DPMM.jl

Ekin Akyürek
CSAIL, Massachusetts Institute of Technology
Koç University
akyurek@mit.edu

July 9, 2019

Abstract

Contents

1 Introduction

This document is a on a parallel implementation of Dirichlet process mixture models (DPMM) in Julia by Ekin Akyürek with supervision of John W. Fischer. In the following sections, we will explain DPMMs and present different samplers. Then, we will elucidate the software architecture of DPMM.jl[1] to demonstrate composability and expressiveness of the package. In the final section, we will compare the speed and convergence properties of the implemented samplers. Finally, we will benchmark our package with the existing DPMM softwares.

2 Background

2.1 Mixture Models

Mixture models are statistical models for representing data which comes from a set of different probability distributions. A mixture model is generally parameterized through a mixture (i.e component) of parameters and weights assigned to each distribution which can be considerd as prior probabilities of the mixtures in the Bayesian setting. The more weight attached to a mixture, the more data is generated from that mixture. In this work, we will mostly use the notation presented in the document[5]:

Data

N	Number of data vectors
D	Dimension of data vectors
$x_i \in R^D$	i^th data vector
$X = (x_1, x_2, ..., x_N)$	Set of data vectors
X_i	Set of data vectors, excluding x_i
X_k	Set of data vectors from mixture component k
X_{ki}	Set of data vectors from mixture component k, excluding x_i
N_k	Number of data vectors from mixture component k
N_{ki}	Number of data vectors from mixture component k, except x_i

Model

K	Number of components in a finite mixture model
z_i	Discrete latent state indicating the component observation x_i belongs to
$\mathbf{z} = (z_1, z_2, ..., z_N)$	Latent states for all observations $x_i, ..., x_n$
$\mathbf{z}_i$	Set of all latent states, excluding z_i
θ_k	Component parameters (e.g. μ_k, Σ_k in a Gaussian mixture model)
π_k	Prior probability that data vector x_i will be assigned to mixture component k
$\pi = (\pi_1, \pi_2, ..., \pi_K)$	Vector of prior assignment probabilities for each of K components
β	Hyper-parameters of the prior for θ in a Bayesian setting

[1]`https://github.com/ekinakyurek/DPMM.jl`

2.2 Dirichlet Process Mixture Models (DPMM)

2.2.1 Dirichlet Process (DP)

There are many ways to interpret DP which are discussed in the following sections in detail. Formally, it is a random process whose realizations are probability distributions. Moreover, the constructive definition of Dirichlet process relies on the stick-breaking processes. Also, Chinese Restaurant Process(CRP) and Pólya Urn Scheme also lead to DP. All these definitions are related with each other according to de Finetti's exchangeability theorem.

DP is parameterized by concentration parameter α and G_0 base distribution. One can show that α controls how similar draws will be to the G_0. G is used to denote samples drawn from DP.

2.2.1.1 Stick-Breaking Construction

Stick-Breaking provides a simple way to draw samples from DP, but it requires countably infinite summation to be performed. Stick-Breaking steps are as follows.

$$v_1, v_2, ..., v_i, ... \sim Beta(1, \alpha) \tag{1}$$

$$\pi_i = v_i \prod_{j=1}^{i-1}(1 - v_j) \tag{2}$$

$$\theta_1, \theta_2, ..., \theta_i, ... \sim G_0 \tag{3}$$

$$G = \sum_{i=1}^{\infty} \pi_i \delta_{\theta_i} \tag{4}$$

Here δ_{θ_i} is an indicator function centered on θ_i, namely $\delta_{\theta_i}(\theta)$ is zero everywhere except $\theta = \theta_i$. Note that π_i's approach to zero as i goes to infinity which makes it possible to approximate G using a finite summation instead of an infinite one.

2.2.1.2 Chinese Restaurant Process (CRP)

Chinese restaurant process is a discrete process which is named after the analogy of seating customers at tables in a restaurant. There is two options for every arriving customer, she can be seated in an existing table with others already seating at it or she can be assigned to an empty table. Let's say there are customers $1 : N$ and we will seat each customer sequentially with the following probabilities in Equation 2.2.1.2, where c_k is the number of customers currently seated at k^{th} table, i is current number of customers seated at all tables in the restaurant and z_i is the label of the table that $i'th$ customer is seated.

$$z_i|z_1, ..., z_{i-1} = \begin{cases} P(z_i = k) = \frac{c_k}{i-1+\alpha} & \text{for an existing table} \\ P(z_i = K+1) = \frac{\alpha}{i-1+\alpha} & \text{for opening a new table} \end{cases} \tag{5}$$

Note that this processes is independent of the arriving order of customers. This is called the exhangeability property of CRP [4].

If we assume a base measure G_0 as a prior for table/cluster parameters and assign each table a probability measure sampled from G_0, the process becomes CRP Mixture Model or Pólya Urn Scheme.

$$\theta_1 ..., \theta_K, ... \sim G_0 \tag{6}$$

$$z_i | z_1, ..., z_{i-1} = \begin{cases} P(z_i = k) = \frac{c_k}{i-1+\alpha} & \text{for an existing table/cluster} \\ P(z_i = K+1) = \frac{\alpha}{i-1+\alpha} & \text{for opening a new table/cluster} \end{cases} \tag{7}$$

$$X_i | z_i \sim f(X_i | \theta_{z_i}) \tag{8}$$

Exchangeability allows us to show that above model is equivalent to [2]:

$$\begin{aligned} G &\sim DP(\alpha, G_0) \\ \theta_i | G &\sim G \qquad i \in 1, ..., N \\ X_i | \theta_i &\sim f(X_i | \theta_i) \qquad i \in 1, ..., N \end{aligned}$$

2.2.1.3 Formal Definition

We stated that samples from a Dirichlet process are probability distribution themselves. Let one these samples to be a probability distribution over the space S. Let $\{A_i\}_{i=1}^n$ denote a measurable partition of S. For any measurable partition of S below statement holds (*Dir* is Dirichlet Distribution):

$$G \sim DP(\alpha, G_0) \implies (G(A_1), G(A_2), ..., G(A_N)) \sim Dir(\alpha G_0(A_1), ..., \alpha G_0(A_N))$$

This defines a Dirichlet Process, however it doesn't allow to draw samples relying on this definition unlike CRP or Pólya Urn.

2.2.2 Infinite Mixture Models

As we show in CRP section, Dirichlet process is a very good prior on mixture models' parameters as it can model infinite number of clusters. Note that every finite data realization of DPMM is a finite mixture model with Dirichlet distribution as shown in the formal definition. One can use finite a stick-breaking process to construct a DP, however it results in approximation errors. On the other hand, CRP Mixture Model provides an exact way of sampling and enables inference on the mixture data. Finally, these two different representations are summarized in Figure 1. The following section discusses how to do inference on DPMM.

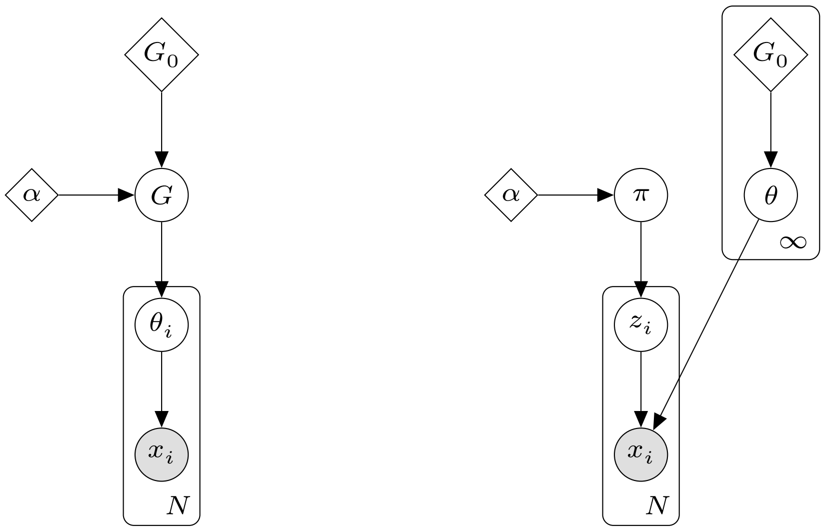


Figure 1: Two equivalent graphical representation of DPMM

2.3 Inference in DPMM

Inference problem that we interested in is to obtain cluster assignments for each data point given base distribution G_0 and α parameter. We will use conjugate prior which assures that the posterior probability of cluster parameters are in the same distribution family with the prior. Let's investigate conjugate priors for Gaussian and Multinomial distributions.

2.3.1 Posteriors & Sufficent Statistics

2.3.1.1 Gaussian Conjugate: Normal-Wishart Distribution

Multivariate Gaussian Distribution is generally parameterized by a mean vector μ and covariance matrix Σ. However, this parameterization results in expensive likelihood computations due to Σ^{-1} term in the pdf function. Therefore, we prefer precision J and μ parameterization. Normal-Wishart(NW) Distribution[7] is the prior distribution for Gaussian with unknown μ and unknown J.

Normal-Wishart is parameterized by:

μ_0	prior mean for μ
Ψ_0	proportional to prior mean Σ
λ_0	how strongly we believe in μ_0
ν_0	how strongly we believe in Ψ_0

NW has separable pdf function composed of a Gaussian and a Wishart($\mathcal{W}$) distribution.

$$p(\mu, J|\mu_0, \lambda_0, \Psi_0, \nu_0) = \mathcal{N}\left(\mu \middle| \mu_0, \lambda J\right) \mathcal{W}(J|\Psi_0, \nu_0)$$

To better understand how to sample from Wishart distribution see the reference for the Bartlett decomposition method [9]. If we use sample mean($\hat{\mu}$) and sample covariance($\hat{\Sigma}$)

matrix as sufficient statistics, posterior parameters for the Normal-Wishart distribution is given by the below equations 9 to 13 [5]:

$$p(\mu, J|\mathcal{X}) = NW(\mu, J|\mu_N, \Psi_N, \lambda_N, \nu_N) \tag{9}$$
$$\mu_N = \lambda_0\mu_0 + N\hat{\mu} \tag{10}$$
$$\lambda_N = \lambda_0 + N \tag{11}$$
$$\nu_N = \nu_0 + N \tag{12}$$
$$\Psi_N = \Psi_0 + \hat{\Sigma} + \frac{\lambda_0 N}{\lambda_0 + N}(\hat{\mu} - \mu_0)(\hat{\mu} - \mu_0)^T \tag{13}$$

It is also possible to store $t = N\hat{x}$ and $T = \sum_{i=1}^{N} xx^T$ as sufficient statistics, which is computationally more efficient. Then the equations become:

$$\begin{aligned}
\mu_N &= \lambda_0\mu_0 + t \\
\lambda_N &= \lambda_0 + N \\
\nu_N &= \nu_0 + N \\
\Psi_N &= \Psi_0 + T + \lambda_0\mu_0\mu_0^T - \lambda_N\mu_N\mu_N^T
\end{aligned}$$

Low-Rank Updates for Ψ_N In the code, we store Ψ_N in Cholesky factorized form to sample from it efficiently. However, we need to calculate Cholesky factorization, which is an $\mathcal{O}(n^3)$ operation, every time if we remove or add single data point. To overcome this challenge, there is a fast algorithm to compute the updated Cholesky factorization from the previous Cholesky factorization [8]. We utilize Rank-1 update and Rank-1 down-date methods to speed up the collapsed Gibbs sampler.

2.3.1.2 Multinomial Conjugate: Dirichlet Distribution

Multinomial probability distribution is used for mixture distribution for discrete data, and Dirichlet distribution is the conjugate prior for Multinomial distribution. Dirichlet distribution is parameterized by $\vec{\alpha}$ where $\vec{\alpha}_i$ corresponds to pseudo-count of Multinomial event i. In addition to, we also store the logarithm of $\vec{\alpha}$ values for fast log-likelihood computations. The likelihood of probabilities of a Multinomial distribution given by a Dirichlet prior is:

$$f(p_1, \dots, p_K; \vec{\alpha}_1, \dots, \vec{\alpha}_K) = \frac{1}{\mathrm{B}(\alpha)} \prod_{i=1}^{K} p_i^{\vec{\alpha}_i - 1} \tag{14}$$

Sufficient statistics t for Dirichlet prior and Multinomial distribution is the count vector of events $t = \vec{c}$ where:

$$\vec{c}_i = \sum_{j=1}^{N} \mathcal{X}_{ik}$$

Then, posterior distribution of Dirichlet is given by:

$$\vec{\alpha}' = \vec{\alpha} + \vec{c}$$

2.3.1.3 Notes on Non-Conjugate Distributions

Posteriors to non-conjugate priors should be calculated as a multiplication of prior and data likelihood. In this case, it does not create an analytic distribution, thus, cannot be normalized analytically. Hence, one should program an effective sampler for this type of posteriors.

2.3.2 Collapsed Gibbs Sampler

When using conjugate priors for cluster parameters, marginal likelihood of data can be obtained by analytically integrating out the cluster parameter, θ_i. For Gaussian parameters the marginal likelihood is

$$p(\mathcal{X}) = \int_{\mu}\int_{\Sigma} p(\mathcal{X}, \mu, \Sigma)p(\mu, \Sigma)d\mu d\Sigma$$

The likelihood of a new data x^* is given by

$$p(x^*|\mathcal{X}) = \frac{p(x^*, \mathcal{X})}{p(\mathcal{X})}$$

It turns out that the posterior likelihood (posterior-predictive) of new data has Multivariate Student's t-distribution [5] for NW-Gaussian pair:

$$p(x^*|\mathcal{X}) = \mathcal{T}(x^*|\mu_N, \frac{(\lambda_N + 1)}{\lambda_N(\nu_N - D + 1)}\Psi_N, \nu_n - D + 1)$$

The posterior predictive likelihood of Dirichlet-Multinomial pair has a rather complicated definition:

$$p(x^*|\mathcal{X}) = \frac{\Gamma(n+1)}{\prod_{j=1}^{K}\Gamma(x_j^*+1)} \frac{\Gamma(\sum_{j=1}^{K}\alpha_j')}{\prod_{j=1}^{K}\Gamma(\alpha_j')} \frac{\prod_{j=1}^{K}\Gamma(x_j^*+\alpha_j')}{\Gamma(n+\sum_{j=1}^{K}\alpha_j')}$$

The collapsed inference in DPMM can be done by using CRP procedure. The difference is that instead of using $f(x|\theta_{z_i})$ likelihood, we use $p(x^*|\mathcal{X})$ posterior-predictive and we never sample θ's explicitly [5].

$$z_i|z_i \sim \begin{cases} P(z_i = k) = p(x|\beta, \mathcal{X}_{ki})\frac{N_{ki}}{N-1+\alpha} & \text{for an existing table/cluster} \\ P(z_i = K+1) = p(x|\beta)\frac{\alpha}{N-1+\alpha} & \text{for a new table/cluster} \end{cases}$$

Pseudo-code [5] of collapsed gibbs sampler is presented in Algorithm 1.

Algorithm 1 Collapsed Gibbs sampler for an infinite mixture model.

```
1: Choose an initial $\mathbf{z}$.
2: for M iterations do                                          ▷ Gibbs Sampling Iterations
3:     for i = 1 to N do
4:         Remove $\mathbf{x}_i's$ statistics from component $z_i$        ▷ Old assignment for $\mathbf{x}_i$
5:         for k = 1 to K do                                    ▷ For all existing clusters
6:             Calculate $P(z_i = k|\mathbf{z}_i, \alpha) = \frac{N_{ki}}{N+\alpha-1}$ as in CRP
7:             Calculate $p(\mathbf{x}_i|\mathcal{X}_{ki}, )$ using posterior predictive distribution.
8:             Calculate $P(z_i = k|\mathbf{z}_i, \mathcal{X}, \alpha, ) \propto P(z_i = k|\mathbf{z}_i, \alpha)p(\mathbf{x}_i|\mathcal{X}_{ki}, )$
9:         end for
10:        Calculate $P(z_i = K+1|\mathbf{z}_i, \alpha) = \frac{\alpha}{N+\alpha-1}$ as in CRP    ▷ Consider opening a new cluster
11:        Calculate $p(\mathbf{x}_i|)$ using prior-predictive distribution.
12:        Calculate $P(z_i = K+1|\mathbf{z}_i, \mathcal{X}, \alpha, ) \propto P(z_i = K+1|\mathbf{z}_i, \alpha)p(\mathbf{x}_i|)$
13:        Sample $k_{new}$ from $P(z_i = k|\mathbf{z}_i, \mathcal{X}, , )$ after normalizing
14:        Add $z_i$'s statistics to the component $z_i = k_{new}$.     ▷ New assignment for $\mathbf{x}_i$
15:        If any component is empty, remove it and decrease K
16:    end for
17: end for
```

Remarks

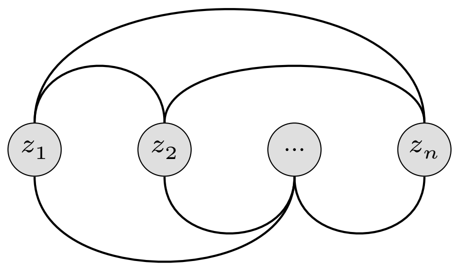
1. **Parallelism Over Data Points:** Collapsed sampler cannot be parallelized over data points, because for each data point we need to update the cluster statistics, yet we don't know which cluster's statistic will change and what they will be beforehand.

2. **Parallelism Over Clusters:** Collapsed sampler *can* be parallelized over clusters on the calculation of data likelihoods. Typically, K is significantly less than N, so cluster parallelism does not help much.

3. **Empirical Convergence**: Collapsed sampler converges slowly when it starts with an underestimated number of clusters in which case close clusters tend to form a single cluster.

2.3.3 Quasi-Collapsed Gibbs Sampler

Note that in collapsed sampling every latent variable z_i is connected with each other in the undirected representation of DP.



So, it is not possible to sample z_i's in parallel for an exact sampling. However, the affect of change in z_i is negligible when $N \to \infty$. Therefore, we can break above connections for an approximate collapsed sampling algorithm which is parallelizable. It also corresponds to removing 4^{th} and 14^{th} lines, and calculation of posterior-predictive distribution before the iteration in Algorithm 1. So, in this revised version of the algorithm, instead of updating posterior-predictive for each data point, we update it only in the beginning of an iteration. The quasi-collapsed sampler algorithm is presented in Algorithm 2 and sampling scheme is presented in the equation:

$$z_i|z \sim \begin{cases} P(z_i = k) = p(x|\beta, \mathcal{X}_k)\frac{N_k}{N+\alpha} & \text{for an existing table/cluster} \\ P(z_i = K+1) = p(x|\beta)\frac{\alpha}{N+\alpha} & \text{for a new table/cluster} \end{cases}$$

Remarks

1. **Parallelism Over Data Points:** Quasi-Collapsed sampler can be parallelized over data points, since we do not update cluster statistics during an iteration.

2. **Parallelism Over Clusters:** Quasi-Collapsed sampler can be parallelized over clusters for the calculation of data likelihoods. Calculation of posterior-predictive distributions is also parallelizable. In most practical examples, K is significantly less than N, so cluster parallelism does not help much.

3. **Empirical Convergence**: Quasi-Collapsed sampler sometimes does not converge to correct number of clusters. Closer clusters tend to form a single cluster in that situation.

Algorithm 2 Quasi-Collapsed Gibbs sampler for an infinite mixture model.

```
1: Choose an initial $\mathbf{z}$.
2: for M iterations do                                   ▷ Gibbs Sampling Iterations
3:     Update posterior-predictive distributions with $\mathbf{z}$
4:     for i = 1 to N do
5:         for k = 1 to K do                             ▷ For all existing clusters
6:             Calculate $P(z_i = k|\mathbf{z}, \alpha) = \frac{N_k}{N+\alpha}$ as in CRP
7:             Calculate $p(\mathbf{x}_i|\mathcal{X}_k, )$ using posterior predictive distribution with old assignments $\mathbf{z}$
8:             Calculate $P(z_i = k|\mathbf{z}, \mathcal{X}, \alpha, ) \propto P(z_i = k|\mathbf{z}, \alpha)p(\mathbf{x}_i|\mathcal{X}_k, )$
9:         end for
10:        Calculate $P(z_i = K+1|\mathbf{z}, \alpha) = \frac{\alpha}{N+\alpha}$ as in CRP    ▷ Consider opening a new cluster
11:        Calculate $p(\mathbf{x}_i|)$ using prior-preditive distribution.
12:        Calculate $P(z_i = K+1|\mathbf{z}, \mathcal{X}, \alpha, ) \propto P(z_i = K+1|\mathbf{z}, \alpha)p(\mathbf{x}_i|)$
13:        Sample $k_{new}$ from $P(z_i = k|\mathbf{z}, \mathcal{X}, , )$ after normalizing
14:    end for
15:    If any component is empty, remove it and decrease K
16: end for
```

2.3.4 Direct Gibbs Sampler

Collapsed and Quasi-Collapsed algorithms use CRP procedure to construct DP. However, there is another approach from which mixture weights and mixture parameters are sampled directly. There is, however, an infinite number of clusters in DP, but we can represent uninstantiated clusters' weights by a single probability π_{K+1}. So, π_{K+1} is the sum of all uninstantiated cluster weights and θ_{K+1} is sampled from the base distribution for a new cluster. Direct sampler scheme is given in the below:

$$
\begin{aligned}
\pi_1 ..., \pi_K, \pi_{K+1} &\sim Dir(N_1, ..., N_K, \alpha) \\
\theta_1 ..., \theta_K &\sim G_0(\theta|z, \beta) \\
z_i|z &\sim \begin{cases} P(z_i = k) = f(X_i|\theta_k)\pi_k & \text{for an existing cluster} \\ P(z_i = K+1) = f(X_i|\theta_{K+1})\pi_{K+1} & \text{for a new cluster} \end{cases}
\end{aligned}
$$

1. **Parallelism Over Data Points:** Direct sampler can be parallelized over data points.
2. **Parallelism Over Clusters:** Direct sampler can be parallelized over clusters for likelihood calculations. Sampling of θ parameters is also parallelizable in a similar fashion. In practice, K is significantly less than N, so cluster parallelism does not help much.
3. **Empirical Convergence**: Direct sampler converges slowly when it instantiated with underestimated number of clusters in which case closer clusters tend to form a single cluster.

Algorithm 3 Direct Gibbs sampler for an infinite mixture model.

```
1:  Choose an initial $\mathbf{z}$.
2:  for M iterations do                                              ▷ Gibbs Sampling Iterations
3:      for k = 1 to K do                                            ▷ For all existing clusters
4:          Sample $\theta_k \sim G_0(\theta|\beta, \mathcal{X}_k)$                                  ▷ if $N_k \neq 0$
5:      end for
6:      Sample $(\pi_1, \pi_2, ..., \pi_{K+1}) \sim Dir(N_1, ..., N_K, \alpha)$
7:      for i = 1 to N do
8:          for k = 1 to K do                                        ▷ For all existing clusters
9:              Calculate $p(\mathbf{x}_i|_k)$ using mixture distribution
10:             Calculate $P(z_i = k|\mathbf{z}, \mathcal{X}, \alpha, ) \propto \pi_k p(\mathbf{x}_i|_k)$
11:         end for
12:         Calculate $p(\mathbf{x}_i|\theta_{K+1})$ using mixture distribution
13:         Calculate $P(z_i = K+1|\mathbf{z}, \mathcal{X}, \alpha, ) \propto \pi_{K+1} p(\mathbf{x}_i|_{K+1})$
14:         $z_i \leftarrow k_{new}$ from $P(z_i = k|\mathbf{z}, \mathcal{X}, , )$ after normalizing
15:     end for
16: end for
```

2.3.5 Quasi-Direct Gibbs Sampler

When $N \to \infty$, we can estimate $(\pi_1, \pi_2, ..., \pi_{K+1})$ with their proportions:

$$\pi_k = \frac{N_k}{N + \alpha}, \pi_{K+1} = \frac{\alpha}{N + \alpha}$$

So, we can get faster algorithm which skips sampling from Dirichlet process:

$$\theta_1 ..., \theta_K \sim G_0(\theta|z, \beta)$$
$$z_i|z_i \sim \begin{cases} P(z_i = k) = f(X_i|\theta_k)\frac{N_k}{N+\alpha} & \text{for an existing cluster} \\ P(z_i = K + 1) = f(X_i|\theta)\frac{\alpha}{N+\alpha} & \text{for a new cluster} \end{cases}$$

We iteratively repeat the above steps. Pseudocode of quasi-direct Gibbs sampler is presented in Algorithm 4.

Algorithm 4 Quasi-Direct Gibbs sampler for an infinite mixture model.

```
 1: Choose an initial z.
 2: for M iterations do                                              ▷ Gibbs Sampling Iterations
 3:     for k = 1 to K do                                            ▷ For all existing clusters
 4:         Sample $\theta_k \sim G_0(\theta|\beta, \mathcal{X}_k)$                                    ▷ if $N_k \neq 0$
 5:     end for
 6:     for i = 1 to N do
 7:         for k = 1 to K do                                        ▷ For all existing clusters
 8:             Calculate $P(z_i = k|\mathbf{z}, \alpha) = \frac{N_k}{N+\alpha}$ as in CRP
 9:             Calculate $p(\mathbf{x}_i|_k)$ using mixture distribution
10:             Calculate $P(z_i = k|\mathbf{z}, \mathcal{X}, \alpha, ) \propto P(z_i = k|\mathbf{z}, \alpha)p(\mathbf{x}_i|_k)$
11:         end for
12:         Calculate $P(z_i = K + 1|\mathbf{z}, \alpha) = \frac{\alpha}{N+\alpha}$ as in CRP   ▷ Consider opening a new
    cluster
13:         Calculate $p(\mathbf{x}_i|\theta^*)$ using mixture distribution
14:         Calculate $P(z_i = K + 1|\mathbf{z}, \mathcal{X}, \alpha, ) \propto P(z_i = K + 1|\mathbf{z}, \alpha)p(\mathbf{x}_i|_{K+1})$
15:         $z_i \leftarrow k_{new}$ from $P(z_i = k|\mathbf{z}, \mathcal{X}, , )$ after normalizing
16:     end for
17: end for
```

Remarks

1. **Parallelism Over Data Points:** Same as Direct Sampler.
2. **Parallelism Over Clusters:** Same as Direct Sampler.
3. **Empirical Convergence**: It doesn't converge robustly.

2.3.6 Split-Merge Gibbs Sampler

DP Split-Merge algorithm [3] is a parallelizable Gibbs sampler for DPMM. It converges well, is easily parallelizable and able to work with non-conjugate priors. Algorithm starts with a non-ergodic restricted sampler assigning cluster labels to data points without creating a new cluster as in equations 15 to 17. Then, another sampler assigns sub-cluster labels within the existing clusters, equations 18 to **??**. Split moves are proposed over subclusters, while merge moves are proposed over existing clusters. Through split-merge moves, algorithm becomes ergodic and has limiting guarantees. Restricted sampler is similar to Direct Sampler, except that it does not create new clusters:

$$\pi_1 ..., \pi_K, \pi_{K+1} \sim Dir(N_1, ..., N_K, \alpha) \tag{15}$$
$$\theta_1 ..., \theta_K \sim G_0(\theta | z, \beta) \tag{16}$$
$$z_i | z \sim P(z_i = k) \propto f(X_i | \theta_k) \pi_k \tag{17}$$

Another sampler assigns subcluster labels for existing clusters as shown in below:

$$(\bar{\pi}_{kl}, \bar{\pi}_{kr}) \sim \mathrm{Dir}(N_{kl} + \alpha/2, N_{kr} + \alpha/2) \tag{18}$$
$$\bar{\theta}_{k\{l,r\}} \sim G_0(\theta | \bar{z}_{k\{l,r\}}, \beta) \tag{19}$$
$$\bar{z}_i \sim \sum_{j \in \{l,r\}} \pi_{z_i} f_x(x_i; \bar{\theta}_{z_i j}) I[\bar{z}_i = j] \tag{20}$$

Split and Merge moves are proposed via Metropolis-Hastings method. With a specific proposal distributions split move's hasting ratio H_{split} is:

$$H_{\text{split}} = \frac{\alpha}{\Gamma(N_k) f_x\left(x_{I_k;\lambda}\right)} \Gamma(N_{kl}) f_x\left(x_{I_{kl}}; \lambda\right) \Gamma(N_{kr}) f_x\left(x_{I_{kr}}; \lambda\right)$$

For newly created clusters one can keep track of marginal data likelihood, $\overline{\mathcal{L}}_m = f_x\left(x_{\{m,\ell\}}; \bar{\theta}_{m,\ell}\right) \cdot f_x\left(x_{\{m,r\}}; \bar{\theta}_{m,r}\right)$, and defer split moves when $\mathcal{L}_m$ oscillates enough. This will save us from computing new auxiliary subcluster variables in each iteration.

Similarly merge move's hasting ratio is:

$$H_{\text{merge}} = 0.01 \frac{\Gamma(N_{k1} + N_{k2})}{\alpha \Gamma(N_{k1}) \Gamma(N_{k2})} \frac{p(x|\hat{z})}{p(x|z)} \frac{\Gamma(\alpha)}{\Gamma(\alpha + N_{k1} + N_{k2})} \frac{\Gamma(\alpha/2 + N_{k1}) \Gamma(\alpha/2 + N_{k2})}{\Gamma(\alpha/2)\Gamma(\alpha/2)}$$

1. **Parallelism Over Data Points:** Restricted sampler can be parallelized over data points. So, the assignments of clusters and subclusters can be done in parallel for a data point.

2. **Parallelism Over Clusters:** Direct sampler can be parallelized over clusters on the calculation of data likelihoods. Sampling of θ and $\bar{\theta}$ parameters is also parallelizable in a similar fashion. In addition, split moves can be proposed in parallel. However, in most practical problems, K is significantly less than N, so cluster parallelism does not help much.

3. **Emprical Convergence**: Split merge has the best convergence characteristics compared to other methods investigated in this report.

Algorithm 5 Split-Merge Gibbs sampler for an infinite mixture model.

```
Choose an initial $\mathbf{z}$.
Choose initial subcluster assignments $\bar{z}$.
for M iterations do                                                  ▷ Gibbs Sampling Iterations
    for k = 1 to K do                                                ▷ For all existing clusters
        Sample $\theta_k \sim G_0(\theta|\beta, \mathcal{X}_k)$
        Sample $\theta_{kr} \sim G_0(\theta|\beta, \mathcal{X}_{k,r})$
        Sample $\theta_{kl} \sim G_0(\theta|\beta, \mathcal{X}_{k,l})$
        Assign splitability by tracking $\mathcal{L}_m$ history.
    end for
    Sample $(\pi_1, \pi_2, ..., \pi_{K+1}) \sim Dir(N_1, ..., N_K, \alpha)$
    Sample $(\bar{\pi}_{kl}, \bar{\pi}_{kr}) \sim \mathrm{Dir}(N_{kl} + \alpha/2, N_{kr} + \alpha/2)$
    for i = 1 to N do                                                ▷ Restricted Sampler
        for k = 1 to K do
            Calculate $p(\mathbf{x}_i|_k)$ using mixture distribution
            Calculate $P(z_i = k|\mathbf{z}, \mathcal{X}, \alpha, ) \propto \pi_k p(\mathbf{x}_i|_k)$
        end for
        $z_i \leftarrow k_{new}$ from $P(z_i = k|\mathbf{z}, \mathcal{X}, , )$ after normalizing
        $\bar{z}_i \leftarrow j_{new}$ from $P(\bar{z}_i = j|\mathbf{z}, \mathcal{X}, , ) = \sum_{j\in\{l,r\}} \pi_{z_i} f_x(x_i; \bar{\theta}_{z_i j}) I[\bar{z}_i = j]$
        for k = 1 to K do                                            ▷ Split Moves
            if k is Splitable then
                Calculate Hastings ratio and accept split move with $p = min(1, H)$.
                $H_{\text{split}} = \frac{\alpha}{\Gamma(N_k) f_x\left(x_{I_k}; \lambda\right)} \Gamma\left(N_{kl}\right) f_x\left(x_{I_{kl}}; \lambda\right) \Gamma\left(N_{kr}\right) f_x\left(x_{I_{kr}}; \lambda\right)$
            end if
        end for
        for k = 1 to K do                                            ▷ Merge Moves
            Calculate Hastings ratio and accept merge move with p=min(1,H).
            $H_{\text{merge}} = 0.01 \frac{\Gamma(N_{k1}+N_{k2})}{\alpha\Gamma(N_{k1})\Gamma(N_{k2})} \frac{p(x|\hat{z})}{p(x|z)} \frac{\Gamma(\alpha)}{\Gamma(\alpha+N_{k1}+N_{k2})} \frac{\Gamma(\alpha/2+N_{k1})\Gamma(\alpha/2+N_{k2})}{\Gamma(\alpha/2)\Gamma(\alpha/2)}$
        end for
    end for
end for
```

2.4 Parallel Inference

Different kinds of parallelism can be implemented for the Gibbs sampling algorithms. We will consider parallelism in two different categories: parallelism over data points, parallelism over clusters.

2.4.1 Parallelism over Data Points

If the sampling of labels is not conditioned on new labels assigned to other data points, we can parallelize the sampler over data points. In this work, we only utilize this type of parallelism due to efficiency, but we will discuss other types of parallelism in the report for completeness. We will step down the generic scheme for the parallelism.

a) First step to obtain parallelism of this type is distributing the data and positions of randomly instantiated labels to the worker processes at the beginning.

b) At each iteration, master process samples mixture weights and cluster parameters or posterior (depending on the sampler), and send it to the worker processes to induce their sampling loop.

c) Each worker process considers its part of the data, samples corresponding cluster labels and subcluster-labels if relevant (split-merge Gibbs sampler).

d) Then, each worker calculates its local sufficient statistics and sends it to master process.

e) The master process gathers the sufficient statistics and updates or samples the cluster and sub-cluster parameters.

f) If only this type of parallelism is deployed, the master process is responsible for split and merge proposals too.

Multi-Machine Parallelism: One may want to parallelize an algorithm using a High Performance Computing (HPC) system which is a cluster of multiple computers. Unfortunately, the parallelism scheme described above works efficiently when all the workers are in the same machine. In a multi-machine system, communication overhead is typically high when sending data from master to another machine's processes. To exploit parallelism fully in a multi-machine system, one must deploy a feodal/fractal-like system [10]. In this deployment, although there is one central master, each other machine takes on more responsibility by controlling the information distribution to its processes itself. The central master communicates only with each machine's master process, not with every single process, where each machine deploys the exact same scheme described above, isolated from the rest of the machines. This approach mainly overcomes the overwhelming overhead due to machine communication. Note that in this project this scheme is not yet implemented and left as a future work.

2.4.2 Parallelism over Clusters

Since the likelihood calculations of data points are independent for each cluster, the operation can be parallelized. Sampling of cluster parameters and calculation of posteriors can also be parallelized. In order to achieve this efficiently, first processes should be given access to all the data, but, they should each be responsible for separate clusters. Clusters would be stored in a distributed array, though adding or deleting a cluster would be a costly operation. Or else, clusters could be transferred to the processes for each calculation which is obviously inefficient. Bottom line is that when K is small we don't expect much benefit from cluster parallelism due to communications overhead. In addition, achieving this type of parallelism along with the data point parallelism is even more challenging.

Additionally, split moves can also be parallelized since they are independent. For split moves, master machine should sent cluster parameters to each cluster and request splitted clusters if a split occurred. One can refer to the article [10] to see how this can be implemented together with data point parallelism.

3 Software Architecture

In this section, we will present DPMM.jl's architecture decisions and technical details of the types introduced. We will discuss distribution kernels, sufficient statistics, posterior calculations, cluster definitions, algorithms and the parallelism.

3.1 Distributions

Initially, we had relied on the Distributions.jl [6] for many of the fundamental distributions. However, slowness was inevitable in at least one of the following operations: log-likelihood calculations, sampling and initialization of the distributions. There are two fundamental reasons due to which this trade-off is inevitable for relying on a generic distributions package. First is the choice of parameterization. An example would the choice between using μ and Σ for a Gaussian distribution rather than μ and Σ^{-1}. In making these choices Distributions.jl typically resorts to the most canonical way which is not optimal for all of our purposes. For instance, having Σ^{-1} available is convenient for log-likelihood calculations and sampling but makes initialization slower. This is exactly what we need from a mixture distribution. On the other hand, we expect fast sampling from the prior distribution. Distributions.jl package does not meet with these computational expectations. Second reason of slowness is due to calculation of the normalization constant every time in the log-likelihood computation. In fact, for the mixture components, we do not need the normalization constant as we normalize over clusters before sampling anyway. Thus, we will use `logαpdf` keyword as an unnormalized log-likelihood function.

There is another reason for slowness which unrelated to trade-off mentioned above, but it has to do with coding style and readability. Julia has a very powerful and generic broadcast operation, and using broadcast is a very common practice in Julia community and Distributions.jl of course. However, it has small overhead compared to for loops in the simple operations. We defined our own kernels with fast for loops and used @simd (single instruction, multiple data) and @inbounds macros on them. Hence, we obtain fast kernels, thus fast algorithms.

Even though we prefer to write our own distributions, we agreed Distributions.jl format, and we define our distributions as a subtype to the `Distribution` abstract type so that we could benefit from predefined generic functions defined in Distributions.jl. Following sections, we will discuss the defined distributions and point out important aspects.

3.1.1 Normal-Wishart

Normal-Wishart distribution is used as a prior to Gaussian distributions with a mean and precision parameter, both unknown. Distributions.jl does not have Normal-Wishart since it is a combination of two distributions. So, we mostly take the implementation from the ConjugatePriors.jl package. We used $(\mu, \lambda, \Psi\ , \mathcal{W}^{-1}, \nu)$ parameterization for the Normal-Wishart distribution. The reason we preferred $\mathcal{W}^{-1}$ instead of $\mathcal{W}$ is that the posterior parameter calculation requires calculating $\mathcal{W}^{-1}$ which makes collapsed Gibbs algorithm particularly slow.

```
struct NormalWishart{T<:Real,S<:AbstractPDMat} <:
↪   ContinuousUnivariateDistribution
    μ::Vector{T} # prior mean
    λ::T  # This scales precision
    Ψ::S  # W^-1
    ν::T  # degrees of freedom
    function NormalWishart... # truncated for simplicity of the report
end
```

Code 1: Parameterization of Normal-Wishart Type

Note that we store Ψ as `AbstractPDMat` type from PDMats.jl package. `AbstractPDMat` is a type that keeps a positive definite matrix along with its Cholesky decomposition. PDMats.jl defines fast inverse and whitening transforms for `AbstractPDMat` and this makes the below sampling function efficient. For the prior distributions, we prefer to return the mixture distribution instead of the parameters which makes rest of the code simpler.

```
function rand(niw::NormalWishart{T,<:Any}) where T
    J   = PDMat(randWishart(inv(niw.Ψ), niw.ν))
    μ   = randNormal(niw.μ, PDMat(J.chol * niw.λ))
    return MvNormalFast(μ, J)
end

function randWishart(S::AbstractPDMat{T}, df::Real) where T
    p = dim(S)
    A = zeros(T,p,p)
    _wishart_genA!(GLOBAL_RNG,p, df,A)
    unwhiten!(S, A)
    A .= A * A'
end
```

Code 2: Normal-Wishart Sampler

3.1.2 Multivariate Gaussian

We define faster multivariate normal distribution for fast log-likelihood calculations. We used mean (μ) and precision (J) parameterization. We calculate normalization constant `c0` at the beginning. Although it is not immediately necessary, it takes small time and makes the distribution suitable for other uses. Similarly, storing J as AbstractPDMat is not required for log-likelihood calculations, however it is required for sampling. So, we decided to keep in this form.

```julia
struct MvNormalFast{T<:Real,Prec<:AbstractPDMat,Mean<:AbstractVector} <:
 ↪  AbstractMvNormal
    μ::Mean
    J::Prec
    c0::T
end
```

Code 3: Parameterization of Gaussian Distribution

We defined a log-likelihood kernel using fast and vectorized for loops by the help of @simd macro. @simd hints compiler that the for loop is vectorizable. We also tried @fastmath macro which changes mathematical operations with the ones which are faster but are not compatible with IEEE floating point arithmetic standards. We couldn't see a significant change in experimental results, so we discarded it.

```julia
function logαpdf(d::MvNormalFast{T}, x::AbstractVector{T}) where T
    D = length(x)
    μ = d.μ
    J = d.J.mat
    s = zero(T)

    y = Vector{T}(undef,D)
    @simd for i=1:D
        @inbounds y[i] = x[i]-μ[i]
    end

   @simd for i=1:D
        for j=1:D
            @inbounds s += y[i] * y[j] * J[i, j]
        end
    end

    return d.c0 - s/T(2)
end
```

Code 4: Fast Log-Likelihood Kernel for Gaussian Distribution

3.1.3 Dirichlet

Dirichlet distribution is used as a prior to multinomial distributions with an unknown probability vector. Distributions.jl has `Dirichlet` and `DirichletCanon` distributions where the former calculates the normalization constant at the initialization, and the latter does not. So, `DirichletCanon` is more suitable for our purposes as we don't need the normalization constant. However, to comply with our architecture decisions, we require from the prior distributions to return a new distribution. So, we redefined `DirichletCanon` as `DirichletFast` with small modifications in the sampling function to make it return the expected distribution in a fast manner:

```julia
struct DirichletFast{T<:Real} <:  ContinuousMultivariateDistribution
    α::Vector{T}
end

function _rand!(rng::Random.MersenneTwister, d::DirichletFast{T},
 ↪  x::AbstractVector{<:Real}) where T
    s = T(0)
    n = length(d)
    α = d.α
    @simd for i=1:n
        @inbounds s += (x[i] = rand(rng,Gamma(α[i])))
    end
    @simd for i=1:n
        @inbounds x[i] = log(x[i]) - s
    end
    MultinomialFast(x)
end
```

Code 5: Dirichlet Distribution and Sampler

3.1.4 Multinomial

We need a generic Multinomial distribution with undetermined k (trial count) because we want to assign a log-likelihood for every data point which has a different trial count. There is no such predefined Multinomial distribution in the Distributions.jl. Thus, we defined our own `MultinomialFast` distribution. Instead of the probability vector, we store the logarithm of it which facilitates a faster log-likelihood computation.

```julia
struct MultinomialFast{T<:Real} <: DiscreteMultivariateDistribution
    logp::Vector{T}
end

@inline function logαpdf(d::MultinomialFast{T}, x::AbstractVector) where
 ↪  T<:Real
    logp = d.logp
    s = T(0)
    D = length(d)
    @fastmath @simd for i=1:D
        @inbounds s += logp[i]*T(x[i])
    end
    return s
end
```

Code 6: Multinomial Distribution and Unnormalized Log-Likelihood Function

Most of the real applications, e.g. document classification, have sparse data matrix. In those cases, log-likelihood can be calculated even faster. To exploit this property, we utilized multiple-dispatch and overloaded `logαpdf` for sparse vectors:

```julia
function logαpdf(d::MultinomialFast{T}, x::DPSparseVector) where T<:Real
    logp  = d.logp
    nzval = nonzeros(x)
    s     = T(0)
    @fastmath @simd for l in enumerate(x.nzind)
        @inbounds s += logp[last(l)]*nzval[first(l)]
    end
    return s
end
```

Code 7: Unnormalized Log-Likelihood Function For Sparse Vector Type

3.2 Posteriors & Sufficient Statistics

Algorithms require a Bayesian update functionality for the calculation of posterior distributions from the prior. To this end, we need to define sufficient statistics types and the Bayesian updates to the priors with fast implementations. Distributions.jl `SufficientStats` provides abstract types with predefined sufficient statistics for Gaussian, Multinomial and many other distribution types. However, these statistics are not implemented with speed as the primary concern. Thus, we defined `DPGMMStats` for Normal Wishart prior and `DPMNMMStats` for Dirichlet prior as a subtype of `SufficientStats`.

We expect from a prior type and the related subtype of `SufficientStats` to implement certain methods in order to be compatible with our package. One can refer to function documentation [2] to see the complete list of required methods to be implemented. If the user needs to implement a non-conjugate prior, they should carefully design the `posterior` function and a sampler for the posterior. An example (truncated) implementation for `DPMNMMStats` and `DirichletFast` is given below for a reference.

```julia
@inline stattype(::DirichletFast{T}) where T = DPMNMMStats{T}
struct DPMNMMStats{T<:Real} <: SufficientStats
    s::Vector{Int}
    n::Int
end
@inline suffstats(m::DirichletFast{T}) where T<:Real =
    DPMNMMStats{T}(zeros(Int,length(m),0))
```

[2] `https://ekinakyurek.github.io/DPMM.jl/latest/`

```julia
@inline suffstats(m::DirichletFast{T},X::AbstractMatrix{Int}) where T<:Real =
    DPMNMMStats{T}(sumcol(X),size(X,2))
@inline suffstats(m::DirichletFast{T},x::AbstractVector{Int}) where T<:Real =
    DPMNMMStats{T}(x,1)

@inline function updatestats(m::DPMNMMStats{T},x::AbstractVector{Int}) where
↪   T<:Real
    DPMNMMStats{T}(add!(m.s,x),m.n+1)
end

@inline function downdatestats(m::DPMNMMStats{T}, x::AbstractVector{Int}) where
↪   T<:Real
    DPMNMMStats{T}(substract!(m.s,x),m.n-1)
end

@inline _posterior(m::DirichletFast{V}, T::DPMNMMStats{V}) where V<:Real = m.α
↪   + T.s

@inline posterior(m::DirichletFast{V}, T::DPMNMMStats{V}) where V<:Real =
↪   T.n!=0 ? DirichletFast{V}(_posterior(m,T)) : m

@inline posterior_predictive(m::DirichletFast{V},T::DPMNMMStats{V}) where
↪   V<:Real = T.n != 0 ? DirichletMultPredictive{V}(_posterior(m,T)) :
↪   posterior_predictive(m)
 ... # truncated
```

Code 8: `DPMNMMStats` and `DirichletFast` Bayesian Updates

3.3 Data Storage

Throughout the report, we assume $\mathcal{X}$ to be a matrix where each column is i.i.d multidimensional data. In the common scenarios, the matrix can be in the form of three different concrete types:

- `Matrix{<:AbstractFloat}`: a data generated from a continuous mixture, e.g Gaussian Mixtures.
- `Matrix{<:Integer}`: a data generated from a discrete mixture, e.g Multinomial Mixtures.
- `DPSparseMatrix{<:Integer}`: a sparse data assumed to observed from discrete sparse mixtures. e.g. bag of words for a document collection

3.3.1 DPSparseMatrix

It is our redefinition of Julia's default `SparseMatrixCSC` which has slow column indexing (i.e. `X[:,i]`) and gathering (i.e. `X[:,inds...]`). Column indexing and gathering are critical operations for Gibbs samplers, thus `DPSparseMatrix` is designed to be convenient for these operations.

In `DPSparseMatrix`, see 9, we store matrices as an array of columns which are sparse vectors. Therefore, we are able to get `views` (reading without copying) of the columns by indexing a flat array. After getting the pointers, we can form another `DPSparseMatrix` which points to same data without any other computation. Since we never mutate $\mathcal{X}$ in the Gibbs samplers, pointing same data in many sparse matrices does not constitute an issue. We have written a convenient constructor for `DPSparseMatrix` which takes in Julia's default `SparseMatrixCSC` and returns a `DPSparseMatrix` out of it.

```
struct DPSparseMatrix{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
    m::Int                                # Number of rows
    n::Int                                # Number of columns
    data::Vector{DPSparseVector{Tv,Ti}}       # Stored columns as sparse vector
end

function DPSparseMatrix(X::SparseMatrixCSC{Tv,Ti}) where {Tv,Ti}
    DPSparseMatrix{Tv,Ti}(X.m,X.n,map(i->DPSparseVector(X[:,i]),1:size(X,2)))
end

@inline Base.view(X::DPSparseMatrix, ::Colon, ind::Integer) = X.data[:,ind]
@inline Base.view(X::DPSparseMatrix, ::Colon, inds::Vector{<:Integer}) =
 ↪  DPSparseMatrix(X.m,length(inds),X.data[inds])
```

Code 9: DPSparseMatrix Type and Column Indexing

3.3.2 DPSparseVector

`SparseVector` is our redefinition of Julia's default `SparseVector` which has unwanted behaviour on the summation operation. In Julia, `SparseVector` + `Vector` yields to a `SparseVector` by default. We do not want this behavior as it makes sufficient statistics sparse, which is not, indeed, the case in practice. As a result, it slows down the code due to slowness of `setindex` operation in `SparseVector`'s. In our redefinition, `SparseVector` + `Vector` yields to a `Vector` again. In addition, we defined a fast inplace summation operation, which we used for the summation of more than one sparse columns to get sufficient statistics. Since, we frequently sum the columns in the algorithms, this is implemented in `sumcol` function in Code 10.

```
struct DPSparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
    n::Int                # Length of the sparse vector
    nzind::Vector{Ti}     # Indices of stored values
    nzval::Vector{Tv}     # Stored values, typically nonzeros
    function DPSparseVector ... # truncated for cleanliness of the report
end

function sumcol(X::DPSparseMatrix{Tv,<:Any}) where Tv
```

```julia
    y = zeros(Tv,X.m)
    for i=1:X.n
        @inbounds add!(y,X[:,i])
    end
    return y
end

@inline sumcol(X) = vec(sum(X,dims=2))
@inline sum(x::DPSparseVector) = sum(x.nzval)
@inline +(x::DPSparseVector, y::DPSparseVector) = add!(Vector(x),y)
@inline -(x::DPSparseVector, y::DPSparseVector) = substract!(Vector(x),y)

function add!(x::AbstractVector, y::DPSparseVector)
    for (i,index) in enumerate(y.nzind)
        @inbounds x[index] += y.nzval[i]
    end
    return x
end
```

Code 10: DPSparseVector Type and Fast Summations

3.4 Models

A model is a Dirichlet Process defined by a prior and an α value and it is the intermediate types between prior and algorithm. This allows users to use the same prior but define a customized initialization, different sufficient statistics and different posterior calculations. Every model has to be a subtype of `AbstractDPModel`. This framework allows models to work without defining any posterior or sufficient functions if the prior already has one. `AbstractDPModel` definition. A sample `DPGMM` model is shown in the Code 11:

```julia
abstract type AbstractDPModel{T,D} end
@inline length(::AbstractDPModel{<:Any,D}) where D = D
@inline stattype(m::AbstractDPModel)  = stattype(prior(m))
@inline suffstats(m::AbstractDPModel) = suffstats(prior(m))
@inline suffstats(m::AbstractDPModel, X) = suffstats(prior(m),X)
@inline posterior(m::AbstractDPModel) =  prior(m)
@inline posterior(m::AbstractDPModel, T::SufficientStats)  =
↪  posterior(prior(m),T)
@inline posterior_predictive(m::AbstractDPModel) =
↪  posterior_predictive(prior(m))
@inline posterior_predictive(m::AbstractDPModel,  T::SufficientStats) =
↪  posterior_predictive(prior(m),T)
@inline _posterior(m::AbstractDPModel, T::SufficientStats) =
↪  _posterior(prior(m),T)
```

Code 11: `AbstractDPModel` with Generic Functions

```
struct DPGMM{T<:Real,D} <: AbstractDPModel{T,D}
    θprior::NormalWishart{T}
    α::T
end
@inline prior(m::DPGMM) = m.θprior
function DPGMM(X::AbstractMatrix{T}; α::Real=1) where T<:Real
    DPGMM{T}(T(α), vec(mean(X,dims=2)),(X*X')/size(X,2))
end
@inline DPGMM{T,D}(α::Real) where {T<:Real,D} =
    DPGMM{T,dim}(NormalWishart{T}(D),T(α))
@inline DPGMM{T}(α::Real, μ0::AbstractVector{T}) where T<:Real =
    DPGMM{T,length(μ0)}(NormalWishart{T}(μ0),T(α))
@inline DPGMM{T}(α::Real, μ0::AbstractVector{T}, Σ0::AbstractMatrix{T}) where
↪   T<:Real =
    DPGMM{T,length(μ0)}(NormalWishart{T}(μ0,Σ0), T(α))
```

Code 12: `DPGMM` Model

3.5 Clusters

Subtypes of `AbstractCluster` represents the clusters in DPMM, and they store what is needed by the specific algorithm that they belong to. For instance, a `CollapsedCluster` stores a predictive distribution, where a `DirectCluster` stores the sampled distribution. Moreover, a `SplitMergeCluster` stores information related to right and left subclusters. The definitions of the clusters are listed in the Code 13 for comparison.

```
struct CollapsedCluster...
    n::Int
    predictive::Pred
    prior::Prior
end
```

```
struct DirectCluster...
    n::Int
    sampled::Pred
    prior::Prior
end
```

```
struct SplitMergeCluster{Pred<:Distribution, Post<:Distribution ...
    n::Int; nr::Int; nl::Int
    sampled::Pred; right::Pred; left::Pred
    s::SufficientStats
    post::Post; rightpost::Post; leftpost::Post
    prior::Prior
    llhs::NTuple{3,Float64}
    llh_hist::NTuple{4,Float64}
end
```

Code 13: Cluster Type Definitions

Every cluster needs some common functionality such as a constructor, a proportional likelihood function for a new data point and a population function that returns the number of data points. To label and access them, we store the clusters in a dictionary object, `Dict{Int,<:AbstractCluster}`. It also facilitates addition and deletion of clusters.

Therefore, cluster types need to implement an initialization function which returns a `Dict{Int,<:AbstractCluster}` structure that accommodates initial clusters.

We've also defined custom functions named as $+$ and $-$ for data addition and removal in some of the cluster types e.g. `CollapsedCluster`. An example implementation for a `CollapsedCluster` is shown in Code 14:

```julia
@inline population(m::CollapsedCluster) = m.n

@inline CollapsedCluster(m::AbstractDPModel) = CollapsedCluster(m,
 ↪  suffstats(m))

@inline CollapsedCluster(m::AbstractDPModel,X::AbstractArray) =
    CollapsedCluster(m, suffstats(m,X))

@inline CollapsedCluster(m::AbstractDPModel, s::SufficientStats) =
    CollapsedCluster(s.n, posterior_predictive(m,s), m.θprior)

@inline CollapsedCluster(m::AbstractDPModel,new::Val{true}) =
    CollapsedCluster(floor(Int,m.α),posterior_predictive(m),m.θprior)

@inline -(c::CollapsedCluster{V,P},x::AbstractVector) where
 ↪  {V<:Distribution,P<:Distribution} =
    CollapsedCluster{V,P}(c.n-1,
     ↪  downdate_predictive(c.prior,c.predictive,x,c.n), c.prior)

@inline +(c::CollapsedCluster{V,P},x::AbstractVector) where
 ↪  {V<:Distribution,P<:Distribution} =
    CollapsedCluster{V,P}(c.n+1, update_predictive(c.prior,c.predictive,x,c.n),
     ↪  c.prior)

@inline logαpdf(m::CollapsedCluster,x)  = logαpdf(m.predictive,x)

CollapsedClusters(model::AbstractDPModel, X::AbstractMatrix,
 ↪  z::AbstractArray{Int}) =
    Dict((k,CollapsedCluster(model,X[:,findall(l->l==k,z)])) for k in
     ↪  unique(z))

SuffStats(model::AbstractDPModel, X::AbstractMatrix, z::AbstractArray{Int}) =
    Dict((k,suffstats(model,X[:,findall(l->l==k,z)])) for k in unique(z))

CollapsedClusters(model::AbstractDPModel, stats::Dict{Int,<:SufficientStats}) =
    Dict((k,CollapsedCluster(model,stats[k])) for k in keys(stats))
```

Code 14: `CollapsedCluster` Functionality

3.6 Algorithm Types & `fit` Interface

We created Algorithm types to specify configurations of the algorithms. One can specify initial number of clusters, parallel vs sequential versions when constructing the algorithm. Every algorithm is a subtype of `DPMMAlgorithm{P}` where `P` stands for parallel. We also implemented a `fit` function which can work with any `DPMMAlgorithm{P}` subtype. `fit` automatically setups the parallel processors, initializes clusters and runs the algorithm. In this design, algorithm types need to implement a constructor with data and optional keyword arguments, a random label generator for the data, a cluster initialization function, an empty cluster initialization function and a `run!` function that takes the output of the mentioned functions as argument. In Code 15, we present interface definitions for Collapsed Gibbs sampler as an example of adding a new sampler to the package.

```julia
struct CollapsedAlgorithm{P,Q} <: DPMMAlgorithm{P}
    model::AbstractDPModel
    ninit::Int
end

run!(algo::CollapsedAlgorithm{false,false},X,args...;o...) = # collapsed
↪   algorithm
    collapsed_gibbs!(algo.model,X,args...;o...)
run!(algo::CollapsedAlgorithm{false,true},X,args...;o...) =  # quasi collapsed
↪   algorithm
    quasi_collapsed_gibbs!(algo.model,X,args...;o...)
run!(algo::CollapsedAlgorithm{true,false},X,args...;o...) =
    error("Collapsed Gibbs Sampler is not parallelizable!")
run!(algo::CollapsedAlgorithm{true,true},X,args...;o...) =  # parallel
↪   quasi-collapsed algorithm
    quasi_collapsed_gibbs_parallel!(algo.model,X,args...;o...)

random_labels(X,algo::CollapsedAlgorithm) = rand(1:algo.ninit,size(X,2))
create_clusters(X,algo::CollapsedAlgorithm,labels) =
↪   CollapsedClusters(algo.model,X,labels)
empty_cluster(algo::CollapsedAlgorithm) =
↪   CollapsedCluster(algo.model,Val(true))
```

Code 15: Collapsed Gibbs Algorithm Definitions

Note that the `run!` functions call the implementation functions for aesthetic reasons. We could inline those functions to the body. One can refer to the body functions at the repository. After defining an algorithm, `fit` interface can be used directly, examples that are showing the interface listed in Code 16.

```julia
labels = fit(X; algorithm=DirectAlgorithm, quasi=true) # quasi direct gibbs
↪   algorithm
labels = fit(X; algorithm=DirectAlgorithm, quasi=true, ncpu=4) # parallel
labels = fit(X; algorithm=SplitMergeAlgorithm) # split-merge
labels = fit(X; algorithm=SplitMergeAlgorithm, ncpu=4) # parallel
```
```

**Code 16:** fit

## 3.7 Parallelism

We explained parallelism over data points in Section 2.4. Here we will show how it is implemented using Julia's `Distributed` and `SharedArrays` packages only. Distributing the data is done automatically in the `initialize_clusters` for any `DPMMAlgorithm`, provided in Code 17. `initialize_clusters` initializes random `labels` in a `SharedArray` if the algorithm is parallel. `SharedArray`'s are convenient as they are readable and writable by all processes. We index relevant data for each worker by obtaining local indices of that worker. Then, we easily sent the data to the `Main` module of the processor by using `@spawnat` macro.

```
function initialize_clusters...
...
@sync for (i,p) in enumerate(procs(labels))
    xworker = X[:,range_1dim(labels,i)]
    @spawnat(p, Core.eval(Main, Expr(:(=), :_X, xworker)))
end
...
```

Code 17: Distributing The Data

In Code 18, we used Distributed library's `remotecall_fetch` method to distribute sampling work to parallel workers. In all algorithms, we call a remote sampling kernel with the updated clusters and `labels` array, and we get the new sufficient statistics in return. We then gather those sufficent statistics together in the main process. Update of the `labels` happens in the worker process as it is a `SharedArray`.

```
...
stats = Dict{Int,Tuple{<:SufficientStats, <:SufficientStats}}[]
@sync begin
   for p in procs(labels)
       @async push!(stats,remotecall_fetch(splitmerge_parallel!,p,labels,
                                                             clusters,logπs,logsπs))
   end
end
update_clusters!(model, clusters, gather_stats(stats))
....
```

Code 18: Distributing The Sampling Work in a Split-Merge Iteration
```

Cores x Machines	1	2	4	8
Collapsed Sampler	1049.37	-	-	-
Quasi-Collapsed Sampler	806.86	417.16	157.74	84.86
Direct-Sampler	113.67	51.02	39.49	21.76
Split-Merge Sampler	71.39	35.74	19.70	11.36

Table 1: Time (sec) to run 100 DP-GMM iterations for d=2, N=1M, K=6.

4 Results

In the following sections we discuss the performance and parallel efficiency of the implemented algorithms. We used `NormalWishart` priors for Gaussian experiments, and `Dirichlet` prior for Multinomial experiments. We set α to 1 and initial number of clusters to 1 for all the experiments. Prior of the Split-Merge algorithm is different at initialization then other samplers. In Split-Merge, we initialized `NormalWishart` with an identity covariance matrix and `Dirichlet` with a `ones` vector which are weak priors. In the Collapsed and Direct algorithms, we used stronger priors namely average covariance for `NormalWishart` and average count vector for `Dirichlet` which we called as strong priors. The reason is that the other algorithms are intrinsically inclined to increase the number of clusters if they have a stronger prior, while split-merge can do more split moves if the prior is weak to explain whole data.

4.1 Serial Comparisons

We evaluated Collapsed Sampler, Direct Sampler and Split-Merge sampler performances demonstrated in Tables 1 and 2. In the Gaussian mixture experiments, we generate 10^6 samples from an underlying Gaussian mixture model with K=6 clusters in d=2 dimensions, while we use d=100, N=1M, K=6 settings for Multinomial experiments to be consistent with previous work [10]. We did not include Quasi-Direct Sampler because it is almost the same with the Direct Sampler in our setup. We run the algorithms for 100 iterations.

By default, we set the initial number of clusters to 1 except for the Direct sampler. This is because Direct sampler doesn't converge to the correct number of clusters if starts with a single initial cluster. So, we time it by starting with correct number of random clusters. Split-Merge sampler is similar to Direct Sampler in terms of timing. Regardless the initial number of clusters, Split-Merge sampler converges to the correct number of clusters. As expected, Collapsed Sampler is the slowest sampler, nevertheless it converges the correct number of clusters robustly within the 100 iterations. Quasi-Collapsed sampler converges similarly to and is faster than Collapsed sampler. Overall, Split-Merge is significantly faster all other samplers, while maintaining an accurate convergence.

4.2 Parallel Comparisons

We compared parallel performances of Quasi-Collapsed, Direct and Split-Merge samplers in Tables 1 and 2. Because Quasi-Collapsed sampler is the slowest parallel sampler, its parallel utilization is the highest which has perfect scaling to 8 cores in the Gaussian

Cores x Machines	1	2	4	8
Collapsed Sampler	2817.99	-	-	-
Quasi-Collapsed Sampler	2344.90	1139.46	589.24	366.38
Direct-Sampler	78.71	37.63	20.51	15.72
Split-Merge Sampler	73.28	53.46	32.20	18.30

Table 2: Time (sec) to run 100 DP-MNMM iterations for d=100, N=1M, K=6.

experiments, 6.28x scaling with 8 cores in the Multinomial experiments. Note that some samplers have unintuitive results with multi-cores where the utilization is higher than the maximum i.e. Quasi-Collapsed becomes more than two times faster where the number of cores only doubles from 1 core to 2 cores. This is due to randomness in the sampling process which affects cluster creation or deletion time points. It is almost impossible in a parallel algorithm to arrange random numbers to be used for the same operations in the same order as they would have been in the serial version. Direct Sampler and Split-Merge samplers have better a parallel utilization in the Gaussian data because the Gaussian posterior updates are slower than the Multinomial's.

4.3 Benchmark Results

We benchmark our split-merge implementation with the other C++ and Julia implementations respectively: [10] and [3]. We run experiments with synthetic Gaussian and Multinomial mixtures on Intel Xeon E-5270v2 processor which is similar to the system described in [10], Intel Xeon E-5270v3, with a small difference in multi-core performance in our disadvantage. Although this might have slightly negatively affected our parallel efficiency, we did not have access to an identical computational resource.

In the Gaussian mixture experiments, we generate 10^6 samples from an underlying Gaussian mixture model with K=6 clusters in d=2 dimensions. Our implementation successfully clusters the data into 6 clusters. In terms of runtime, our implementation is at the same speed as [3] which is at least 15.4 times faster than the Julia_bnp implementation [10] as in Table 3. There are a set of reasons underlying this improvement in performance. First, to expedite both the sampling and likelihood calculations, we implemented our own kernels. Secondly, in Julia_bnp implementation, things like cluster parameters, sufficient statistics etc. are stored in distributed arrays. This has several downsides such as array reshape overhead when a cluster gets added or removed. Conversely, we store all cluster related parameters/statistics in structs that are accessed through a dictionary which has no reshaping overhead. However, we'll see that this becomes slightly costly with increasing number of cores as the dictionary should be passed around after cluster updates. Both of these improvements are described in more detail in Section 3.

In terms of parallel efficiency, using 8 cores we achieved 6.3 times faster runtime. This is because our implementation is already significantly faster than [10], thus, suffers proportionally more from communication overhead as the number of cores increases. Moreover, the dictionary which accommodates cluster information as structs, should be passed around other worker processes.

Runtime benchmarking results for the multinomial experiments are presented in the bottom section of Table 3. We used 6 clusters where dimension is set to 2 and generated a million points. As a result, our implementation is 1.8 and 3.2 times faster than C++ [3] and

Julia_bnp [10] implementations, respectively. Considering parallel efficiency, DPMM.jl is 4 times faster with 8 cores, however, is still faster than other implementations with all number of cores.

We performed another set of experiments where we altered the number of clusters and dimension as in Tables 4 and 5. For the Gaussian experiment with 8 cores, we also increased the dimension to 30 and observed that DPMM.jl is 6.5 and 3.3 times faster than [10] and [3] respectively. Moreover, we set the actual number of clusters to be 60 instead of 6 and saw that our implementation is 2 and 3 times faster than [10] and [3], respectively.

	Cores x Machines	1	2	4	8	8 x 2	8 x 3	8 x 4
	C++	76.94	40.57	22.23	13.01	-	-	-
Gaussian	julia_bnp	1101.97	572.50	345.58	172.30	107.58	80.10	63.55
	DPMM.jl	71.39	35.74	19.70	11.36	-	-	-
	C++	134.25	77.55	40.97	23.60	-	-	-
Multinomial	julia_bnp	234.40	136.43	87.34	55.10	34.58	31.50	32.61
	DPMM.jl	**73.28**	**53.46**	**32.20**	**18.30**	-	-	-

Table 3: Time (sec) to run 100 iterations for DP-GMM (d=2, N=1M, K=6) and DP-MNMM (d=100, N=1M, K=6)

Cores x Machines	8	8 x 2	8 x 3	8 x 4
C++	798.94	-	-	-
julia_bnp	398.67	218.42	146.71	124.55
DPMM.jl	**78.57**	-	-	-

Table 4: Time(sec) to run 100 DP-GMM iterations of d = 30, N = 1M, K = 6.

Cores x Machines	8	8 x 2	8 x 3	8 x 4
C++	145.50	-	-	-
julia_bnp	241.31	142.89	109.95	98.95
DPMM.jl	**75.56**	-	-	-

Table 5: Time(sec) to run 100 DP-MNMM iterations of d = 100, N = 1M, K = 60.

4.4 New York Times Results

We test our implementation on New York Times articles dataset [1]. It is a bag-of-word dataset of 300,000 documents and 102,660 dimension dictionary. We used Dirichlet-Multinomial prior and sparse data representation. We ran Split-Merge algorithm 50 iterations which resulted in 79 clusters. We created word clouds from nine clusters to visualize results in Figure 2.

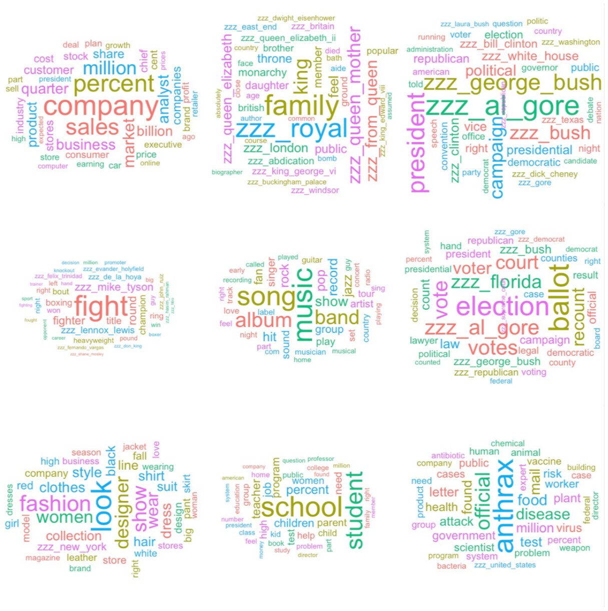


Figure 2: Word Clouds from NYTimes Clusters

5 Conclusion

We successfully implemented sequential and parallel versions of Collapsed Gibbs Sampler, Direct Gibbs Sampler and Split-Merge Sampler in Julia, in a fast and accurate manner. The bodies of the algorithms are implemented in such a way that they bear the simplicity of the pseudocodes presented here in the sections above. This is inline with our design goal of providing an easy-to-understand implementation. Therefore, we are not investigating the actual implementations of the algorithms in detail here, as one can always refer to the code repository and functional documentation for detailed information.

The results shows that DPMM.jl outperforms other existing packages in single core and multi core performances for both Gaussian and Multinomial data. It doesn't yet provide multi-machine parallelism which we leave as a future work.

References

[1] K. Bache and M. Lichman, Uci machine learning repository, 2013.

[2] D. Blackwell, J. B. MacQueen, et al., Ferguson distributions via pólya urn schemes, The annals of statistics, 1 (1973), pp. 353–355.

[3] J. Chang and J. W. Fisher, III, Parallel sampling of dp mixture models using sub-clusters splits, in Proceedings of the Neural Information Process Systems (NIPS), Nov 2013.

[4] B. Fang, Introduction to dirichlet process, 2016.

[5] H. Kamper, Gibbs sampling for fitting finite and infinite gaussian mixture models, 2013.

[6] D. Lin, J. M. White, S. Byrne, D. Bates, A. Noack, J. Pearson, A. Arslan, K. Squire, D. Anthoff, T. Papamarkou, M. Besançon, J. Drugowitsch, M. Schauer, and other contributors, JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions, May 2019.

[7] K. P. Murphy, Conjugate bayesian analysis of the gaussian distribution, def, 1 (2007), p. 16.

[8] M. Seeger, Low rank updates for the cholesky decomposition, (2004).

[9] J. Wishart, The generalised product moment distribution in samples from a normal multivariate population, Biometrika, 20 (1928), pp. 32–52.

[10] A. Yu, Parallel and distributed MCMC inference using Julia, PhD thesis, Massachusetts Institute of Technology, 2016.