An Efficient Algorithm for Multiview-Clustering

Bayesian Nonparametric Inference via Pitman-Yor Process

Jumil Reyes

December 3, 2025

Contents

Chapter 1

Introduction: Inference via Gibbs Sampling

This document details the C++ implementation of the **Multi-View Hierarchical Pitman-Yor Process (H-PYP)** model. This Bayesian nonparametric mixture model uses the **Chinese Restaurant Franchise (CRF)** to create a flexible, two-level clustering structure capable of handling the sharing of latent components across multiple data views.

1.1 Why Use the Gibbs Sampler?

The primary challenge in inference for complex Bayesian models like the H-PYP is that the joint posterior distribution of all latent variables (cluster assignments, means, and hyperparameters) is analytically intractable. We cannot calculate it directly.

The **Gibbs Sampler**, a specific type of **Markov Chain Monte Carlo (MCMC)** method, is the ideal solution because it allows us to sample from this complex joint distribution indirectly.

1.1.1 The Power of Conditional Distributions

The Gibbs Sampler works by exploiting the principle that, while the joint posterior $P(\Theta \mid Y)$ is unknown, the **full conditional distributions** $P(\theta_i \mid \Theta_{-i}, Y)$ (where Θ_{-i} represents all other parameters) are often simple and known (or easy to sample from).

In our H-PYP model, this translates to sequential sampling across all layers:

1. **Assignment Sampling:** We iteratively sample the assignment of customer i to a table t^*, conditioning on the assignments of all other customers and the current hyperparameter values. This process is highly efficient due to the conjugacy inherent in the Pitman-Yor Process structure.

2. **Hyperparameter Updates:** Parameters like the concentration (α), discount (σ), and noise variance (τ_v) are updated using **Metropolis-Hastings (MH)** steps within the Gibbs loop, as their conditional posteriors are non-conjugate. This hybrid approach is common and necessary for adaptive models.

1.2 Markov Chain Convergence Properties

For the MCMC chain to produce valid estimates of the posterior distribution, it must be **ergodic**. This is achieved if the chain possesses the following key properties:

- **Irreducibility:** The chain must be able to move from any possible cluster configuration or parameter setting to any other in a finite number of steps. This is guaranteed by the sequential re-sampling process (which allows for implicit split/merge moves) and the broad proposal distributions used in the MH steps.

- **Aperiodicity:** The chain must not be stuck in a deterministic cycle. This is ensured by the inherent randomness of the sampling process and the probabilistic nature of the MH acceptance/rejection step.
- **Harris Recurrence:** This strong condition ensures the convergence of the sample averages (e.g., mean cluster assignments) to the true theoretical posterior means. Given the continuity of the model (Gaussian likelihoods) and the satisfied irreducibility/aperiodicity, the chain is assumed to be Harris-recurrent.

These properties collectively guarantee that the chain will eventually converge to a unique stationary distribution, which is our target posterior distribution.

1.3 Workflow and Code Structure

The parameter τ_v (tau) represents the **residual noise variance** for view v. It quantifies the spread of the data points around the cluster mean (μ_{vk}). Since the data is assumed to be Gaussian, τ_v is crucial for determining the width of the likelihood function for each view. It is learned from the data using the MH procedure, typically assuming an Inverse-Gamma prior.

The code is modularized to reflect this inference flow:

- `multiview_state.cpp`: Defines the **Global State** and all variables (assignments, counts, sufficient statistics) required to define the conditional distributions.
- `multiview_utils.cpp`: Implements the core logic to calculate the conditional probabilities for assignment sampling and manages state transitions (e.g., customer removal and addition).
- `multiview_hyper.cpp`: Contains the MH machinery for updating the non-conjugate hyperparameters (α, σ, τ).

By continuously cycling through these conditional updates, the Markov Chain eventually converges to the true joint posterior distribution, allowing us to estimate the final cluster assignments.

Chapter 2

Global State Definition (`multiview_state.cpp`)

This file acts as the central storage unit for the sampler. It defines the global variables that represent the current state of the Markov Chain. Using global variables in this specific context simplifies the function signatures across the Rcpp interface and avoids passing large state objects by reference repeatedly.

2.1 Dimensions and Data

The basic dimensions and observation data are stored statically:

- `int n, d`: The number of observations (customers) and the number of views, respectively.
- `vector<vector<double»  y`: The raw data, organized as `y[view_index][observation_index]`.

2.2 Global Partitioning State (CRF Level 1)

These variables track the assignment of customers to tables (the global clustering structure):

- `int T`: The current number of active tables (clusters).
- `vector<int> table_of`: An array of size n, where `table_of[i]` is the table index assigned to customer i.
- `vector<int> n_t`: Stores the number of customers currently sitting at each table t.
- `vector<vector<int» customers_at_table`: An adjacency list for efficient retrieval of all customers belonging to a specific table. This is crucial for split-merge moves or efficient removal operations.
- `alpha_global, sigma_global`: The hyperparameters governing the global Pitman-Yor process.

2.3 View-Specific State (CRF Level 2)

These variables track the assignment of tables to dishes (view-specific clustering) and view-specific parameters.

2.3.1 Table-to-Dish Mapping

- `vector<vector<int»` `dish_of`: A matrix-like structure where `dish_of[v][t]` indicates the dish (local cluster) chosen by table t in view v.

2.3.2 The `ViewState` Structure

The vector `views` contains objects of type `ViewState`, which encapsulate all parameters unique to a specific view v:

- **Parameters:** α_v, σ_v (local PYP params) and τ_v (noise variance).
- **Counts:**
  - `l_vk`: The number of *tables* assigned to dish k.
  - `n_vk`: The number of *customers* assigned to dish k (indirectly via tables).
- **Sufficient Statistics:** `sum_y` and `sum_y2` for each dish, enabling $O(1)$ likelihood updates.

2.4 MCMC Storage

To analyze the posterior distribution after the simulation, samples are stored in memory:

- `saved_table_of`: Stores the vector `table_of` at each thinning interval.
- `saved_dish_of`: Stores the assignment of tables to dishes.
- `saved_loglik`: Tracks the log-likelihood for convergence diagnostics.

Chapter 3

Hyperparameter Inference (`multiview_hyper.cpp`)

This module is critical for the adaptivity of the model. It implements the Bayesian inference for the hyperparameters governing the Pitman-Yor Processes (PYP) and the observation noise. Since the posterior distributions for these parameters are not available in closed form, we employ **Metropolis-Hastings (MH)** steps within the Gibbs Sampler.

The parameters updated in this module are:

- $\alpha_{\text{glob}}, \sigma_{\text{glob}}$: Concentration and discount for the global PYP (grouping customers into tables).
- α_v, σ_v: Concentration and discount for each view-specific PYP (grouping tables into dishes).
- τ_v: The residual noise variance for each view (assuming Gaussian data).

3.1 Observation Noise Variance (τ_v)

The parameter τ_v represents the variance of the Gaussian likelihood for view v.

$$y_{vi} \mid k, \tau_v \sim \mathcal{N}(\mu_{vk}, \tau_v)$$

3.1.1 Prior and Posterior

We assume an Inverse-Gamma prior for computational convenience and flexibility:

$$\tau_v \sim \text{Inv-Gamma}(a_\tau, b_\tau)$$

The log-prior density is implemented as:

$$\log \pi(\tau_v) = a_\tau \log(b_\tau) - \log \Gamma(a_\tau) - (a_\tau + 1) \log(\tau_v) - \frac{b_\tau}{\tau_v}$$

The log-likelihood, marginalized over the cluster means (or using plug-in estimates), depends on the Sum of Squared Errors (SSE):

$$\log \mathcal{L}(\mathbf{y}_v \mid \tau_v) = -\frac{N}{2} \log(2\pi\tau_v) - \frac{1}{2\tau_v} \sum_k \sum_{i \in C_k} (y_{vi} - \bar{y}_{vk})^2$$

Implementation Note: In `log_posterior_given_tau`, the SSE is calculated efficiently using sufficient statistics ($\sum y, \sum y^2$) stored in `ViewState`:

$$\text{SSE}_k = \sum_{i \in C_k} y_i^2 - \frac{(\sum y_i)^2}{n_k}$$

This avoids iterating over raw data points, ensuring $O(K)$ complexity instead of $O(N)$.

3.1.2 Metropolis-Hastings Update Algorithm

Since the posterior is non-conjugate due to the hierarchical structure, we use a Random Walk MH step on the log-scale to ensure positivity.

1. **Proposal:** We propose τ' from a log-normal distribution centered at the current τ:

$$\log(\tau') = \log(\tau) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \delta_\tau^2)$$

2. **Acceptance Ratio:** The acceptance probability $A(\tau' \mid \tau)$ involves the posterior ratio and the proposal ratio (Jacobian):

$$\log A = \log \frac{P(\tau' \mid \text{data})}{P(\tau \mid \text{data})} + \log \frac{q(\tau \mid \tau')}{q(\tau' \mid \tau)}$$

For a log-normal random walk, the proposal ratio (Jacobian) is:

$$\log \frac{q(\tau \mid \tau')}{q(\tau' \mid \tau)} = \log(\tau') - \log(\tau)$$

3. **Decision:** Accept τ' if $\log(U) < \log A$, where $U \sim \mathcal{U}(0, 1)$.

3.2 Pitman-Yor Process Parameters (α, σ)

The core of the clustering behavior is governed by the concentration α and discount σ. These are updated separately for the global level and for each view.

3.2.1 The Exchangeable Partition Probability Function (EPPF)

The likelihood of the partition given α, σ is defined by the EPPF. The code implements the log-EPPF to avoid numerical underflow.

$$\log P(\Pi \mid \alpha, \sigma) = \left[\sum_{j=0}^{K-1} \log(\alpha + j\sigma)\right] - \left[\sum_{i=1}^{N-1} \log(\alpha + i)\right] + \left[\sum_{k=1}^{K} \sum_{m=1}^{n_k - 1} \log(m - \sigma)\right] \tag{3.1}$$

Hierarchical Distinction in Implementation

A critical aspect of the implementation is the distinction between the two levels of the hierarchy:

- **Global Level (`log_global_EPPF`):**
  - **Items:** Customers ($i = 1 \ldots N$).
  - **Clusters:** Tables ($t = 1 \ldots T$).
  - **Counts:** Uses `n_t` (number of customers per table).
- **View Level (`log_EPPF`):**
  - **Items:** Tables ($t = 1 \ldots T$).
  - **Clusters:** Dishes ($k = 1 \ldots K_v$).
  - **Counts:** Uses `l_vk` (number of tables eating dish k).
  - *Crucial Logic:* Using customer counts (`n_vk`) here would be theoretically incorrect for the CRF model. The code correctly uses `l_vk`.

3.2.2 Update Logic for α (Concentration)

- **Constraints:** $\alpha > 0$ (assuming $\sigma \in [0, 1)$).
- **Prior:** Gamma prior, $\alpha \sim \text{Gamma}(a_\alpha, b_\alpha)$.
- **Algorithm:** Similar to τ, we use a Log-Normal Random Walk MH update.
- **Jacobian:** $\log(\alpha_{\text{new}}) - \log(\alpha_{\text{old}})$ is added to the acceptance ratio.

3.2.3 Update Logic for σ (Discount)

- **Constraints:** $\sigma \in (0, 1)$.
- **Prior:** Beta prior, $\sigma \sim \text{Beta}(a_\sigma, b_\sigma)$.
- **Algorithm:** We use a **Reflected Random Walk**.
  1. Generate drift: $\epsilon \sim \mathcal{N}(0, \delta_\sigma^2)$.
  2. Candidate: $\sigma' = \sigma + \epsilon$.
  3. **Reflection:** If σ' falls outside $(0, 1)$, it is reflected back (e.g., if $\sigma' > 1$, $\sigma' \to 2 - \sigma'$). This preserves the detailed balance for a uniform proposal on a bounded domain.
- **Jacobian:** Since the proposal is symmetric ($q(\sigma' \mid \sigma) = q(\sigma \mid \sigma')$), the Jacobian term is 0 (i.e., ratio is 1).

Chapter 4

Sampling Utilities (`multiview_utils.cpp`)

This file contains the "heavy lifting" of the Gibbs Sampler: calculating probabilities for table assignments, managing data structures, and handling the birth/death of clusters.

4.1 Likelihood Computations: Conditional vs. Marginalized

4.1.1 Standard Likelihood (`compute_f_vk`): The Plugin Approximation

The function `compute_f_vk` calculates the probability density of observation y_{vi} belonging to an existing cluster k. The code uses a **Conditional Gibbs Sampling** approach, calculating the likelihood based on the current empirical statistics of the dish, which serves as a plug-in approximation for the parameter μ_k:

$$f(y_{vi} \mid \mu_k, \tau_v) = \frac{1}{\sqrt{2\pi\tau_v}} \exp\left(-\frac{(y_{vi} - \bar{y}_k)^2}{2\tau_v}\right)$$

The mean $\bar{y}_k$ is estimated using the current customers in the cluster (counts n_{vk} and sum $\sum y$).

Comparison with Literature: This approach **differs** from the **Fully Marginalized Gibbs Sampler** described in the literature, which requires calculating the **Posterior Predictive Distribution (PPD)** by analytically integrating the cluster parameter θ^{**}_{vk} (Formula 3.1). This code employs a computationally efficient **alternative specification** which avoids this complex integration.

4.1.2 New Cluster Likelihood (`compute_f_vk_new`)

Calculating the likelihood for a *new* cluster (dish) relies on the **Prior Predictive Distribution (PPrD)**. Since the mean of a new dish is unknown, we use a predictive distribution with an inflated variance:

$$\sigma^2_{\text{pred}} = \tau_v + \sigma^2_{\text{global\_data}}$$

This inflated variance allows the model to "see" potential clusters far from the origin (origin is the prior mean, assumed 0.0), making it possible to assign outliers to new clusters.

—

4.2 Table Assignment Probabilities

4.2.1 Main Calculation Function (`compute_table_probs_with_cache`)

This function calculates the non-normalized probabilities for assigning customer i to all existing tables (t_{old}) or to a new table (t_{new}).

1. **Existing Table Probabilities (t_{old}):** Calculated based on the Pitman-Yor mass term and the product of view-specific likelihoods:

$$P(t_i = t_{\text{old}}) \propto \underbrace{(n_t^{(i)} - \sigma_{\text{global}})}_{\text{PYP Mass}} \times \prod_{v=1}^{d} f_{vk_t}(y_{vi})$$

2. **New Table Probability (t_{new}):** Calculated based on the Pitman-Yor mass term and the marginalization over possible dish assignments in the second layer:

$$P(t_i = t_{\text{new}}) \propto \underbrace{(\alpha_{\text{global}} + T_{\text{active}}\sigma_{\text{global}})}_{\text{PYP Mass}} \times \prod_{v=1}^{d} P(y_{vi} \mid t_{\text{new}}, \mathbf{k}^{(-i)})$$

4.2.2 Marginalization for New Tables (`compute_marginal_likelihood_new_table`)

This function computes the likelihood $\left[P(y_{vi} \mid t_{\text{new}}, \mathbf{k}^{(-i)})\right]$ for a single view v, which is required for the new table calculation. This is a mixture over all existing dishes (k_{old}) and a new dish (k_{new}) in the view-specific restaurant (Layer 2 of the HDP):

$$P(y_i \mid t_{\text{new}}) = \frac{1}{\alpha_v + L_v}\left[\sum_{k \in \text{existing}} (l_{vk} - \sigma_v) f_{vk}(y_{vi}) + (\alpha_v + K_{\text{active}}\sigma_v) f_{vk_{\text{new}}}(y_{vi})\right]$$

Here, l_{vk} is the count of tables assigned to dish k, and L_v is the total number of tables in view v. This term constitutes the likelihood factor in the t_{new} probability.

4.2.3 Optimizations

- **Caching:** The likelihood values f_{vk} are computed once per dish k and stored in the `cache_fvk` map. Since multiple tables can share the same dish, this avoids redundant computations within the loop over all tables.
- **Workspace Management:** A pre-allocated workspace (`std::vector<std::unordered_map>`) is reused in each iteration, reducing memory allocation overhead.

—

4.3 State Transition Utilities

4.3.1 Customer Removal (`remove_customer`)

This function implements the "leaving the restaurant" step for customer i:

- **Table Counts (Layer 1):** Decrements n_t for the old table t and removes i from `customers_at_table`.
- **Dish Counts (Layer 2):** For every view v, it decrements n_{vk} for the assigned dish k and updates the sufficient statistics ($\sum y$, $\sum y^2$).
- **Table Clean-up:** If n_t reaches zero, the table is considered empty. The function removes the table t from the active list T using a **swap-and-pop** optimization, and decrements the table count l_{vk} for the associated dishes in every view.

4.3.2 Customer Addition (`add_customer_to_existing_table`)

This function adds customer i back to an existing table t, updating all associated counts and statistics:

- Updates n_t and adds i to `customers_at_table`$[t]$.
- For every view v, it updates n_{vk}, $\sum y$, and $\sum y^2$ for the assigned dish $k = \text{dish\_of}[v][t]$.

4.3.3 Table/Dish Creation

- `create_empty_table`: Appends a new entry to the table-level state vectors (`n_t`, `customers_at_table`, `dish_of`) and increments T.
- `add_customer_to_new_table`: Assigns customer i to the newly created table t_{new} and sets $n_{t_{\text{new}}} = 1$.
- `sample_dish_for_new_table`: When a new table is chosen, the dishes for that table must be sampled according to the PYP probabilities (weights based on l_{vk} and the likelihood f_{vk}), following the structure described by the mixture in `compute_marginal_likelihood_new_table`. If a new dish is sampled, it is created by incrementing $V.K$ and pushing back zeros to the dish state vectors.
- `assign_dishes_new_table`: Calls `sample_dish_for_new_table` for each view v to determine k_{vt} and updates the dish-level counts (l_{vk}, n_{vk}) and statistics.

Chapter 5

Main Loop and Initialization (`multiview_gibbs.cpp`)

This file serves as the entry point for the R interface and contains the main Gibbs Sampler loop. It orchestrates the initialization, the iterative sampling of latent variables, and the hyperparameter updates.

5.1 State Initialization

The function `initialize_state_from_data` sets up the initial configuration of the chain to ensure a valid starting point.

5.1.1 Random Partitioning

Instead of starting with a single cluster or one cluster per data point, the initialization performs a random assignment:

1. **Tables:** N customers are randomly assigned to $T_{\text{init}} = 4$ tables.
2. **Dishes:** Each table, in each view, is randomly assigned to $K_{\text{init}} = 2$ dishes.

This random start helps prevent the chain from getting stuck in local modes immediately, although a proper burn-in period is still required.

5.1.2 Data Statistics and Hyperparameters

Critical statistics are pre-calculated during initialization:

- `n_t`, `n_vk`, `l_vk`: Counts are populated based on the random assignments.
- `sum_y`, `sum_y2`: Sufficient statistics for the clusters are computed.
- **Variance Initialization:** The view-specific variance τ_v is initialized based on a fraction of the empirical variance of the data: $\tau_v \approx 0.25 \times \text{Var}(Y_v)$. This provides a reasonable starting scale for the Gaussian kernels.

5.2 The Gibbs Sampler Loop

The function `gibbs_sampler` implements the Markov Chain Monte Carlo (MCMC) algorithm.

5.2.1 Algorithm Structure

The loop runs for M iterations. In each iteration, it performs a full sweep over all latent variables:

Algorithm 1 Multiview Gibbs Sampler Structure

```
 1: Input: Data $Y$, Iterations $M$, Burn-in $B$, Thinning $h$
 2: Initialize workspace memory (Hash Maps)
 3: for iter = 1 to $M$ do
 4:     for $i = 1$ to $N$ do
 5:         Remove customer $i$ from current table (remove_customer)
 6:         Calculate probabilities for all existing tables and new table
 7:         Sample new table assignment $t^*$
 8:         if $t^*$ is new then
 9:             Create new table
10:             Sample dishes for new table in all views (assign_dishes_new_table)
11:         end if
12:         Add customer $i$ to table $t^*$ (add_customer_*)
13:     end for
14:     Update Hyperparameters $\alpha, \sigma, \tau$ (update_hyperparameters)
15:     if iter $> B$ and iter $\pmod{h} == 0$ then
16:         Save state (save_state)
17:     end if
18: end for
```

5.2.2 Performance Optimizations

A key optimization implemented in this file involves memory management for the probability calculation.

```cpp
// Workspace allocation outside the loop
std::vector<std::unordered_map<int, double>> workspace_cache(d);
for(auto &m : workspace_cache) m.reserve(64);

// Inside the loop
compute_table_probs_with_cache(i, ..., workspace_cache);
```

By allocating the `workspace_cache` vector of maps **outside** the main loop and passing it by reference, we avoid allocating and deallocating memory $N \times M$ times (which would be millions of operations). The maps are simply cleared and reused, reducing overhead significantly.

Comparison of Gibbs Sampling Approaches for HDP-PYP with Gaussian Kernels Analysis for Boss December 3, 2025

Conclusion: The **structure** of the HDP sampler (the use of the two-level counts and masses) is implemented correctly in the user's code. However, the **numerical basis** for the likelihoods (f_{vk}) is an approximation (Plugin Estimator) rather than the exact PPD (Marginalized Integration) shown in the paper's Formula (3.1).

Chapter 6

R Interface and Gibbs Sampler Workflow

This document outlines the workflow for running the **Multi-View Hierarchical Pitman-Yor Process (H-PYP) Gibbs sampler**, executed via the `Rcpp` interface, and the subsequent analysis of the clustering results in the R environment.

6.1 Setup and Dependencies

The analysis requires several R libraries for data manipulation, plotting, statistical analysis, and the Rcpp interface.

```r
library(dplyr)
library(ggplot2)
library(Rcpp)
library(mcclust)
library(mcclust.ext)
# ... other libraries
Rcpp::sourceCpp("multiview_gibbs.cpp")
```

The key step is `Rcpp::sourceCpp("multiview_gibbs.cpp")`, which compiles the C++ sampler and makes the function `run_gibbs_cpp` available directly in R.

6.2 Data Generation (Data Generation Step)

The data is simulated to create known ground truth clusters across multiple views ($V = 5$) to validate the sampler's accuracy.

6.2.1 Data Structure

A dataframe `x` is created with 200 data points (customers, $N = 200$).

- **Views 1, 3, 4, 5:** Simulated with two distinct Gaussian clusters. The ground truth has 2 clusters.
- **View 2:** Simulated with three distinct Gaussian clusters. The ground truth has 3 clusters.
- **Input Format:** The C++ sampler requires a list of numeric vectors, where each vector represents a view.

```r
data_views <- list(
  as.vector(x$view1),
  as.vector(x$view2),
  # ... and so on for view3, view4, view5
)
```

6.3 Gibbs Sampler Execution and Output Analysis

6.3.1 Execution Parameters

The main C++ function, `run_gibbs_cpp`, is called with the data and MCMC parameters.

```r
nsim <- 10000
burn_in <- 9000
thin <- 1

res_gibbs <- run_gibbs_cpp(
  data_views = data_views,
  M = nsim,
  burn_in = burn_in,
  thin = thin
)
```

- **`M`:** Total number of MCMC iterations (≈ 10000).
- **`burn_in`:** The number of initial iterations to discard (≈ 9000). This period allows the Markov chain to converge to the stationary distribution.
- **`thin`:** The thinning interval (here 1). Since M and *burn_in* are close, only the last ≈ 1000 iterations are kept for analysis.

6.3.2 Gibbs Sampler Output Structure (`res_gibbs`)

The C++ function returns a list (`res_gibbs`) containing the saved state of the model across the post-burn-in iterations. The structure reflects the two-level nature of the H-PYP and uses nested R lists to track the saved states S:

- **`table_of`:** A **list of vectors** (length S). Each vector (length N) maps customer i to its **Table ID** (Layer 1 of the H-PYP).

  $$\texttt{res\_gibbs\$table\_of}[[s]] \rightarrow \text{Customer ID} \rightarrow \text{Table ID}$$

- **`dish_of`:** A **list of lists** (length S). Each sub-list (length V) contains a vector mapping the Table ID to its **Dish ID** (Layer 2 cluster).

  $$\texttt{res\_gibbs\$dish\_of}[[s]][[v]] \rightarrow \text{Table ID} \rightarrow \text{Dish ID (Cluster)}$$

- **Hyperparameters:** Vectors saving the values for α, σ, and τ (at global and view levels) for diagnostics and further analysis.

Example of Two-Level Output (Simplified):
If Customer 1 is at Table 3, and Table 3 is assigned to Dish 7 in View 1, the output structure is:

- `table_of`$[i = 1] = 3$
- `dish_of`$[v = 1][t = 3] = 7$

The final cluster assignment for Customer 1 in View 1 is **Dish 7**. The total length of these lists (S) is the number of iterations saved after burn-in and thinning.

6.4 Result Extraction and Mapping (`get_final_clusters`)

The function `get_final_clusters` performs the necessary mapping to obtain the final, usable cluster assignments, typically using the last saved state ($s = S$):

```r
# 1. Get final assignments (last iteration)
raw_tables <- res_gibbs$table_of[[last_iter_idx]]
raw_dishes <- res_gibbs$dish_of[[last_iter_idx]]

# 2. Adjust indices (C++ uses 0-based; R uses 1-based)
tables_r_index <- raw_tables + 1

# 3. Map Customer -> Table -> Dish (Cluster)
for (v in 1:n_views) {
  dishes_for_view <- raw_dishes[[v]]
  # Lookup: dishes_for_view[table_ID] = dish_ID
  cluster_matrix[, v] <- dishes_for_view[tables_r_index]
}
```

The result is `my_clusters`: an $N \times V$ matrix where $N = 200$ and $V = 5$. Each element contains the final cluster ID (dish ID) assigned to that customer in that view.

6.5 Visualization and Evaluation

6.5.1 Visualization (`plot_predicted_interaction`)

The custom function `plot_predicted_interaction` visualizes the interaction between two views based on the **predicted** clusters.

- The **shape** of the point is determined by the cluster assignment found in the X-axis view.
- The **color** of the point is determined by the cluster assignment found in the Y-axis view.

This visualization illustrates how the H-PYP provides different (**view-specific**) cluster assignments for the same data points, capturing the model's core multi-view nature.

6.5.2 Evaluation (Adjusted Rand Index)

The **Adjusted Rand Index (ARI)** is used to quantitatively compare the predicted cluster assignments (`my_clusters`) against the known ground truth (`true_clust_list`). An ARI close to 1 indicates high concordance between the predicted and true clustering.

```r
ari_scores <- sapply(1:5, function(v) mcclust::arandi(my_clusters[,v],
    true_clust_list[[v]]))
```

The resulting vector `ari_scores` provides a measure of accuracy for each of the five views.