in the pattern and the terms in the likelihood
- All matches are aggregated according to the terms in the pattern
- The matches are filtered for coherence and the resultant set is returned in the form of substitutions (eg:- $W_1 = \alpha(k, v)$)

While simple pattern matching only used information in the form of strings and string matching, complex pattern matching uses additional information available in the standard tree representation used in the system. This involves adding auxiliary information to nodes in a strategic manner to obtain more information about likelihood components. The following additional information is added to nodes:

- parameter information - indicates if a node is a parameter

- statistic information - indicates if a node is a statistic
- dimensionality information - where a node corresponds to an entity that has multiple dimensions (arrays etc.), the corresponding dimensions are recorded as well.
- commutativity and associativity information - for operations such as + and -, where these properties need to be handled explicitly.

Schema (section 6.1) allow users to exploit many properties in the likelihood and graphical elements in the variant generation process. The symbolic support for schema is provided primarily through complex pattern matching and this is where parameter and statistic information are especially useful.

For the purpose of clarity we will discuss the pattern matching algorithm as applying on strings, which is a valid form of the standard representation. However, it should be noted that internally the algorithm operates on tree based representations as mentioned in 5.4.1.1.

A high level view of the complex pattern matching algorithm is given in Algorithm 6. Details of subroutines are found in Algorithms 7 and 8.

```
MatchPattern(Likelihood L, Pattern P):
L = Transform(L)
P = Transform(P)
for all termTypes t ∈ P do
    for all components C ∈ P[t] do
        for all terms T ∈ L[t] do
            if T.matches(C) then
                componentMatches[C].add(T)
            end
        end
    end
end
return coherentMatches(componentMatches)
```

Algorithm 6: Pattern Matching

The algorithm proceeds as follows: initially, a transform is applied to split both pattern and likelihood into terms in an identical fashion.

The result of the transform has 2 different types of components, and matching is done on the different types in a mutually exclusive fashion (multiplicative pattern components are matched with multiplicative likelihood terms and divisive pattern components with divisive likelihood terms).

The applyLog function applies the logarithm function to all parts of the expression, including propagating it over multiplication, division and products using the rules $log(a *$

```
Transform(Expression E):
LogE = applyLog(E)
PositiveTerms = empty
NegativeTerms = empty
for all terms t delimited by +log or −log ∈ LogE do
    if delimiter is '+ log' then
        PositiveTerms.add(t)
    end
    if delimiter is '- log' then
        NegativeTerms.add(t)
    end
end
return PositiveTerms,NegativeTerms
```

Algorithm 7: Pattern Matching

$b) = log(a) + log(b)$ and $log(a/b) = log(a) - log(b)$. The result of this function is also an expression. Note that since this is applied on a string/tree based representation, considerable symbolic processing is required to propagate the function over the expression.

Once the transform is applied, simple pattern matching can be carried out to check which terms match a particular pattern. This is done by matching all likelihood terms of a particular type against the pattern components of the same type.

```
for all termTypes t do
    for all components C ∈ P[t] do
        for all terms T ∈ L[t] do
            //Simple pattern matching used to check match
            if T.matches(C) then
                componentMatches[C].add(T)
            end
        end
    end
end
```

Algorithm 8: Part of the Pattern Matching algorithm

The result of these lines is a complete set of matches indicating which terms are potential candidates for being part of an overall match corresponding to the overall pattern.

For example consider matching $\frac{W_1}{W_2}$ against $\frac{a*b}{c*d}$. The pattern components would correspond to W_1 and W_2 and the likelihood terms would correspond to a, b, c, d. Then for this example, the component matches would be $W_1 \in \{a, b\}$ and $W_2 \in \{c, d\}$.

Once the component matches are generated, it is necessary to consolidate them into overall pattern matches. Consider the following observations regarding component matches:

- The higher the number of wildcards a pattern component has, the more restrictive it will be in terms of being able to add other matches into the overall pattern. For

example if a component has as many wildcards as the entire pattern, that single component will decide all possible wildcard matches for the entire pattern and will play a crucial role in either validating or invalidating other matches. Consider the pattern $\frac{W_1+W_2+W_3+W_4}{W_1}$. Then, pattern component $W_1+W_2+W_3+W_4$ would have a considerably higher level of restrictivity than $\frac{1}{W_1}$.

- Every component needs to have at least one match. This is obvious because otherwise the pattern has no matches in the likelihood. Consider matching $\frac{W_1+W_2+W_3+W_4}{W1}$ against $a+b+c+d$. Clearly, the component $\frac{1}{W_1}$ does not have a match in this case, and therefore there is no valid match for the pattern as a whole.

Since every component needs to have at least a single match, and more restrictive pattern components are harder to match. By ordering all matching based on the number of wildcards, it is possible to filter out potential likelihood terms faster. This applies on two separate levels: at the component-term match level, if a single pattern component does not have any term matches, it is indicative that the entire pattern does not have any matches. In this situation, the matching can be terminated early and return with no matches. During the consolidation of the overall match for the pattern, we can treat the wildcards as variables that need solving. Thus, each unmatched wildcard would contribute to the degrees of freedom available in the pattern "equation". Thus, picking pattern components with more wild cards initially will fix those wildcards in place, and remove more degrees of freedom from the pattern, potentially causing failure due to not having a coherent match earlier, rather than later.

With this in mind, components matches are sorted by decreasing order of restrictivity of the pattern. The primary contributor in this sense is the number of wildcards in the component, however there are some additional considerations regarding the types of wildcards used as well. For example, the "any match" criteria is less restrictive than

the “any parameter” criteria.

```
coherentmatches(componentMatches,i,solutionSet):
//componentMatches is sorted by decreasing restrictivity
//components with higher restrictivity are handled first.
if i ≥ allMatches.length then
    //At least one match has been added for each pattern component
    //This is then a valid match for the overall pattern
    RegisterMatch(solutionSet)
    return;
end
for each match M ∈ componentMatches[i] do
    component c = M.component
    for each wildcard w ∈ c do
        coherent = true
        if w ∈ solutionSet.components then
            //This wildcard is already in the solution
            //See if they match
            if M[w] != SolutionSet[w] then
                //Mismatch! This match is not suitable
                coherent = false
                break
            end
        end
    end
    if coherent = true then
        //This match is coherent when combined with the solution so far
        //The two sets are combined
        solutionset = solutionset + M
        coherentMatches(componentMatches,i+1,solutionset)
        solutionset = solutionset - M
    end
end
return coherentMatches(componentMatches)
```

Algorithm 9: Pattern Matching - coherent matches

At each stage of this algorithm, the following invariant is maintained: solutionSet contains a valid match for the overall pattern using all wildcards available in components in the first $i-1$ components with the highest restrictivity. This arises from the fact that the components in componentMatches are ordered by decreasing restrictivity. Here, i is a counter indicating the number of components processed, in other words, i points to the next component to be processed. The iteration in the algorithm serves to ensure that the match M current being checked, can be incorporated into the solutionSet without causing a contradiction. Consider the following example:

$$a * b * (b + c); match(W_1 + W_2) * W_1$$

Clearly $W_1 + W_2$ is the component with higher restrictivity and matches with $b + c$ as $\{W_1 : b, W_2 : c\}$. Now this has 2 potential matches to combine with $\{W_1 : b\}$ and $\{W_1 : a\}$. Of these, the former will be coherent, as there is no contradiction, but the latter will not be coherent as $\{W_1 : b\}$ and $\{W_1 : a\}$ will contradict each other, thus rendering them incompatible. This is because $\{W_1 : b, W_2 : c\} \cap \{W_1 : b\} = \{W_1 : b, W_2 : c\}$ whereas $\{W_1 : b, W_2 : c\} \cap \{W_1 : a\}$ results in a set with no valid element for W_1. Note that where a particular element is missing (for example W_2 in $\{W_1 : b\}$, it is assumed to be equal to the universal set, as it being absent is indicative of no constraints on that wildcard.

By extending this invariant with i (parameter in Algorithm 9) set to the number of components, we see that only completely coherent matches which contain non-contradicting matches for all wildcards will actually reach the RegisterMatch function. At this stage, the match is added to a set of matches, and this stores the overall result of the algorithm.

Additionally, during matching, the original position of each term in the initial likelihood is tracked, and used to indicate potentially multiple components matching the same term. Once detected, such matches are skipped over, thus preventing any two components to match to the same term. An example would be:

$$a * b * (b + c); \quad match(W_1 + W_2) * (W_1 + W_3)$$

Clearly, we would not want both components of the pattern to match $b + c$ in such a scenario to give a result of $\{W_1 : b, W_2 : c, W_3 : c\}$.

Simple Pattern Matching

Simple pattern matching is used to provide basic pattern matching functionality, including matching with wildcards. The distinction between simple and complex arises from the fact that simple pattern matching is unable to operate on likelihoods in order to support schema applicability constraint matching in any meaningful way primarily due to the inability to handle commutativity and associativity. Simple pattern matching is provided using tree-based search on the expression tree. SymPy's pattern matching functionality provides additional support in this regard, as well.

It should be noted that simple pattern matching does not handle commutativity and associativity implicitly. Therefore, since complex pattern matching leverages simple pattern matching to an extent, it needs to account for these possibilities. This is done at 2 levels. At the likelihood level, multiplication and division are handled by the

algorithm implicitly. At the term level (or at the operation level), this is done explicitly and combined with the other type of auxiliary information to provide more powerful matching functionality.

As an example consider the simple terms $a + b + c$ with a being a statistic, b being a parameter, and c being a constant. Then let us consider an example where a user wishes to match $(W_1 + W_2)$ with the restrictions of W_1 being a statistic and W_2 being any non-zero (no restrictions). Simple pattern matching would need to consider all 6 permutations of $a + b + c$ and possible bracketing combinations (used to indicate associativity in terms of which goes first) to finally settle on the matches of $(a) + (b + c)$ and $(a) + (c + b)$. With the additional auxiliary information available in the modified standard tree representation, however, it is possible to greatly simplify this process.

5.3.5 Symbolic Reasoning

Symbolic reasoning encapsulates the processes which allow the system to reason about data structures and formulae in order to conduct analysis on a model. This is distinct from pattern matching, as pattern matching only resolves the issue of identifying the presence of a symbolic pattern. While symbolic reasoning builds on different forms of pattern matching, it also uses other available information to reason out a course of action and is crucial for the overall graphical and symbolic processing workflows.

Symbolic reasoning is mainly used in the following functions:

- Update reasoning: due to the complex interactions present between parameters and the need for automatic updates during the sampling process, it is necessary to provide symbolic reasoning for performing cache updates at run-time, particularly for cases involving cascading cache updates. This is discussed in more detail under the section on symbolic updates.
- Dependency identification: In creating the Gibbs sampler for a parameter, it is helpful to first isolate the terms in the overall model likelihood that are impacted by the sampling of the parameter. This requires symbolic reasoning over all terms in the likelihood. It is important to note that indirect dependencies also have to be considered. For example a statistic may have the form $c1[m][k]$ where $k = Z[m][n]$. So clearly, when Z is sampled, c1 needs to be updated as well. This requires an additional symbolic substitution on top of the standard check which is done for direct dependencies.
- Full conditional derivation: during the derivation of the full conditional likelihood for a parameter, symbolic reasoning is used in identifying the behaviour of functions over iterations of the Markov chain. There is also an additional requirement that

this be set up in a dynamic manner as certain quantities (for example sufficient statistics) present in the likelihood formula might differ in value from one iteration of the chain to the next.

- Likelihood format determination: the same parameter/distribution pair can have different formats for the likelihood representation. This is especially obvious in collapsed versus uncollapsed model variants, where the likelihood for the collapsed variant generally relies on the sufficient statistics whereas the likelihood for the uncollapsed version relies on the naive likelihood for the model.

5.3.5.1 Example of symbolic processing functions

The effect of these operations can be seen in the following derivation of the sampler for the GaP [9] model:

$$\begin{aligned} \phi_{k,v} &\sim Gamma(c_v, d_v) \\ \theta_{d,k} &\sim Gamma(a_k, b_k) \\ n_{d,v} &\sim Poisson((\theta\Phi)_{d,v}) \end{aligned}$$

The following identifiability [15] constraint also applies.

$$\sum_v \Phi_{kv} = 1 \tag{5.1}$$

We also apply the following augmentation to break $n_{d,v}$ into parts:

$$\begin{aligned} n_{d,v} &= \sum_k n_{d,v,k} \\ n_{d,v,k} &\sim Poisson(\theta_{d,k}\phi_{k,v}) \\ p(n_{d,v,k}|\theta,\phi) &= \frac{(\theta_{d,k}\phi_{k,v})^{n_{d,v,k}} e^{-\theta_{d,k}\phi_{k,v}}}{n_{d,v,k}!} \end{aligned}$$

Simplification proceeds as follows:

Index simplification (as per section 5.3.4.1)

$$
\begin{aligned}
\prod_{d,v,k} \quad & \frac{(\theta_{d,k}\phi_{k,v})^{n_{d,v,k}} e^{-\theta_{d,k}\phi_{k,v}}}{n_{d,v,k}!} \\
= \prod_{d,v,k} \quad & \frac{\theta_{d,k}^{n_{d,v,k}} \phi_{k,v}^{n_{d,v,k}} e^{-\theta_{d,k}\phi_{k,v}}}{n_{d,v,k}!} \\
= (\prod_{d,v,k} \quad & \theta_{d,k}^{n_{d,v,k}})(\prod_{d,v,k} \phi_{k,v}^{n_{d,v,k}})(\prod_{d,v,k} \frac{e^{-\theta_{d,k}\phi_{k,v}}}{n_{d,v,k}!}) \\
= (\prod_{d,k} \quad & \theta_{d,k}^{\sum_v n_{d,v,k}})(\prod_{v,k} \phi_{k,v}^{\sum_d n_{d,v,k}})(\prod_{d,v,k} \frac{e^{-\theta_{d,k}\phi_{k,v}}}{n_{d,v,k}!}) \\
= (\prod_{d,k} \quad & \theta_{d,k}^{n_{d,(\cdot),k}})(\prod_{v,k} \phi_{k,v}^{n_{(\cdot),v,k}})(\prod_{d,v,k} \frac{e^{-\theta_{d,k}\phi_{k,v}}}{n_{d,v,k}!})
\end{aligned}
$$

Simplification using the identifiability constraint (as per section 5.3.4.2)

$$
\begin{aligned}
\sum_v \quad & \Phi_{k,v} = 1 \\
\prod_{d,v,k} \quad & e^{-\theta_{d,k}\phi_{k,v}} \\
= \prod_{d,k} \quad & \prod_v e^{-\theta_{d,k}\phi_{k,v}} \\
= \prod_{d,k} \quad & e^{\sum_v -\theta_{d,k}\phi_{k,v}} \\
= \prod_{d,k} \quad & e^{-\theta_{d,k}(\sum_v \phi_{k,v})} \\
= \prod_{d,k} \quad & e^{-\theta_{d,k}}
\end{aligned}
$$

Finally,

$$
p(n,\theta,\phi) = \prod_{d,k} \theta_{d,k}^{n_{d,(\cdot),k}} \prod_{v,k} \phi_{k,v}^{n_{(\cdot),v,k}} \prod_{d,k} \quad e^{-\theta_{d,k}} \prod_{d,v,k} \frac{1}{n_{d,v,k}!} p(\theta)p(\phi)
$$

$$
p(\theta) = \prod_{d,k} \frac{a_k^{b_k}}{\tau(a_k)} \theta_{d,k}^{a_k - 1} e^{-\theta_{d,k}^{b_k}}
$$

The prior for ϕ is similar to that of θ. Therefore, the analysis simplification are similar as well (upto the identifiability simplification).

$$\begin{aligned} p(n,\theta,\phi) &= \prod_{d,k} \theta_{d,k}^{n_{d,(\cdot),k}+a_k-1} \prod_{v,k} \phi_{k,v}^{n_{(\cdot),v,k}+c_v-1} \prod_{d,k} e^{-(b_k+1)\theta_{d,k}} \prod_{d,v,k} \frac{1}{n_{d,v,k}!} \\ &= \prod_{d,k} \theta_{d,k}^{n_{d,(\cdot),k}+a_k-1} e^{-(b_k+1)\theta_{d,k}} \prod_{v,k} \phi_{k,v}^{n_{(\cdot),v,k}+c_v-1} \prod_{d,v,k} \frac{1}{n_{d,v,k}!} \end{aligned}$$

Collapsing using Dirichlet and Gamma normalizers (applied as symbolic modifiers based on pattern matching during the variant generation process).

$$\frac{\Gamma(n_{(\cdot),k,v}+c_v)}{\Gamma(n_{(\cdot),k,(\cdot)}+c_v)} \frac{(b_k+1)^{(n_{d,(\cdot),k}+a_k)}}{\tau(n_{d,(\cdot),k}+a_k)}$$

which gives the update equation for $n_{d,v,k}$

$$\frac{n_{(\cdot),k,v}+c_v}{n_{(\cdot),(\cdot),k}+c_v} \frac{b_k+1}{n_{d,(\cdot),k}+a_k}$$

5.3.6 Variant Generator

This section provides a brief overview of the variant generator as part of the architecture. Chapter 6 contains a detailed overview of this system. The variant generator is responsible for providing different variants for the same initial model specification. This allows for the creation of different sampling strategies for the parameters of the model. The general workflow followed by the variant generator is to run a depth first search (DFS) over the graph with changes at each level of recursion occurring based on the operation carried out at that level.

At a very high level, the algorithm for this generation process is as follows:

VariantDFS(Graph G):
AttemptSamplerGeneration(G)
G.generateOperations()
for *all operations* $o \in G$ **do**
    G.performOperation(o)
    VariantDFS(G)
    G.undoOperation(o)
end

Algorithm 10: Variant Generation Search Algorithm

5.3.7 SymPy wrapper

A wrapper has been implemented to enable some functionality of SymPy within the system. The main functionality exposed from SymPy is as follows:

- Pretty printer: The SymPy prettyprinter is used for visualization of formulae
- Expression Multiplier: The multiply function from SymPy is used for symbolic processing of likelihoods in rare situations. Note the clash arising here in terms of the type of standard representations used, which is discussed further in section 5.4.1.
- Latex printer: The SymPy latex printer is used to enable the extraction of latex formulae from the model.

5.3.8 Graph Processor

The graph processor is primarily used in updating graphical structures during the variant search process. In addition to carrying out the obvious operations of removing a node for collapsing and adding a node for augmentation, this module also updates the corresponding connectivity in the graph and maintains references to previously removed (or added) nodes which may need to be removed (added) in future operations (for situations such as performing undo operations during variant search).

5.4 Sampler Generation

Once variant generation is complete. The results are analyzed in order to generate relevant samplers for the parameters present in the graph. This section provides more details on the different components involved in the sampler generation process.

5.4.1 Symbolic Representation

Due to the need to support complex symbolic operations, several types of symbolic representations are used within the workflow of the model analysis process. These are explained in more detail below with examples of the different representations of the same formula. An example is given in figure 5.2. .

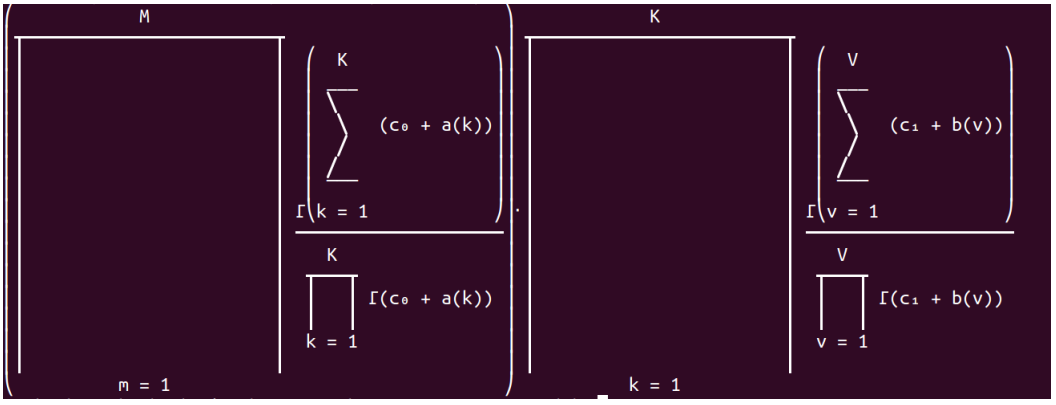
Standard Representation
This representation is used for the initial generation of the likelihood and also for simple pattern matching operations.

```
(Product(Gamma(Sum(c0+a(k),(k,1,K)))/Product(Gamma(c0+a(k)),(k,1,K)),(m,1,M))
*Product(Gamma(Sum(c1+b(v),(v,1,V)))/Product(Gamma(c1+b(v)),(v,1,V)),(k,1,K)))
```

Here, Product(F(k),(k,1,K)) $= \prod_{k=1}^{K} F(k)$ and Sum(F(k),(k,1,K)) $= \sum_{k=1}^{K} F(k)$ and these two constructs are used to build the overall expression in string format.

Formula Representation

This is a format used for visualization of the standard format and is more reader-friendly.



Latex Representation

Conversion to Latex is supported to enable easier integration of model formulae into papers and reports. The example denotes first the Latex code, followed by its compiled version.

```
\left(\prod_{m=1}^{M} \frac{\Gamma{\left (\sum_{k=1}^{K} \left(c_{0} + a{\left
(k \right )}\right) \right )}}{\prod_{k=1}^{K} \Gamma{\left (c_{0} + a{\left
(k \right )} \right )}}\right) \prod_{k=1}^{K} \frac{\Gamma{\left (
\sum_{v=1}^{V} \left(c_{1} + b{\left (v \right )}\right) \right )}}{
\prod_{v=1}^{V} \Gamma{\left (c_{1} + b{\left (v \right )} \right )}}
```

$$\left(\prod_{m=1}^{M} \frac{\Gamma\left(\sum_{k=1}^{K} \left(c_{0} + a(k)\right)\right)}{\prod_{k=1}^{K} \Gamma\left(c_{0} + a(k)\right)}\right) \prod_{k=1}^{K} \frac{\Gamma\left(\sum_{v=1}^{V} \left(c_{1} + b(v)\right)\right)}{\prod_{v=1}^{V} \Gamma\left(c_{1} + b(v)\right)}$$

Dot Representation

The Dot representation enables expression trees to be converted into a graphical form which is easier to print. GraphViz is used to render the resulting representation for visualization.

Expression Tree Representation

The expression tree representation is a tree based representation which is more conducive for complex pattern matching operations. An example is provided in 5.4.1. The

Cacheable Tree structure and the Computable Tree representations structure are extended from this representation and are discussed in detail under the relevant sections 5.4.2.6 and 5.4.2.7.

5.4.1.1 Implications on pattern matching

While the developed pattern matching schemes can be applied on any of the above representations, it was primarily developed with the Standard Representation in mind. This is because this representation is:

- easier to debug,
- can be converted to and from all other representations, and
- offers the possibility for incorporation into other probabilistic programming languages due to the fact that it is easily generated from a given model.

As such, complex pattern matching was implemented as a string matching algorithm applicable to the standard representation. There are two forms of the standard representation that need to be considered.

The **simplified normal form** contains the likelihood that is most commonly used for visualization, with similar terms aggregated together. However, this increases the difficulty of matching terms during the symbolic pattern matching process. To simplify the pattern matching process, likelihood representations in the standard form are maintained in an **expanded normal form** with no major simplification being performed on it prior to visualization. This has the added benefit of saving computation time required to perform symbolic multiplication in the likelihood computation stage, as this is only necessary during the final stages of model reporting when human readable output is generated.

While at its most basic, the expression tree only contains symbolic information, extensions were done in order to simplify many different computational aspects of the system. Since the expression tree contains nodes, these were strategically extended with more information about the symbols they contained. This is especially true in parameter or counter (sufficient statistic) nodes in symbolic format. Additional information such as node type, dimensions, symbolic reasoning constraints, distributional information were all added into the tree in order to simplify several processes such as pattern matching, code generation, exection and symbolic simplification.

5.4.2 Sampling Manager

The sampling manager is responsible for the overall sampling strategy for the inference process, including deciding on which sampler to use for sampling a particular variable. When the system is operating in simulation mode, it also maintains the sampling data structures between iterations of the Markov chain and contains mechanisms to ensure that the state of the chain is consistent and that caches are updated automatically.

5.4.2.1 Gibbs Sampler

The Gibbs sampler implements a sampler for the model based on Gibbs sampling. Gibbs sampling is a Markov chain Monte Carlo (MCMC) algorithm for generating a sequence of samples which are drawn approximately from a specified probability distribution, when direct sampling from the distribution in question is difficult.

Gibbs sampling constructs a Markov chain of samples, with nearby samples displaying a high degree of correlation. Samples generated by the initial parts of the chain do not generally represent the desired distribution, and a burn-in period is introduced during which samples are discarded. Gibbs sampling is a special case of the Metropolis Hastings algorithm. The statistical operations mentioned above (marginalizing and augmenting) can be used to simplify the model in order to improve Gibbs sampling.

Gibbs sampler
Initialize from prior $x^{(0)} \sim q(x)$
for *i = 1,2,...* **do**
$x_1^{(i)} \sim p(X_1 = x_1 | X_2 = x_2^{(i-1)}, X_3 = x_3^{(i-1)}, \ldots, X_D = x_D^{(i-1)})$
$x_2^{(i)} \sim p(X_2 = x_2 | X_1 = x_1^{(i)}, X_3 = x_3^{(i-1)}, \ldots, X_D = x_D^{(i-1)})$
$\vdots$
$x_D^{(i)} \sim p(X_D = x_D | X_1 = x_1^{(i)}, X_2 = x_2^{(i)}, \ldots, X_{D-1} = x_{D-1}^{(i)})$
end

The construction of the Gibbs sampler relies heavily on the symbolic processor in order to generate the full conditional likelihood.

5.4.2.2 Sampled Parameter

This data structure maintains the state of parameter variables in the Markov chain. As sampling for a particular parameter is carried out, statistics which rely on that parameter are updated automatically by the Cache Updater.

The sampled parameter provides two main types of sampling functionality:

- Carrying out the **initialization** of the parameter state during the instantiation of the Markov chain.
- Performing **sampling updates** and propogating those updates throughtout the network via the cache updater.

5.4.2.3 Cached Statistic

This data structure maintains the state of sufficient statistics in the Markov chain. These cached structures allow the efficient look-up of quantities crucial to the computation of the full conditional probability in O(1) time and provide considerable savings in run-time for the sampling process.

There are two main types:

- Primary statistics are derived directly from the model graph and depend on the distributions and parameters they relate to. These are updated when the relevant parameters are sampled.
- Secondary statistics are derived from primary statistics and require further symbolic reasoning to set up. These are updated whenever the relevant primary statistic is updated.

5.4.2.4 Data Block

This data structure maintains the data variables and arrays required for the computation of the full conditional probability of a parameter. In the case where transformed versions of the data is required for this computation (for example, a sum over a particular row of a data array), this is identified ahead of time through symbolic reasoning and is also stored as a separate Data Block. This ensures that all data lookup required by the model can be performed in O(1) time.

5.4.2.5 Cache Updater

Automatic updates of statistic caches is a complex process and is handled by the cache updater. The system uses encapsulation to ensure each component of the update process only has access to the information directly relevant to it. A message passing system is used to facilitate the cascading of updates. The software engineering design pattern of Observer-Observable [23] is used to achieve this.

Each primary cached statistic is an observer of the parameters it is related to, while each secondary cached statistic is an observer of the primary cached statistic it is derived from. This means that a sampled parameter can only ever be an observable, while a cached statistic may be an observer,an observable or both. Accordingly the parameter class inherits from the observable class, while the statistic class inherits from the observable class and implements the observer interface (multiple inheritance).

When a sampling step is carried out, the sampled value and other contextual information regarding the sampled parameter are passed on to the Cache Updater, which proceeds to forward that information to the relevant observers of that parameters. These observers use symbolic reasoning to identify the structure of the update they need to carry out to maintain their integrity, followed by launching another update to *their* observers leading to a cascade of updates. Memoization is used to enable saving symbolic links to any required look-up terms so that future updates of the same type result in considerably reduced symbolic reasoning overhead.

In situations where a cache update is triggered with potentially one or more dimensions of the update being unknown, this is indicative of the need to perform an update for all elements of that dimension. This usually arises in situations where a parameter being sampled is also an index for another parameter. Consider a clustering model where a cluster is drawn using a categorical distribution. This cluster indicator is then used to index the probabilities unique to that cluster. Therefore, when the cluster is sampled, any statistics arising related to that cluster would need to be updated along with the new assignment for the cluster. If there are multiple items associated with the cluster (for example, multiple documents), it would be necessary to update the relevant statistics for all of these items. This situation is automatically identified and handled by the cache updater by performing a lookup on the missing indices' range, and performing the planned update for all values of the index in that range.

5.4.2.6 Computable Tree

The sampling step for a parameter involves the computation of the full conditional probability for that parameter based on the data structures discussed. As this step is repeated continuously throughout the Markov chain, it is essential that this step be carried out as efficiently as possible. The computable tree is a tree-based representation of the full conditional probability that achieves this. It is extended from the expression tree representation, but additionally provides symbolic links to the requisite elements. During the first call of the sampling step for a particular parameter, the computable tree contains a symbolic representation of the full conditional probability formula which is used in order to symbolically reason about the terms that the formula requires. Once this is done, the result is memoized in the corresponding node of the computable tree

as a direct symbolic link to the term in question. This allows future traversals of the computable tree the capability of looking up relevant terms in O(1) time. If this was not memoized, the symbolic steps associated with computing the entity would create an overhead every time it was invoked.

5.4.2.7 Cacheable Tree

The Cacheable Tree representation is used in order to generate the secondary cached statistics required for a particular sampler. This tree based representation extends the expression tree format but additionally isolates computational terms related to primary statistics. This allows for the replacement of this computation with a new variable in the symbolic representation of the full conditional probability. This same process is also carried out for transformed data variables which allows for efficient look-ups of all data terms in constant time. This is also a strategy for trading memory space for running time as the result would now need to be stored, but it removes the need to run, for example, addition over all values in a cache, in order to compute it's sum.

5.5 Simulation and code generation

Once all samplers have been processed, it is possible to use them in order to generate code or to execute the model. This section describes the modules involved in this process.

5.5.1 Simulator

The simulator is responsible for executing the model variant developed during the analysis phase. The final result of the analysis phase is a collection of samplers corresponding to the different parameters of the initial model. The code simulator has jurisdiction over deciding which parameters are sampled and in which order. The simulator provides a sandbox environment for the model analysis and code generation process as more contextual information is available at each stage of the model. This allows for more robust error reporting and more dynamic sampling strategies compared to the code generated version of the model.

The simulation process proceeds as follows:

1. Variant is checked to see if each active parameter has a viable sampler. The parameters are also checked for lexicographical ordering, to ensure code is generated in the correct order.

2. Each sampler is prepared for sampling. This involves performing symbolic reasoning to identify the form of the update equation, determine the range of each sampling loop and identify all cached statistics that would be updated based on the parameter under consideration.

3. Based on update equations, distributions types and other sampler information, the initialization for the parameter is carried out.

4. When all parameters have been initialized, training is intitated.

5. Based on sampler information, the equation for updating this parameter in each epoch is determined and memoized for future reuse. It is converted into a format that can have values substituted in quickly, and computed quickly as well. This is important because this computation is repeated many times.

6. Execution is repeated until convergence/maximum limit of iterations is reached.

7. Results are visualized and any metrics are computed and saved.

During simulation, any parameter updates will trigger automatic updates of all related formulae and statistics. This is explained in more detail under section 5.4.2.5.

5.5.2 Code Generator

The code generator handles the code generation functions of the system. In general, generated code will be simple and easy to understand, allowing for the possibility of relatively straightforward empirical analysis compared to simulated models.

Code generation proceeds as follows:

1. Variant is checked to see if each active parameter has a viable sampler. The parameters are also checked for lexicographical ordering, to ensure code is generated in the correct order.

2. Each sampler undergoes initial setup. This involves preparing the necessary data structures on a per-sampler basis.

3. Update equations are generated for each sampler.

4. Based on update equations, distributions types and other sampler information, the initialization for the parameter is determined.

5. Code for initialization is generated. This is added to the abstract code for this variant, under initialization code.

6. Where the sampler is the result of an operation (e.g. augmentation), any additional simplifications, initializations and routines are determined and added to the abstract code for this variant, under the relevant sections.

7. Based on sampler information, the code for updating this parameter in each epoch is determined. This is added to the abstract code for this variant, under update code.

8. Internal validation is carried out on the generated abstract code using a dummy dataset. If verification fails, any errors are reported and execution is halted.

9. Boilerplate code is generated for the intialization process, in the desired target language.

10. Code for the initialization process is generated based off the initialization section from the abstract code structure.

11. Boilerplate code is generated for the training iteration.

12. Code for the updating process is generated based on the abstract code structure.

13. Code for visualizing results/intermediate metrics is generated by linking with existing helper functions in the system standard library.

14. A final compilation test is carried out to check if the code compiles correctly with no errors. This is done with a compiler/interpreter for the target language.

It should be noted that many of the symbolic steps required for simulation are also carried out during code generation, but are not explicitly mentioned for the sake of brevity. Code generation is carried out on a variant once all samplers have been generated. In order to validate the robustness of the generated code, the generated abstract code structure is used to simulate the model on an internally generated dummy dataset. Internal validation helps to identify any potential issues with the generated code such as statistic updates, semantic errors in the model specification or the schema. This step makes use of the existing support for simulation present in the system. Since the behaviour of the model is predictable in any given epoch (notwithstanding issues in convergence), it is possible to use a much smaller automatically generated data-set to identify any potential crashes caused by semantic errors such as incorrect index usage in the model, incorrect placement of statements in the schema and any other semantic error that would cause crashes in the system.

Once validation is complete, the abstract code structure can be compiled into the target programming language. Support for different languages is provided by means of a language configuration file, which maps between the abstract code language and the target language.

An excerpt of the generated code for MetaLDA is shown below. Comments have been added manually to provide further clarity.

```
for (int d = 0; d < D; d++) {
  int I = w[d].length;
  for (int i = 0; i < I; i++) {
    \\ c0[][] is document-topic stats
    c0[d][z[d][i]]--;
    c0_1[d]--; \\ row totals of c0[][]
    \\ c1[][] is topic-word stats
    c1[z[d][i]][w[d][i]]--;
    c1_1[z[d][i]]--;  \\ row totals of c1[][]
    double[] p = new double[K];
    for (int k = 0; k < K; k++) {
      p[k] = Math.pow(c0_1[d]+a_1[d],-1)*
      Math.pow(c1_1[k]+b_1[k],-1)*
      (c0[d][k]+a[d][k]) *
      (c1[k][w[d][i]]+b[k][w[d][i]]);
    }
    for (int k = 1; k < K; k++)
      p[k] += p[k - 1];
    int k;
    double val=Math.random()*p[K-1];
    for (k = 0; k < K; k++) {
      if (p[k] > val) break;
    }
    z[d][i] = k;
    c0_1[d]++;
    c0[d][z[d][i]]++;
    c1_1[z[d][i]]++;
    c1[z[d][i]][w[d][i]]++;
  }
}
```

Someone experienced in implementing the collapsed Gibbs sampler for LDA [30] would be familiar with the incrementing and decrementing of totals done here. Many steps have been taken to simplify the form of the representation. For example, subtraction and division are not treated as individual operations, but as inverse operations of addition and multiplication (addition with negative value, and multiplication by 1/value). These can be checked and updated during the code generation stage if necessary, for example matching with regular expressions for "Math.pow(X, -1)" would be sufficient to find all instances where division would appear.

There are several key advantages of generating simple code as in the example above:

- Easy to understand and debug.

- Easy to derive complexity (both time and space).
- Ability to extend using custom code in the generated language.

5.5.2.1 Multiple Language Support

Due to the design considerations mentioned in section 4.1.1, primarily language simplicity, the code generated by the system is straightforward and not reliant on any classes, interfaces or high level abstractions in the code. It is generated at a level where a beginner programmer would be able understand the mechanics of the algorithm, but perhaps not the underlying statistical reasoning. However, to someone accomplished in statistical analysis, the simple code provides an easy means of interacting with the generated algorithm.

In initial versions of the system, the code was generated in a single language, however, due to the limited set of operations supported due to the need for simplicity, the possibility for further abstraction became apparent. The design of the system incorporates the generation of Abstract Code Structures, which can be used to represent a generalized version of the code that would otherwise be used to run the algorithm.

The support for programming language constructs in the system is as follows:

- Support for basic types (Integer, Floating point, etc).
- Support for array-like structures (static and dynamic).
- Simple iteration - of the form for i = 1:N, and not of the form for (s in container) which would be more language dependent.
- simple function calls (primarily to support sampling calls corresponding to basic distributions - the code for these calls is encapsulated in separate functions to avoid needless bloating of the generated code, and to also simplify it for more readability). The function code is available in a helper file which is generated alongside the main algorithm code for inspection by users.

These are the abstract constructs currently supported by the system. Multiple language support is provided by specializing the abstract code for different languages using templates. Currently templates exist for Java and C++.

5.5.2.2 Data

Data can be provided to the system in many different forms. While CSV (Comma seperated value) is the expected format for data, within this format many different styles are supported.

Data can contain string tokens, variable sized lists and sparse data. Where a variable indicated as data is provided in a sparse format, the sparse format is maintained internally as well. Mappings and inverse mappings (for example feature document mapping and its inverse in MetaLDA, discussed in section 4.3) are supported by the system, allowing for straight-forward use both within code and schema.

Additionally, results can be logged seperately during:

- initialization,
- over the burn-in period
- over other training epochs, and
- over user-defined ranges of training epochs.

5.5.3 Comparison of Simulation vs Code generation.

As simulation and code generation are the main workflows in the system, this section contains a comparison between them.

Symbolic reasoning
All symbolic reasoning must be done during the code generation phase for generated code, whereas in the simulated models, some of this can be done at run-time with memoization of reasoned results leading to overall better performance as compared to simulation without memoization.

Cache updates
During the simulation phase cache updates are handled implicitly by the system through the message passing system, while during code generation additional symbolic reasoning is carried out to ensure the code contains simpler explicit cache updates.

Data structures
Structures used in simulation are encapsulated and provide more flexibility in terms of implementation, as only the end result of the sampling process is shown to users. During code generation, structures are restricted to standard language data structures such as arrays, which limits the options in terms of what data structures can be used.

Performance
The overall performance of simulation will be lower due to containing more complex data structures and performing more reasoning during execution, while the simpler structure of generated code and the possibility of further compiler optimization leads to generated code being faster in terms of execution. This is somewhat similar to the comparison

between compiled vs interpreted langauges, since during simulation each step must be "interpreted" individually.

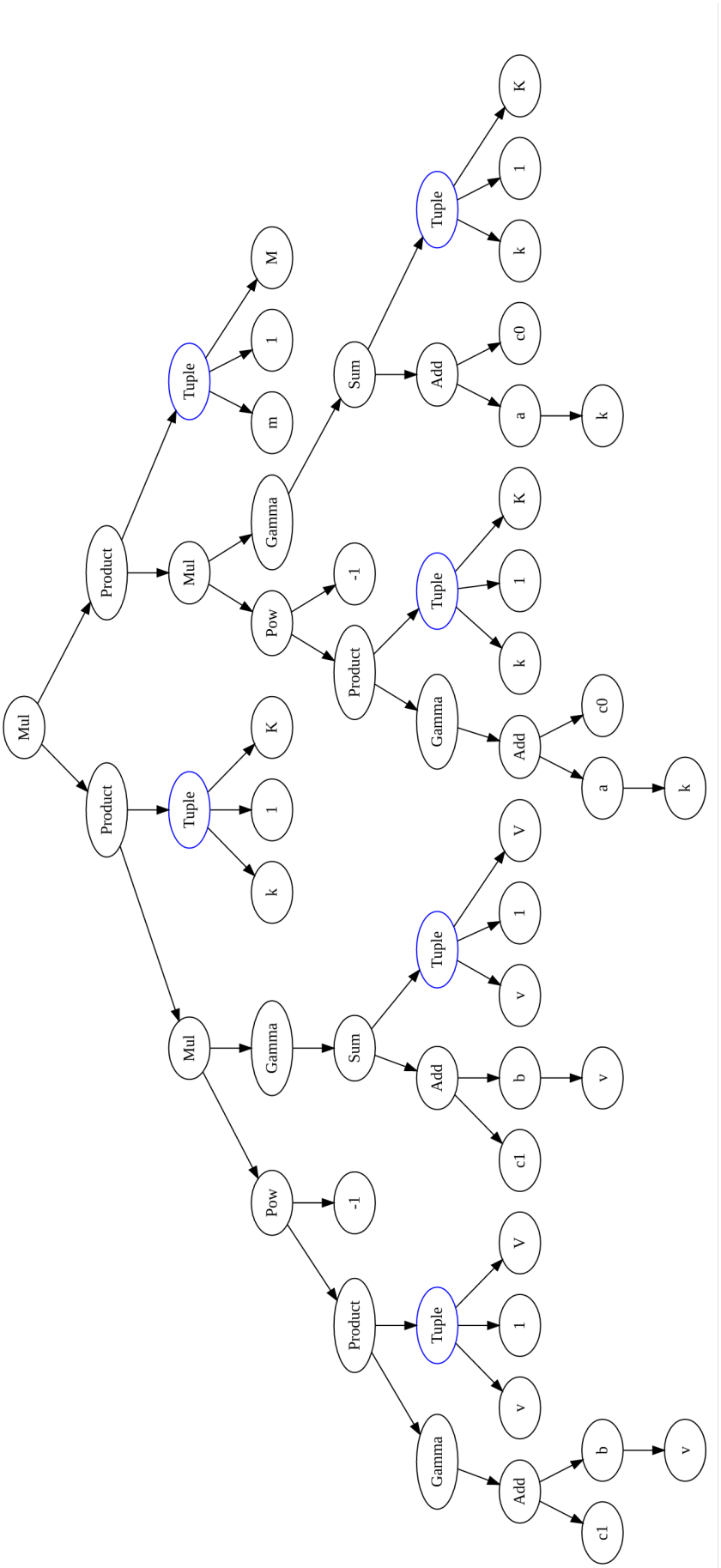


FIGURE 5.2: Expression Tree for LDA model likelihood

Chapter 6

Properties of Variant Generation

This chapter provides details regarding the variant generation process employed by the system. While Chapters 4.1.1 and 5 respectively approach this topic from a design and architectural perspective, this chapter provides a more detailed look at the inner workings of variant generation and schema. In particular, proofs are provided of some interesting properties of the operations generated by the schema-based system, and we also show that the variant search process used by the system is able to generate all possible variants for that model specification.

Schema were introduced into the system as a way of generalizing symbolic operations into a more flexible form, but was extended as a general framework for supporting all operations that can be applied to a given intermediate variant (graph and likelihood pair). Schemas are used as a means of providing a blueprint for operations in the variant generation process like collapsing, augmenting and simplification.

6.1 Schema Templates

This section describes the general template used for schemas.

Schema Template	
Schema field	Explanation
Name	Used as an identifier for the schema. Useful in debugging.
Description	A description of what the schema does. Useful for inclusion in the logs generated during the model analysis process.
Type	Type of schema. Possibilities include collapsing, augmentation, simplification or custom.
Applicability Constraint	Indicates the conditions needed for the schema to be applied. Can include wildcards for pattern matching, simple patterns and graphical patterns.
Handling multiple matches	Indicates the method of handling the presence of multiple matches of the applicability constraint. Options include generating all operations or generating a single operation (based on first match).
Action	Actions to be performed on graph/likelihood on confirmed presence of the applicability constraint.
Unique action?	A flag for indicating if the action taken is unique for the generated match. If true, when the same match occurs elsewhere in the variant generation process, the first instance of this action will be duplicated rather than generating a new instance of the action.
Unique signature	If Unique action? is set to true, a signature for identifying the particular instance of the action must be provided in the form of a string. The string may use identified terms from the applicability constraints (wildcards etc).
Match schema action	In the case the current variant matches the applicability criteria, the action taken on the part of this schema. Options include REMOVE - remove from the queue of schema, REQUEUE - remove from the current position and add to the end of the schema queue or NONE - leave schema is same position
No match schema action	In the case the current variant does not match the applicability criteria, the action taken on the part of this schema. Options include REMOVE - remove from the queue of schema, REQUEUE - add to the end of the schema queue or NONE - do nothing.
pre-simplify?	Indicates whether the likelihood should be simplified into the **simplified normal form** prior to applying the pattern matching operation. This form is discussed in more details under section 5.4.1.1. When set to false, pattern matching is conducted on the **expanded normal form** representation.

While many of these fields are self-explanatory, several could benefit from further discussion, as presented below. Further, an example schema is given in Table 6.1 and more are presented in Appendix B.

Applicability Constraint is primarily intended as a string matching operation, where a likelihood constraint is applied. The matching of this constraint or "pattern" using the complex pattern matching algorithm described under symbolic processing forms the

trigger for the activation of the schema. Additionally, it is possible to define constraints on the graph itself, using properties of the nodes themselves. This is especially useful where the nodes are parameters, and have properties that need to be exploited for the variant generation process. An example of this is in collapsing, where conjugacy between a node and its prior is a prerequisite. In this case, several checks may be conducted:

1. the type of a pair of nodes suspected of being conjugate to each other;
2. adjacency (direct connectivity) between the two nodes, including directionality, since clearly, the prior needs to be a parent of the node being checked for collapsing; and
3. pattern matching based on conjugacy pattern: this includes defining the general form of the conjugate likelihood using wildcards. The corresponding string representation is used in complex pattern matching to identify the presence of such a pattern, thus validating the presence of this conjugacy.

Table 6.1: Example Schema for Dirichlet Multinomial Collapsing

Schema field	**Value**
Name	CollapserDirMult
Description	Performs collapsing according to the Dirichlet multinomial Conjugacy.
Type	Collapsing Schema
Applicability Constraint	b: parameter, type is categorical or multinomial a: parameter, type is Dirichlet, prior of b
Handling multiple matches	Generate all matches as operations.
Action	Remove b: introduce a new constant node a_b with the terms from collapsing operation (based on the normalizer for Dirichlet multinomial)

The syntax used in the system is more code-like in nature, whereas the schema here has been provided with the intention of explaining the operations for clarity. Note that when certain fields from the initial schema template are missing, they are assigned default values by the system. This is to simplify the process of schema creation, but still provide flexibility as needed.

A benefit of the modularity of the system should become apparent from the above, as any given conjugacy only needs to be defined once, using a minimalistic way of representing the key properties relating to the statistical behavior displayed by the collapsing operation.

Handling multiple matches is also a very important consideration for a schema. For example, the proof of correctness of the variant generation process, discussed in section

6.3, assumes the generation of all matches pertaining to a particular schema applicable to the variant under consideration at that time. This is also discussed in more detail under section 6.4, where the proof relies on all operations that can be generated from a schema for a particular variant instance, being generated in a single pass of the schema, rather than relying on recursion to generate multiple different operations from the same schema being examined multiple times. This also has the added benefit of improving the run-time of the algorithm, as the number of recursive calls and duplicated work (in terms of setting up data structures to do pattern matching etc.) is reduced considerably. It also provides the opportunity for considerable object reuse at the code level, leading to better software design as well.

This shows the design principle behind schema being a forward-looking mechanism, intended to support usage that users might come up with, and having the support for such usage ready ahead of time or attainable with a relatively low amount of future work.

Unique action? is a flag used for understanding schema behavior in relation to multiple occurrences of the exact same operation arising multiple times due to the backtracking/recursion process. By default this is set to true, however the option to set it to false is also provided to users, in case it is needed. Setting this to true, is an example of memoization where computing time is traded for memory. The idea is that if the same function is called with the same arguments, the result will be the same. Therefore, by saving the result the first time around, running time can be reduced in consequent calls, at the cost of the memory used to store those results, and the information required to identify that instance.

6.1.1 Operation Memoization

There is a unique benefit for augmentation which arises from the fact that a new parameter needs to be generated. Generally, the system generates these with an arbitrary character ("q") and by keeping track of the count of currently conducted augmentations. So augmented parameters will follow a series like $q_1, q_2, ..., q_{count}$. Clearly, these need to be named uniquely as there may be overlap between when these parameters exist in the same variant, and even across multiple variants, it is better to have different names rather than reused names to provide better readability - if two algorithm variants both have a q_1 where one is different to the other, it serves to confuse users. The exception to this is when there are two parameters q_m and q_n which both actually correspond to the same parameter, but have been named differently. However, this situation can be avoided by the mechanism mentioned above, where the inputs used to generate the operation of the schema are used as the "signature" to memoize the resultant operation.

The next time the same input, schema combination (i.e. the "signature") is seen the result is simply returned without any additional computations.

Clearly, this situation does not require the exact same variant which was processed alongside the schema to be present in both situations (leading to q_m and q_n), but rather, the component which caused the identical match in both cases. For example, if component B causes the match, then both A*B*C and E*B*F could be valid likelihoods for two different variants that trigger the match. Thus, the *uniquesignature* needs to be allowed access to the matches of the pattern matching operations in order to ensure the uniqueness of the match and to avoid potential false positives when checking the schema on a different, yet similar component of the likelihood/graph.

6.1.2 Benefits of Modularity

There are some key advantages to using a modular schema based system for managing the variant generation process.

- Ability to provide customizability
- Reusability of schema
- Ability to maintain a schema library for users (plug and play)
- Approachability for users with lower programming expertise
- Flexibility and optimizability for advanced users

The support for custom schema provides a large degree of **customizability** as users can implement their own schema and change how the overall algorithm generation process is instigated. The fact that schema are self-contained and are not strictly dependent on the system but are rather decoupled from the system means that there is **modular reusability** over multiple models. In fact, it is possible to maintain a library of schema for users to select from. This means it is much easier for newer programmers to use the schema to conduct algorithm generation on a given model. They can simply use all the available schema to see if a suitable algorithm is generated. This provides an almost black-box approach for users who do not wish to be mired down with low level detail. For more expert users who would like to work in more detail, there is the possibility of fine-tuning the selection of schema, and potentially adding their own. Notably, expert users (with sound knowledge of statistical modelling) will be able to improve the running time of the variant generation process by manually removing schema that they deem to be unnecessary.

6.2 Expanded Variant Generation Process

This section provides an overview of how the variant generation process occurs in more depth with a focus on the interplay between schema, operations and variant generation.

For the purpose of detailed technical discussion, a variant V is defined by $V(G, L, O, S)$ Where G is a graph, L a likelihood, O a set of pending operations and S a set of schema that would be applicable to at least one variant but whose applicability to any particular variant would first be checked before an operation is generated. The seed variant $V_0(G_0, L_0, O_0, S_0)$ is generated by the initial parsing of the model specification and by generating likelihoods for each node in the model specification. The set of schema S_0 is generated based on the schema provided to the system (users may add or remove schema before running the system as they please). The initial set of operations O_0 is empty.

The following algorithm 11 is a conceptual overview of how schema and operations are used in variant generation. Note that the depth(d) of the recursive tree is also tracked alongside the list of operations already performed on a variant in order to provide a more complete overview once the variant generation process is complete. Note that while S is present in the formal definition of a variant, it is absent in the recursive algorithm due to the fact that it is not modified by the algorithm in any way and thus is not part of the managed state of the variant, and is thus referenced as a global data structure.

For the purpose of algorithm clarity we have split up the queue of operations as *Op* - the container for pending operations, and *Od* the container for completed or "done" operations. Notice that the underlying algorithm does not depend on the container used (list, queue, stack etc. are all fine). For understanding this algorithm considering it as a stack will be more intuitive.

With the given algorithm 11, processing of a variant for sampler generation only happens in a recursive call if no operations are applied in that call, and if no schema are triggered in that call as well. In terms of the recursive call tree, this node would then not have any children (due to to not triggering any recursive calls) and would thus be a leaf node.

6.2.1 Ordering Operations

The ordering of schema and recursive calls is intentionally set up so that all possible orderings of schema are supported. This is to allow for any permutation of schema to be generated, thus making the variant generation process independent of the order in which schema are specified in the configuration of the system. This is important as it allows users to define schema without worrying about potential implications of having

```
VariantGeneration(G, L, Opn, Od):
if Opn is not empty then
    //recurse without performing operation
    o = Opn.pop()
    VariantGeneration(G, L, Opn, Od)
    //perform operation o on G and L
    G', L' = performOperation(o, G, L)
    //o is now done and no longer pending
    Od.push(o)
    G' and L' are new instances
    VariantGeneration(G', L', Opn, Odn)
    //reset current state for maintaining proper DFS
    Od.pop()
end
if there are schema applicable to (G, L) then
    //generate new operatons, recurse
    for all applicable schema s ∈ S do
        //Multiple operations are possible from a single schema
        O = s.generateOperations(G,L)
        for all operations o ∈ O do
            Opn.push(o)
        end
        VariantGeneration(G, L, Opn, Odn)
        //Maintain DFS state
        for all operations o ∈ O do
            Opn.pop()
        end
    end
end
//Check if any operations were performed or generated at this level(depth)
If not, this is a leaf node in the recursive tree
if this is a leaf node then
    registerAlgorithmCandidate(G, L)
end
```

Algorithm 11: Variant Generation Search Algorithm

them in the incorrect order. There are also options available for advanced users to take advantage of with ordering schema in a particular ordering (for example by imposing $S_1 > S_2 > S_3 > ... > S_n$ in terms of schema priority for execution, and additionally enforcing that this ordering is maintained in the variant generation). This would also provide faster execution of the variant generation process, since all permutations of the schema do not need to be considered.

In ensuring that any ordering of schema is possible, it is possible that schema that do not overlap with each other, i.e. operate on different parts of the graph, can be applied to the same graph in different orderings. This would lead to multiple occurrences of the same variant at different stages of the variant generation process. The handling of this is primarily done by identifying duplicate variants by hashing the likelihood function.

Note that the hashing in this case can be handled by first splitting over multiplication and sorting by alphabetical order to remove a large degree of the issues surrounding commutativity and associativity. Thus, entire branches of the recursive call tree can be pruned to optimize efficiency. Note that, since the set of schema does not change for the variant generation process, it is safe to prune such a branch. This is because all the schema available to the original occurrence of the branch will be available here as well, leading to identical variant generation for both branches. This is discussed in more detail in section 6.4 along with other important properties and implications of non-intersecting operations.

Additionally, it should be clear that the same operation may be generated multiple times during the variant generation process, by the same schema. In this situation, only a part of the likelihood and graph will be involved in the operation generation, alongside a single schema. Therefore, operations are memoized/cached upon first generation and subsequent invocations of the same operation are done by a simple look-up based on a combination of schema,likelihood terms and nodes involved (operation signature). A more detailed discussion of this is handled in sections 6.1 and 6.4.

6.2.2 Algorithm Candidates

An algorithm candidate is defined as a variant which has a working sampler for all parameters present in the graph. A leaf variant, is a variant that is at a leaf node of the recursive tree generated by the depth-first search variant generation process. All leaf node algorithm candidate are considered by the variant generation scheme for the generation of samplers.

In particular, for graph G of variant $V(G, L, O, S)$. $G = (N, E)$ where N is the set of all nodes, and E is the set of all edges. Then let P be the set of all parameters of the graph, with $P \subseteq N$. Then, in order for a particular variant to be viable for use for inference, each parameter $p \in P$ would need to have a sampler associated with it, otherwise that parameter would form a missing link in the overall training process. At intermediate stages of the variant generation process, this will generally not be true. Intermediate variants can achieve this property by removing parameter nodes (through collapsing) or by introducing new nodes that allow other nodes to be sampled (through augmentation/simplification). While it is possible for a variant to be an algorithm candidate without reaching a leaf node, it is necessarily the case that any such variant will be represented by some leaf node. The proof of this is as follows:

Let $V(G, L, O, S)$ be a variant with the following conditions, all parameters of V have a sampler available for it. Then what needs to be proven is the fact that there exists a leaf node variant $V'(G, L, O', S')$ with the exact same configuration as V, with the

same graph, set of parameters, graph and likelihood as V. Now consider the lines $o = Op.\text{pop}()$ and VariantGeneration(G, L, Opn, Od) in the algorithm above. These lines essentially "skip" the current operation pointed to by the top of the pending operations container and traverses to a deeper level of the depth first search recursive tree. At a later stage, it is possible that more operations are assigned by schema that finds a match on this variant. However, in such a case, the same process can be repeated and the operations created by such schema can be "skipped" again. Inductively, this means that the current variant can be "maintained" in terms of properties right down to a leaf node. Therefore, for any variant $V(G, L, O, S)$, there exists a variant $V'(G, L, O', S')$ with the exact same configuration as V, with the same graph, set of parameters, graph and likelihood as V present as a leaf node in the recursive tree of the depth first search process. In fact, provided the identity operation I, which performs no change to a variant V, such that $IV(G, L, O, S) = V(G, L, O, S)$ is disallowed by definition (since there is no logical reason for allowing the existence of an identity schema), we can guarantee that there exists a unique variant V' that matches the properties discussed previously. Note that the parameters in a graph are a part of its set of nodes, and therefore cannot be changed without changing the graph itself (and probably the likelihood as well). Therefore, for any algorithm variant present at a non-leaf node in the recursive tree, there exists a unique algorithm variant identical to it taking up a leaf node position. This follows from the fact that the multiple identical non-leaf variants all converge to the same leaf node variant (identical to all of them in all respects other than recursive depth) in the absence of an identity schema.

This directly supports the fact that potential algorithm variants only need to be processed at leaf-node positions, greatly reducing the overall run-time complexity of the overall variant generation process.

6.3 Proof of Variant Coverage

A key contribution of our system is the fact that it can generate multiple variants from the provided model specification. Considering the fact that schema are meant to be reusable with minimum required configuration, the variant generation process needs to be robust enough to handle a variety of schema and still ensure that all variants of interest are generated. In this context, variant coverage is defined as follows:

Variant coverage is the percentage of algorithm candidate leaf variants (as defined previously), that can be generated by the system as compared to the set of algorithm candidates that can be generated. It is important to consider leaf variants as these are the variants that are actually processed, since algorithm generation is only carried out on leaf variants as per the algorithm definition above.

The set of algorithm candidates that can be generated is assumed to be the set of different variants that might be generated by an oracle with unlimited statistical, symbolic and computational capabilities, applying the operations exactly as they are defined by the system, but always applying them in an optimal fashion in order to generate the highest number of algorithm candidates possible. Clearly, the set of operations are limited to those generated by the schema provided alongside the model specification, as this limitation determines the algorithm candidates generated for a particular model specification.

Our claim about variant coverage is given in the theorem.
Theorem: The variant generation process achieves a variant coverage of 100% or total coverage of all variants.

Proof: We have previously proven that for any algorithm candidate, there exists a unique leaf variant algorithm candidate, provided the identity operation/schema is not supported. A generalization of this proof is also possible, where we can show that for any given variant, there exists a unique leaf variant with the same properties. This arises from the fact that we did not use any special property of algorithm candidates in the proof above and thus all statements could be generalized for general nodes and leaf nodes.

Therefore to ensure coverage for any algorithm candidate leaf node, it is sufficient to show there exists an algorithm candidate with the same properties in some node of the recursive depth-first tree for the variant generation process. Coverage for an algorithm candidate leaf node is required as it is only at the leaf level that variants are processed for sampler generation.

Now let us consider how the oracle would perform generation of a particular algorithm candidate. Since the oracle has perfect knowledge of the steps needed to generate the algorithm candidate, it can generate the steps needed to reach it using the provided schema, and as soon as it reaches it, stop. Let us consider such an algorithm variant $V_t(G_t, L_t, O_t, S_t)$. Let the initial variant derived from the model specification be $V_0(G_0, L_0, O_0, S_0)$. Since V_t has been generated from V_0 using operations generated from schema, we can say without loss of generality, $V_t = O_n O_{n-1} ... O_1 V_0$ where the operations $O_1, O_2, ..., O_n$ are the 1st,2nd,...,n-th operation applied, in order and have been derived based on schema $S_1, S_2, ..., S_n$. Note that the numbering indicates the order of the operation performed, and not the order of items stored in any data-structure/container.

Since the oracle has decided on this set of steps, there are two key aspects that need to be covered in this proof.

1. The system can identify the applicability of these schema

2. This sequence of operations can be generated by the system in that order.

The first statement relates to pattern matching.

Now let us consider the 2nd statement. Since the oracle has managed to generate $V_t = O_n O_{n-1} ... O_1 V_0$, it would have also managed to generate $V_{t'} = O_{n-1} ... O_1 V_0$ at some stage. In going from $V_{t'}$ to V_t, the oracle has performed operation O_n. As the oracle is bound to using the schema specified by the system, there exists a schema s such that $s \in S$ and s is applicable on $V_{t'}(G'', L'', O'', S'')$. In fact, $O_n \in s.generateOperation(G'', L'')$, in other words O_n is one of the operations that can be generated by s on the variant $V_{t'}$. Now, let us assume our system can generate $V_{t'}$ from V_0 using some sequence of operations and schema activations. We know that $s \in S$, that the schema that generates O_n is present in the current set of schema and that s is applicable to $V_{t'}$. Then, let O' be the set of pending operations at this stage for $V_{t'}$ in our system. If any of these operations are performed, the variant traverses down a different path as $V_{t''} = O_{n'} O_{n-1} ... O_1 V_0$ where $O_{n'} \in O'$. However, consider the following lines in the algorithm:

if *Opn is not empty* **then**
  //recurse without performing operation
  $o = Op$.pop()
  VariantGeneration(G, L, Opn, Od)
  //perform operation o on G and L
  G', L' = performOperation(o, G, L)
  //o is now done and no longer pending
  Od.push(o)
  G' and L' are new instances
  VariantGeneration(G', L', Opn, Odn)
  //reset current state for maintaining proper DFS
  $Od.pop()$
end

Algorithm 12: part of Variant Generation Search Algorithm

The lines $o = Op$.pop() and VariantGeneration(G, L, Opn, Od) allows us to skip the first of these operations by ignoring the current operation at the top of the pending container and applying to a deeper recursive level while not actually applying the operation in the graph. As this is a recursive call, this process can be repeated indefinitely, effectively "skipping" the entire container of operations.

Now consider the following lines:

```
if there are applicable schema then
    //generate new operatons, recurse
    for all applicable schema s ∈ S do
        //Multiple operations are possible from a single schema
        O = s.generateOperations(G,L)
        for all operations o ∈ O do
            Op.push(o)
        end
        VariantGeneration(G, L, O, S, d + 1, op)
        //Maintain DFS state
        for all operations o ∈ O do
            Op.pop()
        end
    end
end
```

Algorithm 13: part of Variant Generation Search Algorithm

Since the operations in the container were "skipped", the next lines are executed. We have already established that $s \in S$ and that s is applicable to the current variant. Therefore, consider any other schema that may be check before s and are found to be applicable. Let s' be such a schema. Let $O_{s'}$ be the set of operations generated by this schema. Then the queue O is updated by adding all elements of $O_{s'}$ to it, and then VariantGeneration(G, L, Opn, Odn) is called, leading to deeper recursion. Now there would be more pending operations in the pending operations container, but we can simply "skip" these once more, as before. This process can be repeated for all schema found to be applicable before processing of s is conducted. Now, since $s \in S$ and s is applicable, s is guaranteed to be processed, creating operation O_n. O_n is added to the list of pending operations and a recursive call is made. This time, we consider the following lines:

//perform operation o on G and L
G', L' = performOperation(o, G, L)
//o is now done and no longer pending
Od.push(o)
G' and L' are new instances
VariantGeneration(G', L', Opn, Odn)
//reset current state for maintaining proper DFS
$Od.pop()$

Since O_n is applied on the variant, $V_t = O_n O_{n-1} ... O_1 V_0$ is generated, and since V_t is an algorithm candidate, by our proof that any algorithm candidate has a unique algorithm candidate leaf variant, provided the identity operation is not supported. We can impose

the restriction of the identity operation to ensure a unique leaf node that contains the information required for processing this variant.

We have proven that if $V_{t'} = O_{n-1}...O_1V_0$ is present as a variant in our system, then $V_t = O_nO_{n-1}...O_1V_0$ can be generated. In doing so, we have not assumed any particular properties of the graph, likelihood or operations. We have based our argument on the fact that the schema corresponding to this "bridging" operation is present in the set of schema, which is required to be true because otherwise such a variant cannot be generated. Since this reasoning does not depend on anything but the set of schema themselves, this logic can be further generalized to show that $V_{t_k} = O_k...O_1V_0$ can be generated in our system provided $V_{t_{k-1}} = O_{k-1}...O_1V_0$ can be generated. We also know by definition, that $V_0(G_0, L_0, O_0, S_0)$ can be generated by the system. Thus, we can conclude using the principle of mathematical induction that $V_t = O_nO_{n-1}...O_1V_0$ can be generated by our system. This concludes the proof of the fact that the sequence of operations selected by the oracle can be generated by the system in that order.

6.4 Bounded Runtime

The implication of section 6.3 is that any possible variant supported by the schema can be generated by the variant search process. This requires some further explanation, especially with regards to the recursive steps taken by the algorithm.

Consider the following scenario: A schema is activated and generates an operation o. A recursive call to o is triggered and o is then "skipped" along with other operations and execution proceeds until operation generation (by schema) once more. If at this stage, the same schema is triggered (and we know that it can be triggered, since its triggering is what led us to this scenario), then an infinite loop is created. This is avoided in practice due to two main reasons:

1. Schema are set up so that all possible matches are generated and the corresponding operations are queued. This means that there is no logical reason for the same schema to be activated more than once on any given call of the recursive function. Note that even if an operation is skipped, it is removed from the queue by calling Op.pop().

2. Thereby, reaching the end of the iteration over schema can be considered as the endpoint for the variant generation in that level of the recursive call. When this stage is reached, it is possible to check for necessary conditions to decide whether the current variant is a potential algorithm candidate leaf variant or not.

In particular; if any operations or schema have been activated in the current recursive call (easily checked by setting Boolean flags), then it is not a leaf node of the recursive call tree and thereby it cannot be an algorithm candidate leaf variant. So the current recursive call can be ended and control of execution can be passed to the previous recursive level.

It is important to note that no schema is run while there are operations pending (as previously discussed in section 6.3. This is not the same as saying all operations are executed before schema are generated (some or all of the operations may have been skipped).

Let us define a few properties of variants. The inverse O_m of operation O_n is such that $O_m O_n V = IV = V$. While the system implicitly performs the inversion process (because of the recursive calls), thereby ensuring the impossibility of this happening due to reversing operations on the part of the system, it is still possible for users to create schema that lead to the generation of such operations. The issue with such a situation is the fact that it may create infinite loops or repeats of the same operations being generated over and over in the recursive call tree.

However, it is insufficient to merely handle simple situations like this. Let us define non-intersecting operations as follows: operations O_n and O_m are called non-intersecting if $O_n O_m V = O_m O_n V$. This would generally indicate that they operate on separate parts of the graph with no overlapping modifications of nodes or likelihood terms. Now consider a sequence of operations $O_m, O_n, O_m^{-1}, O_n^{-1}$ with O_n and O_m non-intersecting. The overall resultant would be: $O_n^{-1} O_m^{-1} O_n O_m = O_n^{-1} O_m^{-1} O_m O_n = O_n^{-1} I O_m = I$

The mechanism for handling this is straightforward: Note that such a sequence of operations would start at a particular variant $V(G, L, O, S)$ and cause it to appear again at a later stage in the recursion tree. The easiest way to check for this is by maintaining pointers from each unique likelihood to all its occurrences in the recursion tree (as a hash for example). Whenever a new recursive call is made (by an operation), the likelihood for before that operation is saved into the hash, and when recursion returns to that point, the likelihood is removed. This means, if a clash ever occurs, it is in the same branch of the recursive call tree, indicating the possibility of an infinite loop. At this stage, that branch is pruned from the recursive call tree while also warning the user about the list of operations used in generating the infinite loop, for debugging purposes.

Define the applicability of an operation O on a variant V as follows: Applicable(O,V) = true if O is applicable on V (ie O originated from a schema that had its applicability constraint matched against V) and false if it is not applicable on V. Then a crucial rule of the system must be that Applicable(O,OV) = false, to prevent another class of infinite loops. In other words, operations should be self-invalidating. If an operation that is not self invalidating is introduced via a schema, it is easily found by the repeated occurrence

of that same operation over and over again in a particular branch, and can be used to identify branch pruning conditions and debug information for users. In particular, it is possible to identify operations uniquely using their signature. Due to the self invalidating restriction, the same operation shouldn't occur twice in the same chain of operations, and definitely not 5 or 6 times, so this can be used as a criteria for identifying such operations and the offending schema corresponding to them.

A key outcome of these properties is that they also allow a way for handling pruning strategies on the recursive tree. For example, a variant generated entirely by pairwise non-intersecting operations will be the same no matter the order of operations applied, thereby allowing pruning when such a variant is found.

Chapter 7

Experimental Results

This chapter provides details on some of the results generated by the system.

The first section provides a detailed evaluation of two models for which the inference algorithms are not easily generated by current probabilistic programming languages due to either symbolic complexity, model complexity, or both.

Next, a discussion of the results of variant generation is provided, using an example to indicate the differences in the model structure and the final generated code based off a single initial model specification.

7.1 Comprehensive Evaluation of Generated Code

In this section we discuss the evaluation process used to test our system. Testing was carried out on MetaLDA [65] and MIGA [69] and we compared the original Matlab implementations of the models downloaded from the authors' Github[1] with our generated models. These two models selected primarily for the following reasons:

Use of collapsing and augmentation: Both the models made considerable use of these operations.

Symbolic complexity: The analysis process for the models was considerably complicated, requiring many simplification steps.

The primary evaluation criteria used were result similarity and code similarity. Testing was performed on the Web Snippets data-set, a widely used text dataset [14, 36, 69], which contains 12,237 web search snippets and each snippet belongs to one of 8 categories. The vocabulary contains 10,052 tokens, and there are 15 words in one snippet

[1] https://github.com/ethanhezhao

on average. Result similarity was tested using likelihood evaluation, topic-term probability comparison and visualization. Testing was performed with 2000 iterations and 100 topics. All parameter and prior settings were mimicked from the original papers.

Likelihood Evaluation: In this set of experiments, we studied how closely the models generated by our system can "reproduce" the training log-likelihood of the original models. We plotted the training log-likelihood of each model as a function of training iterations with a gap of 10, as shown in Figure 7.1. Moreover, we also computed simple statistics, i.e., mean and standard deviation of each curve in Table 7.1.

Both the plots and the statistics show that the generated models behave very similarly to their original models.

The log likelihood of each model was plotted against the iteration number, and the resulting graph was checked for convergence and similarity. Table 7.1 and Figure 7.1 illustrate these results. Table 7.1 contains the results corresponding to the log likelihoods of each model from iteration 10 to 2000 with an interval of 10. Convergence is implied by figure 7.1 and the final values are indicated under convergence of models in table 7.1, while the means and standard deviations of the curves are indicated in the same table as well.

TABLE 7.1: Results - Log Likelihood

Convergence of Models		
	MetaLDA	**MIGA**
Original	-1.0642×10^6	-1.1500×10^6
Generated	-1.0636×10^6	-1.1290×10^6
Stats of convergence curve - MetaLDA		
	Mean	**Std. Dev.**
Original	-1.0745×10^6	2.6819×10^4
Generated	-1.0709×10^6	2.1258×10^4
Stats of convergence curve - MIGA		
	Mean	**Std. Dev.**
Original	-1.1828×10^6	6.1528×10^4
Generated	-1.1632×10^6	5.9313×10^4

Topic-term probability comparison: was used to provide a comparison of the topics assigned by the generated code vs the original model. The idea is that there are strong topics in the data-set that should be picked up by both types of code. These will be assigned to random indices in the topic-term probability matrix $\phi_{k,v}$. By running the Hungarian algorithm on a distance metric between the rows of two $\phi_{k,v}$ matrices, it should be possible to match these similar topics together. By picking the lowest distanced probability vectors first, the most similar topics represented via proxy by the rows of $\phi_{k,v}$ are matched across the models. A record of the distances of each of these 100 matching

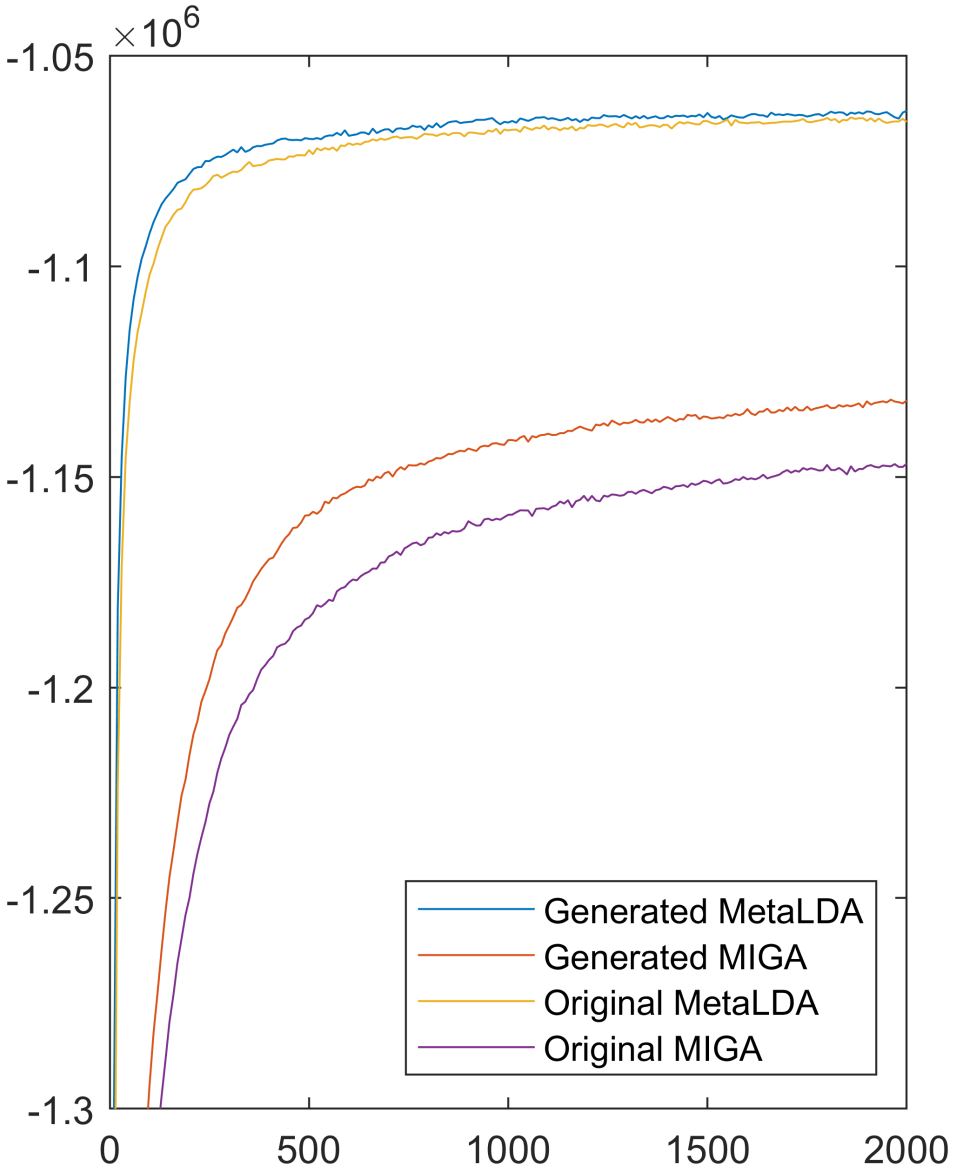


FIGURE 7.1: log likelihood vs iteration - higher is better

steps (one for each topic) is kept and plotted. The comparison is conducted for 5 seperate runs of each algorithm. Comparisons are intra-model (Original vs Original, Generated vs Generated giving $\binom{5}{2} = 10$ plots each) and inter-model (Original vs Generated giving $5 \times 5 = 25$ plots each). Results are shown in figure 7.2. The distance metric used to compare the two topics' word probabilities is Hellinger distance. This shows conclusively that the two algorithms are generating similar kinds of ϕ matrices.

Visualization: incorporated the visualization techniques used by the original authors in order to compare the results of the generated code against those reported by the authors. An excerpt from the result for MetaLDA is given in Table 7.2. This demonstrates that the generated code is resulting in good output topics. Here $\lambda_{l,k}$ indicates the weight

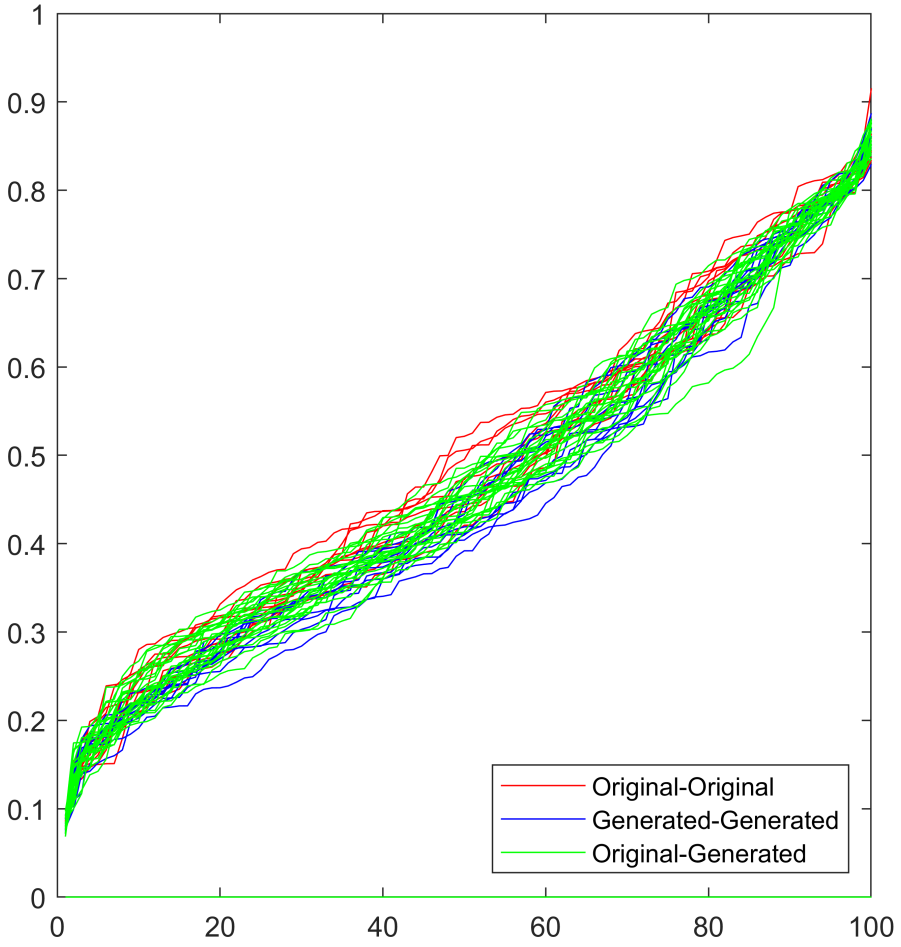


FIGURE 7.2: MIGA O-O,O-G,G-G Hellinger distance (y) vs topic iteration (x)

assigned for topic k with respect to label l.

Code similarity: refers to the structural similarity of the code generated by our system, compared to the original authors' code/algorithm presented in the original papers. Code similarity provides a heuristic means of checking model equivalence without needing to run the model.

In each case, the code used for testing was from the model variant which most closely resembled the model presented in the authors' original paper (all possible collapsing operations performed, and all augmentations performed as well). In general, a single variant will be similar to the existing model for comparison purposes. This is because the original model will always use a sampling strategy similar to one of the variants (assuming the techniques used by the authors have been represented by schema in the system). This then becomes a baseline for comparison against and can help to criticize the other variants.

In checking for code similarity, we found satisfactory similarity, especially when comparing with the published algorithms for the models. In the case of MetaLDA, the published

TABLE 7.2: Top 3 topics per label and words per topic

Label	$\lambda_{l,k}$	Top 3 words (ordered)
Business	10.10	marketing sales advertising
	5.65	bank currency cruise
	4.88	estate buying selling
Computers	11.96	intel apple chip
	10.41	computer architecture computers
	8.74	linux system operating
Culture, arts and entert--ainment	10.45	oscar awards academy
	8.82	music lyrics beatles
	6.48	movie movies imdb
Education and Science	12.80	physics quantum mechanics
	10.65	science computer psychology
	10.14	theorem newton proof
Engineering	16.26	electrical motor electric
	8.45	engine diesel fuel
	5.67	engineering invention science
Health	9.58	cancer physical therapy
	8.02	disease diagnosis health
	6.70	health diet healthy
Politics and Society	11.83	republic president freedom
	9.39	nuclear weapons bombs
	7.54	system government presidential
Sports	10.86	football league soccer
	8.04	goalkeeper maradona diego
	7.86	sports news espn

code was somewhat dissimilar to the generated code mainly due to the fact that MetaLDA's authors had used the MALLET library which uses its own complex structures incorporating sophisticated "fast" LDA operations. MIGA's authors had used MATLAB code, which led to a high degree of similarity, although there were noticeable differences due to the efforts taken in order to vectorize the MATLAB code. Unfortunately, we were unable to find a reasonable metric to quantify the similarity between the original code and our generated code at the syntax level. Using plagiarism detection algorithms seemed like a promising option, but did not yield any conclusive results.

This is illustrated in figures 7.3, 7.4 and 7.5. Here, snippets of the sampling process from the initial paper are overlaid on generated code for the same model for the purpose of comparison.

Speed: while direct speed comparisons between the implementations are not meaningful, due to the code being in 2 different languages and the original code being manually optimized in both cases to improve run-time, they are nonetheless listed in Table 7.3. It is instructive to compare the complexity of the different systems. Our system achieves the same computational complexity as the original algorithms, albeit with higher constants

```
for (int d = 0; d < D; d++) {
  int I = w[d].length;
  for (int i = 0; i < I; i++) {
    c0_1[d]--;
    c0[d][z[d][i]]--;
    c1_1[z[d][i]]--;
    c1[z[d][i]][w[d][i]]--;
    double[] p = new double[K];
    for (int k = 0; k < K; k++) {
      p[k] = (Math.pow(c0_1[d] + a_1[d], -1)) * (Math.pow(c1_1[k] + b_1[k], -1)) * (c0[d][k] + a[d][k]) * (c1[k][w[d][i]] + b[k][w[d][i]]);
    }
    for (int k = 1; k < K; k++) {
      p[k] += p[k - 1];
    }
    int k;
    double val = Math.random() * p[K - 1];
    for (k = 0; k < K; k++) {
      if (p[k] > val) {
        break;
      }
    }
    z[d][i] = k;
    c0_1[d]++;
    c0[d][z[d][i]]++;
    c1_1[z[d][i]]++;
    c1[z[d][i]][w[d][i]]++;
  }
}
```

C. Sampling topic $z_{d,i}$:

Given $\boldsymbol{\alpha_d}$ and $\boldsymbol{\beta_k}$, the collapsed Gibbs sampling of a new topic for a word $w_{d,i} = v$ in MetaLDA is:

$$\Pr(z_{d,i} = k) \propto (\alpha_{d,k} + m_{d,k}) \frac{\beta_{k,v} + n_{k,v}}{\beta_{k,\cdot} + n_{k,\cdot}} \qquad (14)$$

FIGURE 7.3: Code comparison

```
//number of tables in CRP
int q0[][] = new int[D][K];
//beta augmentation variable
double q1[] = new double[D];
for (int d = 0; d < D; d++) {
  for (int k = 0; k < K; k++) {
    if (c0[d][k] > 0) {
      q0[d][k] = 1;
      for (int iter = 1; iter < c0[d][k]; iter++) {
        q0[d][k] += helper.BernoulliSample(a[d][k] / (a[d][k] + iter));
      }
    } else {
      q0[d][k] = 0;
    }
  }
}
for (int d = 0; d < D; d++) {
  q1[d] = helper.BetaSample(a_1[d], (c0_1[d]));
}
```

Process (CRP) [29], $t_{d,k}$ can be sampled by a CRP with $\alpha_{d,k}$ as the concentration and $m_{d,k}$ as the number of customers:

$$t_{d,k} = \sum_{i=1}^{m_{d,k}} \text{Bern}\left(\frac{\alpha_{d,k}}{\alpha_{d,k} + i}\right) \qquad (5)$$

where for each document d, $q_d \sim \text{Beta}(\alpha_{d,\cdot}, m_{d,\cdot})$. Given

FIGURE 7.4: Code comparison, Augmentation variables are assigned names by the system and not externally

due to being not having been scrutinized for manual optimizations such as vectorization as the original code has been.

TABLE 7.3: Speed comparison - runtime in hours

	MetaLDA	**MIGA**
Original	0.41	0.47
Generated	0.94	2.95

```
//Sampler for lambda[l][k]
int[][] sum1 = new int[L][K];
for (int d = 0; d < D; d++) {
  for (int k = 0; k < K; k++) {
    for (int iter : (ArrayList<Integer>) f[d]) {
      sum1[iter][k] += q0[d][k];
    }
  }
}
double[][] sum2 = new double[L][K];
for (int l = 0; l < L; l++) {
  for (int k = 0; k < K; k++) {
    for (int d : inversef[l]) {
      sum2[l][k] += (a[d][k] / lambda[l][k]) * Math.log(q1[d]);
      a[d][k] /= lambda[l][k];
    }
    lambda[l][k] = helper.GammaSample((mu + sum1[l][k]), 1) / (1 / mu - sum2[l][k]);
    for (int d : inversef[l]) {
      a[d][k] *= lambda[l][k];
    }
  }
}
```

$$\mu' = \mu_0 + \sum_{d=1:f_{d,l}=1}^{D} t_{d,k} \tag{8}$$

$$\mu'' = 1/\mu_0 - \sum_{d=1:f_{d,l}=1}^{D} \frac{\alpha_{d,k}}{\lambda_{l,k}} \log q_d \tag{9}$$

$$\lambda_{l,k} \sim \mathrm{Ga}(\mu', 1/\mu'') \tag{7}$$

FIGURE 7.5: Code comparison

7.2 Code for Variants

This section provides an overview of the code generated for different variants of the LDA model (refer to section 1.2.2 for derivation). For each parameter being sampled, the initialization code and MCMC code have been provided with comments delimiting the relevant sections for ease of inspection. The dotted lines in images indicated parameters and edges that have been collapsed out, and are thus inactive in the model.

No Collapsing performed

In this variant, all parameters are present, and need to be sampled explicitly. Alpha and Beta are priors for Theta and Phi respectively, and are constant vectors. Figure 7.6 indicates this variant.

```
//==== NEW SAMPLER FOR theta ====
//Initialization
for (int d = 0; d < D; d++){
    for (int k = 0; k < K; k++){
        theta[d][k]=1.0/K;
    }
}
//For each iteration of the Markov Chain run the following:
```

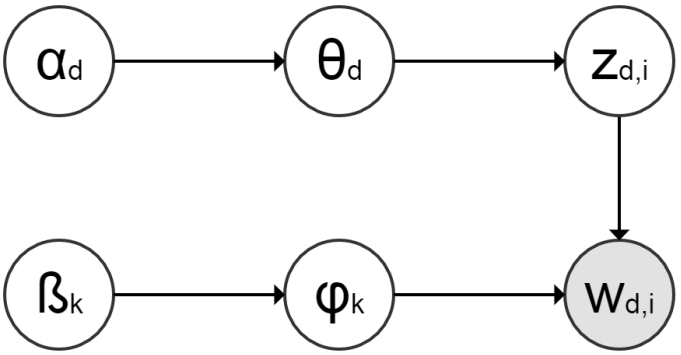


FIGURE 7.6: An LDA variant with no parameters collapsed

```
for (int d = 0; d < D; d++){
    //Sample from full conditional
    theta[d]= DirichletSample(c0[d] + a);
}
//==== NEW SAMPLER FOR phi ====
//Initialization
for (int k = 0; k < K; k++){
    for (int v = 0; v < V; v++){
        phi[k][v]=1.0/V;
    }
}
//For each iteration of the Markov Chain run the following:
for (int k = 0; k < K; k++){
    phi[k]= DirichletSample(c1[k] + b);
}

//==== NEW SAMPLER FOR z ====
//Initialization
for (int d = 0; d < D; d++){
    for (int i = 0; i < I; i++){
        z[d][i]=(int)Math.random()*K;
        c0_1[d]++;
        c0[d][z[d][i]]++;
        c1_1[z[d][i]]++;
        c1[z[d][i]][w[d][i]]++;
    }
}
```

```
//For each iteration of the Markov Chain run the following:
for (int d = 0; d < D; d++){
    for (int i = 0; i < I; i++){
        c0_1[d]--;
        c0[d][z[d][i]]--;
        c1_1[z[d][i]]--;
        c1[z[d][i]][w[d][i]]--;
        //Sample from full conditional
        double[] p = new double[K];
        for (int k = 0; k < K; k++){
            p[k]=(Math.pow(phi[k][w[d][i]],1))*(Math.pow(theta[d][k],1));
        }
        //cumulate values
        for (int k = 1; k < K; k++){
            p[k]+=p[k-1];
        }
        int k;
        double val = Math.random()*p[K-1];
        for (k = 0; k < K; k++){
            if (p[k]>val)break;
        }
        z[d][i]=k;
        c0_1[d]++;
        c0[d][z[d][i]]++;
        c1_1[z[d][i]]++;
        c1[z[d][i]][w[d][i]]++;
    }
}
```

Theta collapsed

In this variant Theta is collapsed out, while Phi remains. This means that Theta does not need to be sampled in the Markov Chain. Figure 7.7 indicates this variant.

```
//==== NEW SAMPLER FOR phi ====
//Initialization
for (int k = 0; k < K; k++){
    for (int v = 0; v < V; v++){
```

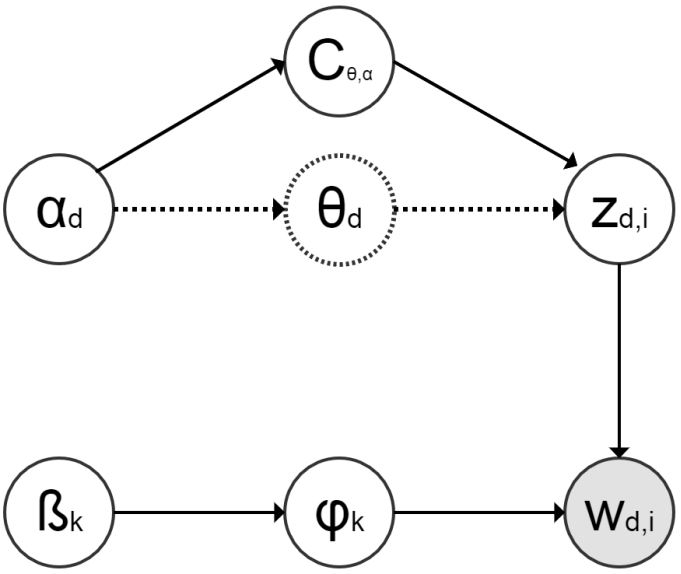


FIGURE 7.7: An LDA variant with θ collapsed

```
            phi[k][v]=1.0/V;
        }
    }
    //For each iteration of the Markov Chain run the following:
    for (int k = 0; k < K; k++){
        //Sample from full conditional
        phi[k]= DirichletSample(c1[k] + b);
    }
//==== NEW SAMPLER FOR z ====
    //Initialization
    for (int d = 0; d < D; d++){
        for (int i = 0; i < I; i++){
            z[d][i]=(int)Math.random()*K;
            c0_1[d]++;
            c0[d][z[d][i]]++;
            c1_1[z[d][i]]++;
            c1[z[d][i]][w[d][i]]++;
        }
    }
    //For each iteration of the Markov Chain run the following:
    for (int d = 0; d < D; d++){
        for (int i = 0; i < I; i++){
            c0_1[d]--;
            c0[d][z[d][i]]--;
```

```
            c1_1[z[d][i]]--;
            c1[z[d][i]][w[d][i]]--;
            //Sample from full conditional
            double[] p = new double[K];
            for (int k = 0; k < K; k++){
                p[k]=(c0[d][k]+a[d][k])*(Math.pow(phi[k][w[d][i]],1));
            }
            //cumulate values
            for (int k = 1; k < K; k++){
                p[k]+=p[k-1];
            }
            int k;
            double val = Math.random()*p[K-1];
            for (k = 0; k < K; k++){
                if (p[k]>val)break;
            }
            z[d][i]=k;
            c0_1[d]++;
            c0[d][z[d][i]]++;
            c1_1[z[d][i]]++;
            c1[z[d][i]][w[d][i]]++;
        }
    }
```

Phi collapsed

Similar to the previous variant, now Phi is collapsed. Figure 7.8 indicates this variant.

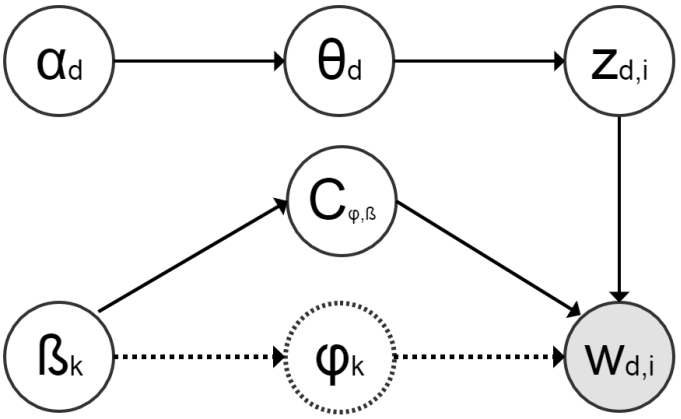


FIGURE 7.8: An LDA variant with φ collapsed

```
//==== NEW SAMPLER FOR z ====
//Initialization
for (int d = 0; d < D; d++){
    for (int i = 0; i < I; i++){
        z[d][i]=(int)Math.random()*K;
        c0_1[d]++;
        c0[d][z[d][i]]++;
        c1_1[z[d][i]]++;
        c1[z[d][i]][w[d][i]]++;
    }
}
//For each iteration of the Markov Chain run the following:
for (int d = 0; d < D; d++){
    for (int i = 0; i < I; i++){
        c0_1[d]--;
        c0[d][z[d][i]]--;
        c1_1[z[d][i]]--;
        c1[z[d][i]][w[d][i]]--;
        //Sample from full conditional
        double[] p = new double[K];
        for (int k = 0; k < K; k++){
            p[k]=(c1[k][w[d][i]]+b[k][w[d][i]])*(Math.pow(theta[d][k],1));
        }
        //cumulate values
        for (int k = 1; k < K; k++){
            p[k]+=p[k-1];
        }
        int k;
        double val = Math.random()*p[K-1];
        for (k = 0; k < K; k++){
            if (p[k]>val)break;
        }
        z[d][i]=k;
        c0_1[d]++;
        c0[d][z[d][i]]++;
        c1_1[z[d][i]]++;
        c1[z[d][i]][w[d][i]]++;
    }
}
//==== NEW SAMPLER FOR theta ====
```

```
//Initialization
for (int d = 0; d < D; d++){
    for (int k = 0; k < K; k++){
        theta[d][k]=1.0/K;
    }
}
//For each iteration of the Markov Chain run the following:
for (int d = 0; d < D; d++){
    //Sample from full conditional
    theta[d]= DirichletSample(c0[d] + a);
}
```

Both Phi and Theta collapsed

In this variant, both Theta and Phi have been collapsed, leaving only Z to be sampled. Figure 7.9 indicates this variant.

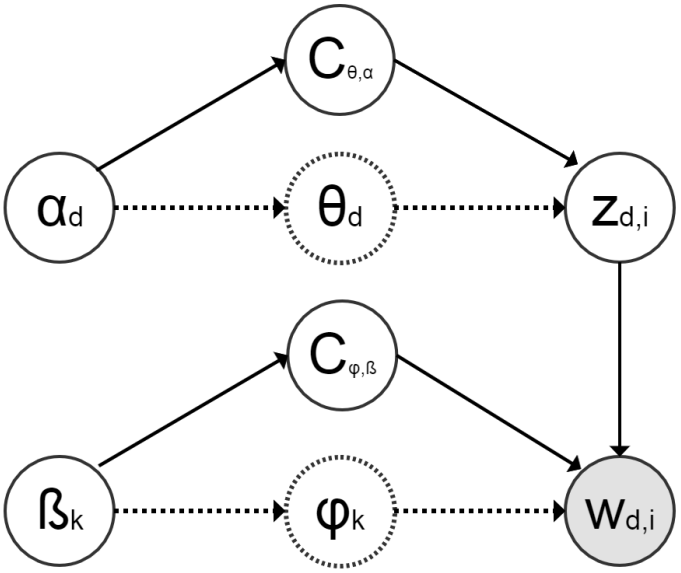


FIGURE 7.9: An LDA variant with θ and φ collapsed

```
//==== NEW SAMPLER FOR z ====
//Initialization
for (int d = 0; d < D; d++){
    for (int i = 0; i < I; i++){
        z[d][i]=(int)Math.random()*K;
        c0_1[d]++;
```

```
            c0[d][z[d][i]]++;
            c1_1[z[d][i]]++;
            c1[z[d][i]][w[d][i]]++;
        }
    }
    //For each iteration of the Markov Chain run the following:
    for (int d = 0; d < D; d++){
        for (int i = 0; i < I; i++){
            c0_1[d]--;
            c0[d][z[d][i]]--;
            c1_1[z[d][i]]--;
            c1[z[d][i]][w[d][i]]--;
            //Sample from full conditional
            double[] p = new double[K];
            for (int k = 0; k < K; k++){
                p[k]=1*(c0[d][k]+a[d][k])*(c1[k][w[d][i]]+b[k][w[d][i]]);
            }
            //cumulate values
            for (int k = 1; k < K; k++){
                p[k]+=p[k-1];
            }
            int k;
            double val = Math.random()*p[K-1];
            for (k = 0; k < K; k++){
                if (p[k]>val)break;
            }
            z[d][i]=k;
            c0_1[d]++;
            c0[d][z[d][i]]++;
            c1_1[z[d][i]]++;
            c1[z[d][i]][w[d][i]]++;
        }
    }
```

Extending LDA with a document cluster

Consider the model specification below. This is a simplification of MIGA [69], but the handling of meta-information has been removed. Note the generated code for full MIGA is given in Appendix A.2.

```
  for (k in 1:K) {
```

```
  phi[k,1:V] ~ ddirich(b);
}
for (t in 1:T){
  theta[t,1:K] ~ ddirich(a);
}
mu[1:T] ~ ddirich(g);
for (m in 1:M) {
  y[m] ~ dcat(mu);
  for (n in 1:N) {
    z[m,n] ~ dcat(theta[y[m]]);
    w[m,n] ~ dcat(phi[z[m,n]]);
  }
}
```

This is an extension of LDA with a cluster y drawn for each document m. When the cluster for a particular document is sampled, it is necessary to consider the allocation of topics within that topic, which leads to a complicated update. This is broken down into several parts and computed as individual products, due to the higher readability provided by doing so.

```
//==== NEW SAMPLER FOR y ====
//==== FORMAT M ====
//Initialization
for (int m = 0; m < M; m++){
    y[m]=Math.Random()*T;
    c0[y[m]]++;
    c1_1[y[m]]++;
}
//For each iteration of the Markov Chain run the following:
for (int m = 0; m < M; m++){
    c0[y[m]]--;
    c1_1[y[m]]--;
    //Sample from full conditional
    double[] p = new double[T];
    for (int t = 0; t < T; t++){
        int counter = 0;
        for (int n = 0; n < N[m]; n++){
            counter++;
        }
        float prod0 = 1.0;
```

```
            for (int k = 0; k < K; k++){
                prod0 = prod0 *
                (Math.pow(RisingFactorial(a_1+c1_1[t] , counter ),-1));
            }
            int counter_T[] = new int[K];
            for (int n = 0; n < N[m]; n++){
                counter_T[z[m][n]]++;
            }
            float prod1 = 1.0;
            for (int k = 0; k < K; k++){
                prod1 = prod1 * (RisingFactorial(c1[t][k]+a[k] , counter_T[k] ));
            }
            p[t]=prod0*prod1*(Math.pow(mu[t],1));
        }
        //cumulate values
        for (int t = 1; t < T; t++){
            p[t]+=p[t-1];
        }
        int t;
        double val = Math.random()*p[T-1];
        for (t = 0; t < T; t++){
            if (p[t]>val)break;
        }
        y[m]=t;
        c0[y[m]]++;
        c1_1[y[m]]++;
    }
```

Chapter 8

Conclusion

This thesis has mainly focused on automatic code generation for statistical models that rely on augmentation and collapsing. A schema based system for performing automated collapsing and augmentation has been presented. Additionally, a symbolic system that is able to perform necessary symbolic manipulations to perform key steps such as simplification and pattern matching has also been presented in this thesis.

8.1 Summary of Thesis Contents

The main content of the thesis can be summarized as follows:

- As the focus of the thesis is on probabilstic programming languages and probabilistic compilers, Chapter 1 provides an introduction to probabilistic programming languages and provides several examples of automatic and manual inference algorithm/model training algorithm generation using current systems in this space. It also provides an overview of the research outcomes that this PhD set out to achieve.
- Chapter 2 provides an overview of Bayesian analysis including key concepts such as conjugacy, statistical operations such as augmentation and collapsing, and sampling schemes such as Gibbs sampling which form the backbone of the designed system.
- Chapter 3 provides a comprehensive review of the related works in the areas of probabilistic programming languages and symbolic processing. Particular attention is paid to systems which motivated the design of the system portrayed in this thesis.

- Chapter 4 provides a high-level overview of the entire system, with a particular focus on the design principles and how these have been accounted for in terms of the system architecture. Additionally, an overarching example covering the behaviour of the system is also provided.
- Chapter 5 focuses on providing a more granular discussion of the overall system, with a particular focus on implementation details. Schema-based operation automation and symbolic support systems are discussed at length.
- Chapter 6 provides details on variant generation, one of the key contributions of this research. This chapter delves into the inner workings of variant generation and schema and how their interaction provides support for automated statistical and symbolic operations. This chapter also contains proofs regarding the overall variant coverage provided by the system.
- Chapter 7 provides details regarding some of the experiments run on the system. Details regarding a comprehensive evaluation using models which require considerable simplification and statistical operations has been provided. Additionally, details regarding code produced as a result of the variant generation process is provided and discussed.

8.2 Summary of Major Contributions

The major contributions of this thesis are as follows:

- A system for **automatic support of collapsing and augmentation** in statistical models and supporting code generation into multiple languages.
- A **symbolic system** which provides support for the above, including support for simplification and pattern matching steps required for performing statistical operations.
- A **schema based system** for managing statistical operations in a modular fashion, promoting the reuse of components in a plug and play fashion.
- The caching and differential operations on sufficient statististic allow the generation of remarkably efficient algorithms reflecting some of the optimised code generated by programmers. This is seen mostly simply in the generated code for LDA shown after Algorithm alg-lda-intro.

It is not claimed, however, that the system developed is a complete or appropriate probabilistic programming language. This is because most of the development work for

this work has been directed towards the functionality and rigour of the symbolic system and the schema system. The probabilistic language functionality was built up in order to experiment and showcase the focus of the research work. In this regard, thesis shows how some existing languages could be extended with these capabilities.

8.2.1 Automatic support of collapsing and augmentation

To the best of our knowledge, this is the first instance of a statistical system capable of automatically supporting augmentation and collapsing. Additionally, the system provides an architecture for code generation into multiple programming languages. This is achieved by the generation of abstract code structures which can be later specialized based on a language specification. Therefore, in order to generate code in a new language, all that needs to be done is to create a language specification in that programming language. Interestingly, this opens up possibilities for the system to be integrated into probabilistic programming languages as a helper tool, where the model specification is first entered into our system, to generate code in a supported language (for example: Julia for Gen), and then the resulting code is used in Gen for the purposes of inference.

An interesting feature of the system is that the algorithms are targeted to be as general as possible. Most pattern matching occurs on string likelihoods, and as such, this can be incorporated into existing probabilistic languages by first converting their likelihood into a standard expression format (as discussed in section 5.4.1). Then, once pattern matching and any other operations are done, all that is required is to interface with the probabilistic programming language to modify the model structure with the changes suggested by our system. As seen in section 6.1, these would not require a lot of work on the part of the probabilistic programming language, as the operations are fairly simple (for example adding a new node, removing a node etc).

8.2.2 Symbolic System

The symbolic system enables statistical operations by providing support by way of symbolic processing and pattern matching. While existing pattern matching algorithms, particularly in the string domain, perform well on general queries, they are not optimized for working on likelihoods. In particular, Bayesian Likelihoods offer a nice format in the string representation that lends itself to further optimization in the area of string-based pattern matching. With this in mind, we have presented a string-based pattern matching algorithm that accounts for associativity and commutativity and is optimized for operating on Bayesian likelihood functions.

We have described this algorithm along with a proof in section 5.3.4.4, along with a detailed discussion of how it is optimized for handling likelihoods.

8.2.3 Schema System

At its core, the schema system is a way to encapsulate considerable low-level implementation details from end-users. This allows relatively newer users to still have a lot of flexibility in what they can try with the system, merely by merit of adding or removing the set of schema that operate on their model specification. Additionally, schema enforce modular reuse, by ensuring that concepts relevant to a particular statistical operation is contained within the schema template. Schema are set up so that the different schema belonging to a particular type (for example, collapsing schema) are largely similar to each other. Refer Appendix B for examples. This makes the process of defining new schema relatively straightforward. In terms of positioning this work with regards to existing literature, it is an almost hybrid approach between programmatic inference (by Gen, for example) and the standard black-box approach (by Stan, for example) with regards to the flexibility of modifying the inference process. The attractiveness of the former is in its ability to provide a large degree of freedom to the end user, at the cost of requiring to learn how to code in that system. The latter, on the other hand, offers a much simpler interface to end users to create a model specification and then the system does most of the work. Our system is in between: plug and play schema allow a user to simply run their model specification to see what happens, advanced users can add their own schema and exploit the flexibility of the system in this way.

8.3 Concluding Remarks

Current machine learning researchers routinely use collapsing and augmentation in their development of algorithms because it can be necessary to generate an effective sampler. This thesis has presented a system that allows collapsing and augmentation to be applied to exponential family probability models (represented as DAGS) in order to generate different variants of the model. The current system automatically generates Java code, but due to the templates and configuration, Javascript or Python could equally well be generated. Our experimental work indicated we could faithfully duplicate sophisticated Gibbs samplers including models requiring considerable simplifications such as MetaLDA and MIGA.

This thesis presents a general architecture which can generate mutliple variants from a given model specification. Combined with simple code generation, and an evaluation of alternatives, the design philosophy is to provide the user with a sandbox environment where the user has the possibility to modify either initial model or final generated code in order to eventuate changes in the model training process. In addition to allowing the user to generate code in the language they are familiar with, code generation has the added benefit of providing access to desirable language features. MATLAB for example,

has in-built vectorization which can significantly speed up computation while Javascript can be embedded into a webpage with ease for visualization.

The collapsing and augmentation scheme and the code generation scheme detailed here are both general and can be integrated into existing probabilistic languages, as they are both independent of our probabilistic programming language and sampling scheme. The former would require symbolic support, which most probabilistic languages should already have built-in, while the latter would require code level access to all modules where sampling takes place (which is part of the reason we have not relied on an external library to do anything except the most basic sampling). The alternative to this is to embed any library calls within the generated code - a hybrid code generation approach which would still make the process more interactive for the user.

An interesting side-effect of generating variants is the possibility to do automated variant testing. Given data in the format the model expects, the system can automatically generate variants, and then run the variants on the given data to identify issues such as speed, accuracy, effective samples per second and other metrics about the variants. Simply on an empirical level, combining all the possibilities provided by variants, parallelization, vectorization and other language features together with automated testing for model selection provides considerable testing coverage for any single model.

Appendix A

Generated Code for Models

A.1 MetaLDA

```
package mu;

import java.util.ArrayList;
import java.util.Arrays;

public class TestCode {

    public void test(ArrayList[] f, ArrayList[] g, int[][] w,
    Tester tester, TestHelper helper int L, int T) {
        int samplingInvterval = 1;
        int K = 20;
        int V = g.length;
        int D = f.length;
        double mu = 1.0;
        double nu = 1.0;
        int burnin = 10;

        //inversef[l][d] == 1 iff f[d][k] == 1 (inverse index of f)
        ArrayList<Integer>[] inversef = helper.invert(f, L);
        ArrayList<Integer>[] inverseg = helper.invert(g, T);

        //Stat c1 = count of times k and v appear together
        int c1[][] = new int[K][V];
        //Stat c1_1 = count of times k appears
        int c1_1[] = new int[K];
```

```
//Stat c0 = count of times d and k appear together
int c0[][] = new int[D][K];
//Stat c0_1 = count of times d appears
int c0_1[] = new int[D];

int z[][] = new int[D][];

double theta[][] = new double[D][K];

double phi[][] = new double[K][V];

double lambda[][] = new double[L][K];

//number of tables in CRP
int q0[][] = new int[D][K];

//beta augmentation variable
double q1[] = new double[D];

double delta[][] = new double[T][K];

//number of tables in CRP
int q3[][] = new int[K][V];

//beta augmentation variable
double q4[] = new double[K];

//alpha argument in dirichlet prior of theta
double a[][] = new double[D][K];

//beta argument in dirichlet prior of theta
double b[][] = new double[K][V];

//initialize alpha
helper.arrayFill(a, 0.1);
//initialize alpha
helper.arrayFill(b, 0.01);
helper.arrayFill(delta, 1.0);
helper.arrayFill(lambda, 1.0);
```

```
/*
Overall Joint = Product(Product(phi(k,v)**c1,(v,1,V)),(k,1,K))*
Product(Product(phi(k,v)**(b(k,v)-1),(v,1,V)),(k,1,K))*
Product(Gamma(Sum(b(k,v),(v,1,V))),(k,1,K))/
Product(Product(Gamma(b(k,v)),(v,1,V)),(k,1,K))*
Product(Product(theta(d,k)**c0,(k,1,K)),(d,1,D))*
Product(Product(theta(d,k)**(a(d,k)-1),(k,1,K)),(d,1,D))*
Product(Gamma(Sum(a(d,k),(k,1,K))),(d,1,D))/
Product(Product(Gamma(a(d,k)),(k,1,K)),(d,1,D))
*/
//==== NEW VARIANT ====
//==== NEW SAMPLER FOR z ====
//==== FORMAT DxI ====
//Initialization of z[d][i] - topics
for (int d = 0; d < D; d++) {
    int I = w[d].length;
    z[d] = new int[I];
    for (int i = 0; i < I; i++) {
        z[d][i] = (int) (Math.random() * K);
        c0_1[d]++;
        c0[d][z[d][i]]++;
        c1_1[z[d][i]]++;
        c1[z[d][i]][w[d][i]]++;
    }
}
//For each iteration of the Markov Chain run the following:
double[] a_1 = new double[D];
double[] b_1 = new double[K];

for (int epoch = 0; epoch < 500000; epoch++) {
    System.err.println("EPOCH : " + epoch);
    //compute sum of alpha and beta and cache it
    for (int d = 0; d < D; d++) {
        a_1[d] = helper.sum(a[d]);
    }
    for (int k = 0; k < K; k++) {
        b_1[k] = helper.sum(b[k]);
    }

    //sampling loop for z[d][i]
```

```
for (int d = 0; d < D; d++) {
    int I = w[d].length;
    for (int i = 0; i < I; i++) {
        c0_1[d]--;
        c0[d][z[d][i]]--;
        c1_1[z[d][i]]--;
        c1[z[d][i]][w[d][i]]--;
        //Sample from full conditional
        double[] p = new double[K];
        for (int k = 0; k < K; k++) {
            p[k] = (((c0[d][k] + a[d][k]) / (c0_1[d] +
            a_1[d])) * ((c1[k][w[d][i]] + b[k][w[d][i]])
            / (c1_1[k] + b_1[k])));
        }
        //cumulate values
        for (int k = 1; k < K; k++) {
            p[k] += p[k - 1];
        }
        int k;
        double val = Math.log(Math.random()) + Math.log(p[K - 1]);
        for (k = 0; k < K; k++) {
            if (Math.log(p[k]) > val) {
                break;
            }
        }
        z[d][i] = k;
        c0_1[d]++;
        c0[d][z[d][i]]++;
        c1_1[z[d][i]]++;
        c1[z[d][i]][w[d][i]]++;
    }
}
if (epoch > burnin) {
    if (epoch % samplingInvterval == 0) {

        //sampler for q0
        for (int d = 0; d < D; d++) {
            for (int k = 0; k < K; k++) {
                if (c0[d][k] > 0) {
                    q0[d][k] = 1;
```

```
                for (int iter = 1; iter < c0[d][k]; iter++) {
                    q0[d][k] += helper.BernoulliSample(a[d][k] /
                    (a[d][k] + iter));
                }
            }
        }
    }
    //sampler for q1
    for (int d = 0; d < D; d++) {
        q1[d] = helper.BetaSample(a_1[d], (c0_1[d]));
    }

    //Sampler for lambda[l][k]
    int[][] sum1 = new int[L][K];
    for (int d = 0; d < D; d++) {
        for (int k = 0; k < K; k++) {
            for (int iter : (ArrayList<Integer>) f[d]) {
                sum1[iter][k] += q0[d][k];
            }
        }
    }
    double[][] sum2 = new double[L][K];
    for (int l = 0; l < L; l++) {
        for (int k = 0; k < K; k++) {

            for (int d : inversef[l]) {
                sum2[l][k] += (a[d][k] / lambda[l][k]) *
                Math.log(q1[d]);
                a[d][k] /= lambda[l][k];
            }

            lambda[l][k] = helper.GammaSample((mu +
            sum1[l][k]), 1) / (1 / mu - sum2[l][k]);

            for (int d : inversef[l]) {
                a[d][k] *= lambda[l][k];
            }
        }
    }
    //sampler for q3
```

```
for (int k = 0; k < K; k++) {
    for (int v = 0; v < V; v++) {
        if (c1[k][v] > 0) {
            q3[k][v] = 1;
            for (int iter = 1; iter <= c1[k][v]; iter++) {
                q3[k][v] += helper.BernoulliSample(b[k][v] /
                (b[k][v] + iter));
            }
        }
    }
}
//sampler for q4
for (int k = 0; k < K; k++) {
    q4[k] = helper.BetaSample(b_1[k], (c1_1[k]));
}
//Sampler for delta[t][k]
int[][] sum3 = new int[T][K];
for (int k = 0; k < K; k++) {
    for (int v = 0; v < V; v++) {
        for (int iter : (ArrayList<Integer>) g[v]) {
            sum3[iter][k] += q3[k][v];
        }
    }
}
double[][] sum4 = new double[T][K];
for (int t = 0; t < T; t++) {
    for (int k = 0; k < K; k++) {
        for (int v : inverseg[t]) {
            sum4[t][k] += (b[k][v] /
            delta[t][k]) * Math.log(q4[k]);
            b[k][v] /= delta[t][k];
        }
        delta[t][k] = helper.GammaSample((nu +
        sum3[t][k]), 1) / (1 / nu - sum4[t][k]);

        for (int v : inverseg[t]) {
            b[k][v] *= delta[t][k];
        }
    }
}
```

```
                }
            }
        }
    }
}
```

A.2 MIGA

```
package mu;

import java.util.ArrayList;
import java.util.Arrays;

public class TestCode {

    public void test(ArrayList[] f, int[][] w, Tester tester, int L,
    int M, int K, int V, double[] a, double[] b) {
        TestHelper helper = new TestHelper();

        int samplingInvterval = 1;
        int D = f.length;
        int T = 100;
        double mu = 1.0;
        int burnin = 10;

        //inversef[l][d] == 1 iff f[d][k] == 1 (inverse index of f)
        ArrayList<Integer>[] inversef = helper.invert(f, L);

        int c3[][] = new int[D][K];
        int c3_1[] = new int[D];
        int c0[][] = new int[D][M];
        int c0_1[] = new int[D];
        int c2[][] = new int[K][V];
        int c2_1[] = new int[K];
        int c1[][] = new int[M][K];
        int c1_1[] = new int[M];
        int y[] = new int[D];
```

```
double psi[][] = new double[D][M];
double theta[][] = new double[M][K];
double phi[][] = new double[K][V];
int z[][] = new int[D][];

double lambda[][] = new double[L][M];

//initialize alpha
Arrays.fill(a, 0.1);
//initialize beta
Arrays.fill(b, 0.01);
//initialize psi
helper.arrayFill(psi, 0.1);
//initialize lambda
helper.arrayFill(lambda, 1.0);

double a_1 = 0.0;
double b_1 = 0.0;
double[] psi_1 = new double[D];

for (int d = 0; d < D; d++) {
    y[d] = (int) (Math.random() * M);
    c0_1[d]++;
    c0[d][y[d]]++;
}

//c0 - d,m ; c2 - k,v ; c1 - m,k
for (int d = 0; d < D; d++) {
    int I = w[d].length;
    z[d] = new int[I];
    for (int i = 0; i < I; i++) {
        z[d][i] = (int) (Math.random() * K);
        c1_1[y[d]]++;
        c1[y[d]][z[d][i]]++;
        c2_1[z[d][i]]++;
        c2[z[d][i]][w[d][i]]++;
        c3[d][z[d][i]]++;
    }
}
```

```
for (int epoch = 0; epoch < 2001; epoch++) {
    System.err.println("EPOCH : " + epoch);

    a_1 = helper.sum(a);
    b_1 = helper.sum(b);
    for (int d = 0; d < D; d++) {
        psi_1[d] = helper.sum(psi[d]);
    }

    for (int d = 0; d < D; d++) {
        int I = w[d].length;
        for (int i = 0; i < I; i++) {
            c1[y[d]][z[d][i]]--;
            c1_1[y[d]]--;
            c2[z[d][i]][w[d][i]]--;
            c2_1[z[d][i]]--;
            c3[d][z[d][i]]--;
            //Sample from full conditional
            double[] p = new double[K];
            for (int k = 0; k < K; k++) {
                p[k] = (Math.pow(a_1 + c1_1[y[d]], -1)) *
                (Math.pow(b_1 + c2_1[k], -1)) *
                (c1[y[d]][k] + a[k]) *
                (c2[k][w[d][i]] + b[w[d][i]]);
            }
            for (int k = 1; k < K; k++) {
                p[k] += p[k - 1];
            }
            int k;
            double val = Math.random() * p[K - 1];
            for (k = 0; k < K; k++) {
                if (p[k] > val) {
                    break;
                }
            }
            z[d][i] = k;
            c1[y[d]][z[d][i]]++;
            c1_1[y[d]]++;
            c2[z[d][i]][w[d][i]]++;
            c2_1[z[d][i]]++;
```

```
                c3[d][z[d][i]]++;
            }
        }

        if (epoch >= burnin) {
            if (epoch % samplingInvterval == 0) {

                //For each iteration of the Markov Chain run the following:
                for (int d = 0; d < D; d++) {
                    int I = w[d].length;
                    c0_1[d]--;
                    c0[d][y[d]]--;
                    for (int i = 0; i < I; i++) {
                        c1[y[d]][z[d][i]]--;
                        c1_1[y[d]]--;
                    }

                    double[] p = new double[M];
                    for (int m = 0; m < M; m++) {
                        int counter = I;
                        double prod2 = 1.0;
                        prod2 = prod2 * (Math.pow(
                        helper.RisingFactorial(a_1 + c1_1[m], counter),-1));
                        int counter_M[] = new int[K];
                        for (int i = 0; i < I; i++) {
                            counter_M[z[d][i]]++;
                        }
                        double prod3 = 1.0;
                        for (int k = 0; k < K; k++) {
                            prod3 = prod3 *
                            (helper.RisingFactorial(
                            c1[m][k] + a[k], counter_M[k]));
                        }
                        p[m] = psi[d][m] *
                        Math.pow(psi_1[d], -1) *
                        prod2 * prod3;
                    }
                    for (int m = 1; m < M; m++) {
                        p[m] += p[m - 1];
                    }
```

```
        int m;
        double val = Math.random() * p[M - 1];
        for (m = 0; m < M; m++) {
            if (p[m] > val) {
                break;
            }
        }

        y[d] = m;
        for (int i = 0; i < I; i++) {
            c1[y[d]][z[d][i]]++;
            c1_1[y[d]]++;
        }
        c0_1[d]++;
        c0[d][y[d]]++;
    }
    int[][] sum1 = new int[L][M];

    for (int d = 0; d < D; d++) {
        for (int iter : (ArrayList<Integer>) f[d]) {
            sum1[iter][y[d]] += 1;
        }
    }
    //beta augmentation variable
    double q1[] = new double[D];
    //sampler for q1
    for (int d = 0; d < D; d++) {
        q1[d] = helper.BetaSample(psi_1[d], 1);
    }

    double[][] sum2 = new double[L][M];
    for (int l = 0; l < L; l++) {
        for (int m = 0; m < M; m++) {

            for (int d : inversef[l]) {
                sum2[l][m] += (psi[d][m]/ lambda[l][m])
                * Math.log(q1[d]);
                psi[d][m] /= lambda[l][m];
            }
            lambda[l][m] =
```

```
                            helper.GammaSample((mu + sum1[l][m]), 1)
                            / (1 / mu - sum2[l][m]);

                            for (int d : inversef[l]) {
                                psi[d][m] *= lambda[l][m];
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Appendix B

Example Schema Templates

Dirichlet Multinomial Collapsing Schema	
Schema field	**Value**
Name	CollapserDirMult
Description	Performs collapsing according to the Dirichlet multinomial Conjugacy.
Type	Collapsing Schema
Applicability Constraint	b: parameter, type is categorical or multinomial a: parameter, type is Dirichlet, prior of b
Handling multiple matches	Generate all matches as operations.
Action	Remove b: introduce a new constant node a_b with the terms from collapsing operation (based on the normalizer for Dirichlet multinomial)

Gamma Poisson Collapsing Schema	
Schema field	**Value**
Name	CollapserGaPois
Description	Performs collapsing according to the Gamma Poisson Conjugacy.
Type	Collapsing Schema
Applicability Constraint	b: parameter, type is Poisson a: parameter, type is Gamma, prior of b
Handling multiple matches	Generate all matches as operations.
Action	Remove b: introduce a new constant node a_b with the terms from collapsing operation (based on the normalizer for Gamma Poisson conjugacy)

As shown above, the schema relevant to a particular type (for example, collapsing) are fairly similar to each other, with the main differences occurring due to symbolic differences depending on the particular schema in question. For example, the main difference in the above arises from the difference in normalizers between the Dirichlet and Gamma distributions.

Beta Variable Augmenting Schema	
Schema field	**Value**
Name	AugmentorBetaVar
Description	Performs augmenting with a set of beta variables.
Type	Augmenting Schema
Applicability Constraint	Presence of $\Gamma(W_1)/\Gamma(W_1+W_2)$ in the likelihood b: match for W_1, parameter a: match for W_2, statistic
Handling multiple matches	Generate all matches as operations.
Action	The matched terms are augmented by a set of Beta random variables $q_{1:M}$. Here $q_m \sim Beta\left(\sum_{t=1}^{T} b(m,t), \sum_{t=1}^{T} a(m,t)\right)$. t and m are indices that are bound during the pattern matching process. Remove matched terms from the graph: introduce a new parameter node q_m as above, and connect it to the appropriate neighbours

An example augmentation would be $q_d \sim Beta\left(\sum_{k=1}^{K} a(d,k), \sum_{k=1}^{K} c_0(d,k)\right)$ as derived in section 4.3.

References

[1] Eric Atkinson, Cambridge Yang, and Michael Carbin. Verifying handcoded probabilistic inference procedures. *arXiv preprint arXiv:1805.01863*, 2018.

[2] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.

[3] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. *Journal of symbolic computation*, 3(1-2):203–216, 1987.

[4] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1–48. Citeseer, 2011.

[5] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent Dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[6] W.L. Buntine and S. Mishra. Experiments with non-parametric topic models. In *20th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, pages 881–890. ACM, 2014.

[7] Wray Buntine. Variational extensions to EM and multinomial PCA. In *European Conference on Machine Learning*, pages 23–34. Springer, 2002.

[8] Wray L Buntine and Swapnil Mishra. Experiments with non-parametric topic models. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 881–890, 2014.

[9] John Canny. GaP: a factor model for discrete data. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 122–129. ACM, 2004.

[10] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell.

Stan: A probabilistic programming language. *Journal of Statistical Software*, 20: 1–37, 2016.

[11] Bruce Char, Keith Geddes, and Gaston Gonnet. The Maple symbolic computation system. *ACM SIGSAM Bulletin*, 17(3–4):31–42, 1983.

[12] C. Chen, W. Buntine, N. Ding, L. Xie, and L. Du. Differential topic models. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 37(2):230–242, February 2015.

[13] Changyou Chen, Lan Du, and Wray Buntine. Sampling table configurations for the hierarchical Poisson-Dirichlet process. In *Proceedings of the 2011 European conference on Machine learning and knowledge discovery in databases*, pages 296–311, 2011.

[14] Mengen Chen, Xiaoming Jin, and Dou Shen. Short text classification improved by learning multi-granularity topics. In *22nd IJCAI*, 2011.

[15] Diana J Cole, Byron JT Morgan, and DM Titterington. Determining the parametric structure of models. *Mathematical biosciences*, 228(1):16–30, 2010.

[16] Mary Kathryn Cowles. Comments on 'The BUGS project: Evolution, critique and future directions'. *Statistics in Medicine*, 28(25):3068–3069, 2009. ISSN 1097-0258. doi: 10.1002/sim.3671. URL http://dx.doi.org/10.1002/sim.3671.

[17] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236, 2019.

[18] Lan Du, Wray Buntine, and Huidong Jin. A segmented topic model based on the two-parameter Poisson-Dirichlet process. *Machine Learning*, 81(1):5–19, 2010.

[19] Lan Du, Wray Buntine, and Mark Johnson. Topic segmentation with a structured topic model. In *Proceedings of NAACL-HLT*, pages 190–200, 2013.

[20] Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics letters B*, 195(2):216–222, 1987.

[21] Burçin Eröcal and William Stein. The Sage project: Unifying free mathematical software to create a viable alternative to Magma, Maple, Mathematica and MATLAB. In *International Congress on Mathematical Software*, pages 12–27. Springer, 2010.

[22] Bernd Fischer and Johann Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13 (3):483–508, 2003.

[23] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[24] Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis*. CRC press, 2013.

[25] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE TPAMI*, (6):721–741, 1984.

[26] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452, 2015.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[28] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[29] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic Programming. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 167–181, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2865-4. doi: 10.1145/2593882.2593900. URL http://doi.acm.org/10.1145/2593882.2593900.

[30] Thomas L Griffiths and Mark Steyvers. Finding scientific topics. *Proceedings of the National academy of Sciences*, 101(suppl 1):5228–5235, 2004.

[31] Gregor Heinrich. Parameter estimation for text analysis. Technical report, Technical report, 2005.

[32] Matthew D Hoffman and Andrew Gelman. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1):1593–1623, 2014.

[33] Changwei Hu, Piyush Rai, and Lawrence Carin. Non-negative matrix factorization for discrete data with hierarchical side-information. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 1124–1132, 2016.

[34] L.F. James, A. Lijoi, and I. Prünster. Posterior analysis for normalized random measures with independent increments. *Scandinavian Journal of Statistics*, 36(1): 76–97, 2009.

[35] Matthew Johnson, James Saunderson, and Alan Willsky. Analyzing hogwild parallel gaussian gibbs sampling. In *Advances in Neural Information Processing Systems*, pages 2715–2723, 2013.

[36] Chenliang Li, Haoran Wang, Zhiqian Zhang, Aixin Sun, and Zongyang Ma. Topic modeling for short texts with auxiliary word embeddings. In *39th ACM SIGIR*, pages 165–174, 2016.

[37] A. Lijoi and I. Prünster. Models beyond the Dirichlet process. In N.L. Hjort, C. Holmes, P. Müller, and S.G. Walker, editors, *Bayesian Nonparametrics*, pages 80–135. Cambridge University Press, 2010.

[38] Kar Wai Lim and Wray Buntine. Twitter opinion topic model: Extracting product opinions from tweets by leveraging hashtags and sentiment lexicon. In *CIKM '14: Proceedings of the 23rd ACM conference on Information and knowledge management*, pages 1319—1328, 2014.

[39] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. Rejoinder to commentaries on 'The BUGS project: Evolution, critique and future directions'. *Statistics in Medicine*, 28(25):3081–3082, 2009. ISSN 1097-0258. doi: 10.1002/sim.3691. URL http://dx.doi.org/10.1002/sim.3691.

[40] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. The BUGS project: evolution, critique and future directions. *Statistics in medicine*, 28(25): 3049–3067, 2009.

[41] David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS–a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, 10(4):325–337, 2000.

[42] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3: e103, 2017.

[43] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. 1 blog: Probabilistic models with unknown objects. *Statistical relational learning*, page 373, 2007.

[44] Tom Minka, John Winn, John Guiver, and David Knowles. Infer .NET 2.4, 2010. Microsoft Research Cambridge, 2010.

[45] Avi Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI*, pages 733–740, 2001.

[46] Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137:96, 2009.

[47] Avi Pfeffer. *Practical probabilistic programming*. Manning Publications Co., 2016.

[48] Martyn Plummer et al. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, volume 124, page 125. Vienna, Austria, 2003.

[49] Johann Schumann, Hamed Jafari, Tom Pressburger, Ewen Denney, Wray Buntine, and Bernd Fischer. AutoBayes program synthesis system users manual. 2008.

[50] D Spiegelhalter, A Thomas, N Best, and D Lunn. OpenBUGS user manual, version 3.0. 2. *MRC Biostatistics Unit, Cambridge*, 2007.

[51] William Stein. Sage days 4. *PDF). Archived from the original (PDF) on*, pages 06–27, 2007.

[52] Yu-Sung Su and Masanao Yajima. R2jags: A Package for Running JAGS from R. *R package version 0.03-08, URL http://CRAN. R-project. org/package= R2jags*, 2012.

[53] Y.W. Teh and M.I. Jordan. Hierarchical Bayesian nonparametric models with applications. In N.L. Hjort, C. Holmes, P. Müller, and S.G. Walker, editors, *Bayesian Nonparametrics*, pages 158–206. Cambridge University Press, 2010.

[54] Y.W. Teh, M.I. Jordan, M.J. Beal, and D.M. Blei. Hierarchical Dirichlet processes. *Journal of the ASA*, 101(476):1566–1581, 2006.

[55] Y.W. Teh, M.I. Jordan, M.J. Beal, and D.M. Blei. Hierarchical Dirichlet processes. *Journal of the American Statistical Association*, 101(476):1566–1581, 2006.

[56] Armando Teixeira-Pinto and Sharon-Lise T. Normand. Comments on 'The BUGS project: Evolution, critique, and future directions'. *Statistics in Medicine*, 28(25): 3075–3078, 2009. ISSN 1097-0258. doi: 10.1002/sim.3679. URL http://dx.doi.org/10.1002/sim.3679.

[57] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

[58] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *International Conference on Learning Representations*, 2017.

[59] Dustin Tran, Matthew D. Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, Alexey Radul, Matthew Johnson, and Rif A. Saurous. Simple, distributed, and accelerated probabilistic programming. In *Neural Information Processing Systems*, 2018.

[60] Jan Wielemaker, S Ss, and I Ii. SWI-Prolog 2.7-Reference Manual. 1996.

[61] Stephen Wolfram et al. *Mathematica*. Cambridge University Press, 1996.

[62] Yichuan Zhang, Zoubin Ghahramani, Amos J Storkey, and Charles A Sutton. Continuous Relaxations for Discrete Hamiltonian Monte Carlo. In *Advances in Neural Information Processing Systems*, pages 3194–3202, 2012.

[63] He Zhao, Lan Du, Wray Buntine, and Gang Liu. MetaLDA: a topic model that efficiently incorporates meta information. In *Proceedings of 2017 IEEE International Conference on Data Mining*, pages 635–644, 2017.

[64] He Zhao, Lan Du, Wray Buntine, and Gang Liu. MetaLDA: A topic model that efficiently incorporates meta information. In *ICDM*, pages 635–644, 2017.

[65] He Zhao, Lan Du, Wray Buntine, and Gang Liu. Leveraging external information in topic modelling. *Knowledge and Information Systems*, pages 1–33, 5 2018. ISSN 0219-1377. doi: 10.1007/s10115-018-1213-y.

[66] He Zhao, Lan Du, Wray Buntine, and Gang Liu. Leveraging external information in topic modelling. *KAIS*, pages 1–33, 2018.

[67] He Zhao, Lan Du, Wray Buntine, and Mingyuan Zhou. Inter and intra topic structure learning with word embeddings. In *ICML*, pages 5887–5896, 2018.

[68] He Zhao, Piyush Rai, Lan Du, and Wray Buntine. Bayesian multi-label learning with sparse features and labels, and label co-occurrences. In *AISTATS*, pages 1943–1951, 2018.

[69] He Zhao, Lan Du, Guanfeng Liu, and Wray Buntine. Leveraging meta information in short text aggregation. In *ACL*, pages 4042–4049, Florence, Italy, July 2019.

[70] Mingyuan Zhou and Lawrence Carin. Negative binomial process count and mixture modeling. *IEEE Trans. PAMI*, 37(2):307–320, 2015.

[71] Mingyuan Zhou, Lingbo Li, David Dunson, and Lawrence Carin. Lognormal and Gamma mixed negative binomial regression. In *ICML*, volume 2012, page 1343, 2012.

[72] Mingyuan Zhou, Yulai Cong, and Bo Chen. Augmentable gamma belief networks. *JMLR*, 17(163):1–44, 2016.