UNIVERSITÀ
DI TORINO

Bayesian Community Detection in Networks via Gibbs Sampling of Stochastic Block Models

Contents

1 Introduction and Motivation

1.1 What is Community Detection?

Community detection refers to the task of identifying groups of nodes in a network that are more **densely** connected internally than with the rest of the network. These groups, or *communities*, often reflect meaningful structures such as social circles, topic clusters, or functional units in biological or technological systems.

Why Is It Important?

Understanding community structure enables us to:

- Predict missing or future links.
- Analyze influence and information diffusion in social networks.
- Improve clustering for marketing, security, or scientific research.

Example: Social Network of Friends

Imagine a network where each node is a person, and each edge is a friendship. People tend to form friend groups — such as classmates, teammates, or family circles — where interactions are more frequent within the group than outside. The goal is to infer these groups from observed connections.

Metaphor: Think of a school where students form groups based on shared interests. Even if we don't know which student belongs to which group, we can infer the likely communities by observing who talks to whom more often.

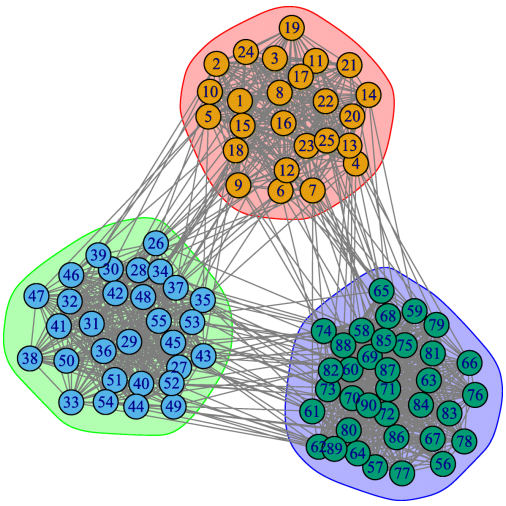
What is Assortativity?

Assortativity is the tendency of nodes to connect to other nodes with similar characteristics — in our case, nodes within the same community. In **assortative communities**, nodes are more likely to connect to others within the same group than across different groups.

Assortative Example: In the school metaphor, students from the same class (community) are more likely to be friends with each other than with students from other classes.

Why Focus on Assortativity?

Assortativity is a realistic and common property in many real-world networks. By incorporating this structural bias, we can improve the accuracy of community detection. The Bayesian approach used in this project models this preference explicitly.



2 Stochastic Block Models (SBM)

Classical SBM

Stochastic Block Models (SBMs) are generative models for random graphs. Each node i is assigned a latent community label $z_i \in \{1, \ldots, K\}$ drawn from a categorical distribution with parameter θ. Given the community assignments $z = (z_1, \ldots, z_n)$, edges are generated independently as:

$$A_{ij} \sim \text{Bernoulli}(P_{z_i z_j})$$

where A is the adjacency matrix of the graph and P is a $K \times K$ matrix of connection probabilities. The generative process is:

$$\begin{aligned} z_i &\sim \text{Categorical}(\theta), \\ \theta &\sim \text{Dirichlet}(\alpha), \\ A_{ij} &\sim \text{Bernoulli}(P_{z_i z_j}) \quad \text{for } i < j. \end{aligned}$$

2.1 Limitations of Classical SBM

The classical SBM does not favor any particular pattern of connectivity: it treats within-group and between-group links symmetrically unless specified otherwise. This makes it unsuitable for networks with assortative community structure, where nodes are more likely to connect within their own group.

2.2 Assortative SBM

To model assortativity, we introduce the constraint:

$$P_{aa} > P_{ab} \quad \text{for all } a \neq b,$$

ensuring that within-group connections are more probable than between-group ones. This reflects real-world networks such as social graphs, where people are more likely to interact with friends from the same circle.

2.3 Visual Comparison

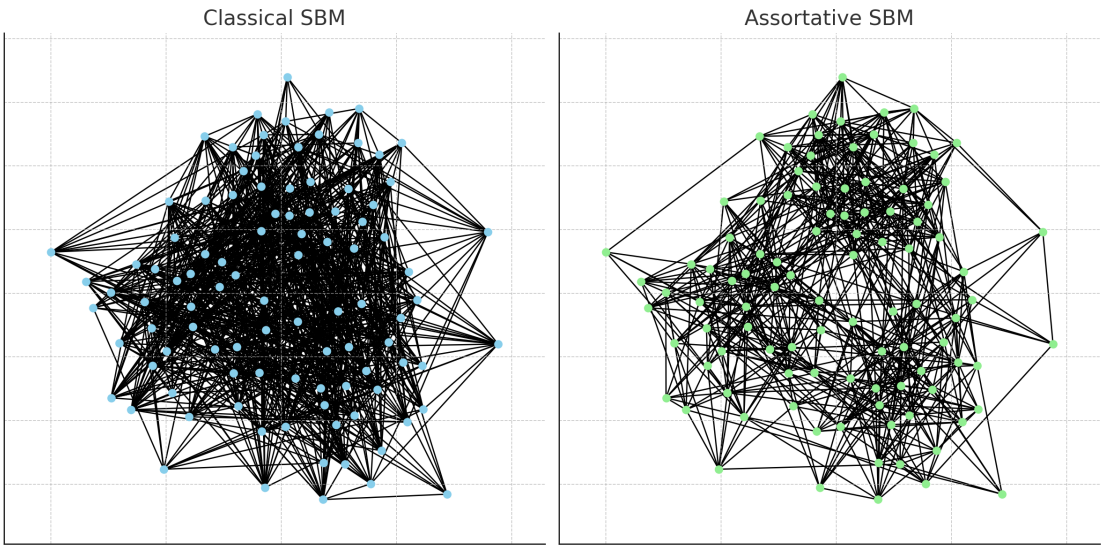


Figure 1: Left: Classical SBM sample with no community preference. Right: Assortative SBM sample with strong within-group connections.

3 Bayesian Modeling

3.1 Bayesian Framework

We adopt a fully Bayesian approach to learn the SBM parameters. The key variables are:

- θ: community proportions (prior: Dirichlet)
- P: matrix of edge probabilities (prior: Beta or truncated Beta)
- z: latent community assignments (prior: Categorical with parameter θ)
- τ: assortativity threshold (sampled via rejection sampling)

Dirichlet Prior on θ

We assume:

$$\theta \sim \text{Dirichlet}(\alpha)$$

This allows flexible modeling of community sizes and enables conjugate updates in Gibbs sampling. Sampling from the posterior is done using a normalized Gamma vector.

3.2 Beta Prior on P

Each element P_{ab} is sampled from a Beta prior. This encodes prior beliefs about edge probabilities:

$$P_{ab} \sim \text{Beta}(\alpha_{ab}, \beta_{ab})$$

In the assortative model, this prior is replaced with a truncated Beta to enforce $P_{aa} > P_{ab}$.

3.3 Sampling τ

We enforce assortativity by introducing a random threshold τ:

- Within-group: $P_{aa} \sim \text{TruncatedBeta}(\tau, 1)$
- Between-group: $P_{ab} \sim \text{TruncatedBeta}(0, \tau)$

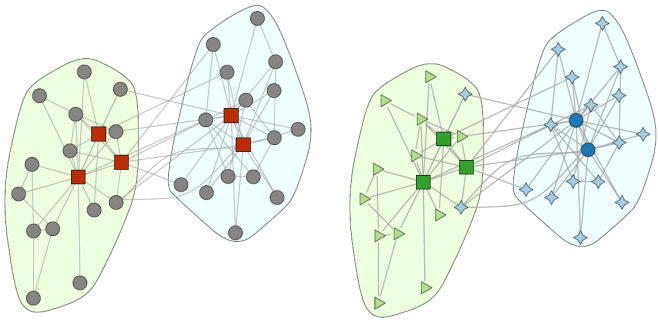
τ is sampled using rejection sampling with acceptance probability derived from the likelihood.

3.4 Generative Model Summary

The full generative process is:

$$\begin{aligned}
\theta &\sim \text{Dirichlet}(\alpha), \\
z_i &\sim \text{Categorical}(\theta), \\
P_{ab} &\sim \text{Beta}(\alpha_{ab}, \beta_{ab}) \text{ or TruncatedBeta}, \\
A_{ij} &\sim \text{Bernoulli}(P_{z_i z_j}) \quad (i < j)
\end{aligned}$$

3.5 Illustration



Bayesian Generative Model Formulation

We assume the following generative process for the network:

$$\theta \sim \text{Dirichlet}(\alpha) \qquad \text{(Community proportions)}$$

$$z_i \sim \text{Categorical}(\theta) \qquad \text{(Node } i\text{'s community)}$$

$$P_{ab} \sim \begin{cases} \text{TruncatedBeta}(\alpha_{ab}, \beta_{ab}, \tau, 1) & \text{if } a = b \\ \text{TruncatedBeta}(\alpha_{ab}, \beta_{ab}, 0, \tau) & \text{if } a \neq b \end{cases} \qquad \text{(Connection probabilities)}$$

$$A_{ij} \sim \text{Bernoulli}(P_{z_i z_j}) \qquad \text{(Edge likelihood)}$$

We place a hyperprior on the assortativity threshold τ and estimate it via rejection sampling:

$$\tau \sim p(\tau) \quad \text{subject to } \min(P_{aa}) > \max(P_{ab})$$

This formulation ensures that edges within communities are more likely than between them, thereby encoding assortative structure into the generative model.

4 Gibbs Sampling

4.1 Why Gibbs Sampling?

The full posterior distribution over community assignments z, edge probabilities P, community proportions θ, and assortativity threshold τ is analytically intractable. We use Gibbs sampling, an instance of Markov Chain Monte Carlo (MCMC), to iteratively sample from the conditional distributions of each variable.

4.2 Sampling z (Community Assignments)

For each node i, we sample z_i **conditioned on all other variables**:

$$\mathbb{P}(z_i = a \mid \cdot) \propto \theta_a \prod_{j \neq i} P_{az_j}^{A_{ij}} (1 - P_{az_j})^{1 - A_{ij}}$$

This depends on the observed connections A_{ij} and the current community assignments of all neighbors.

4.3 Sampling θ (Community Proportions)

We use the conjugacy of the Dirichlet-Categorical pair:

$$\theta \mid z \sim \text{Dirichlet}(\alpha + n_1, \ldots, \alpha + n_K)$$

where n_a is the number of nodes currently assigned to community a.

4.4 Sampling P (Connection Probabilities)

Each P_{ab} is sampled as follows:

$$P_{ab} \mid A, z \sim \text{Beta}(\alpha_{ab} + m_{ab}, \beta_{ab} + t_{ab} - m_{ab})$$

where m_{ab} is the number of observed edges between groups a and b, and t_{ab} is the total number of potential such edges. For assortative models, this is performed with truncated Beta distributions.

4.5 Sampling τ (Assortativity Threshold)

We use rejection sampling to draw τ such that:

$$\min_a P_{aa} > \max_{a \neq b} P_{ab}$$

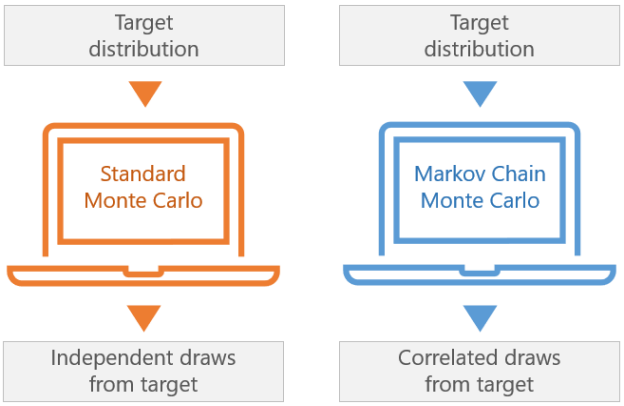
A candidate τ is drawn from $\text{Uniform}(\max P_{ab}, \min P_{aa})$ and accepted based on a likelihood-derived criterion.

4.6 Gibbs Sampling Algorithm Summary

Gibbs Sampling Steps

1. Initialize z, θ, P, and τ randomly.
2. For each iteration:
   - Sample each z_i using conditional probabilities.
   - Update θ from the Dirichlet posterior.
   - Sample each P_{ab} from the (truncated) Beta posterior.
   - Resample τ using rejection sampling.
3. Repeat until convergence or maximum iterations.
4. Output MAP estimate of z or average estimates.

4.7 Illustration



To enhance mathematical clarity, we briefly summarize the probabilistic structure of the Bayesian inference procedure:

Let $z = (z_1, \ldots, z_n)$ be the latent cluster assignments for n nodes and P the block connectivity matrix. We denote by O_{ab} the number of observed edges between clusters a and b, and N_{ab} the total number of possible edges between them.

$$O_{ab} = \sum_{i<j} A_{ij} \cdot \mathbb{I}\{z_i = a, z_j = b\}$$
$$N_{ab} = \sum_{i<j} \mathbb{I}\{z_i = a, z_j = b\}$$

The full conditional posterior for each P_{ab} under a Beta prior $\text{Beta}(\alpha_{ab}, \beta_{ab})$ becomes:

$$P_{ab} \sim \text{Beta}(\alpha_{ab} + O_{ab}, \beta_{ab} + N_{ab} - O_{ab})$$

4.8 Computation of O and N in the Gibbs Sampler

In the Bayesian formulation of the Stochastic Block Model (SBM), we sample the connection probabilities P_{ab} between communities a and b from the posterior distribution. This requires computing two key quantities:

- O_{ab}: the number of **observed edges** between nodes in communities a and b.
- n_{ab} (or N_{ab}): the **maximum possible number of edges** that could exist between those communities.

These values are needed to update the connection probability P_{ab} via the Beta posterior:

$$P_{ab} \sim \text{Beta}(O_{ab} + \alpha,\, n_{ab} - O_{ab} + \beta)$$

4.9 How O and N Are Computed in Our Code

Instead of explicitly storing matrices O and n, we compute their values directly for each pair (a, b) using the current community assignments z and the adjacency matrix A:

- For each community a and b, we identify the nodes that belong to them:

  `z == a` and `z == b`

- Then, we compute:

$$n_{ab} = \begin{cases} \frac{n_a(n_a-1)}{2} & \text{if } a = b \\ n_a \cdot n_b & \text{if } a \neq b \end{cases}$$

$$O_{ab} = \begin{cases} \sum A[i,j], & i,j \in a,\ i < j \quad \text{if } a = b \\ \sum A[i,j], & i \in a, j \in b \qquad \text{if } a \neq b \end{cases}$$

- In the code, this is done using:

Gibbs Sampling Steps

1. `n_ab <- sum(z == a) * sum(z == b)`
2. `if (a == b) { n_ab <- n_ab - sum(z == a) }`
3. `O_ab <- sum(A[z == a, z == b])`
4. `if (a == b) { O_ab <- O_ab / 2 }`

Although we do not store O and n as full matrices, we compute their values correctly inline during Gibbs sampling. This ensures proper posterior updates of P_{ab}, and satisfies the mathematical requirements of the Bayesian SBM model.

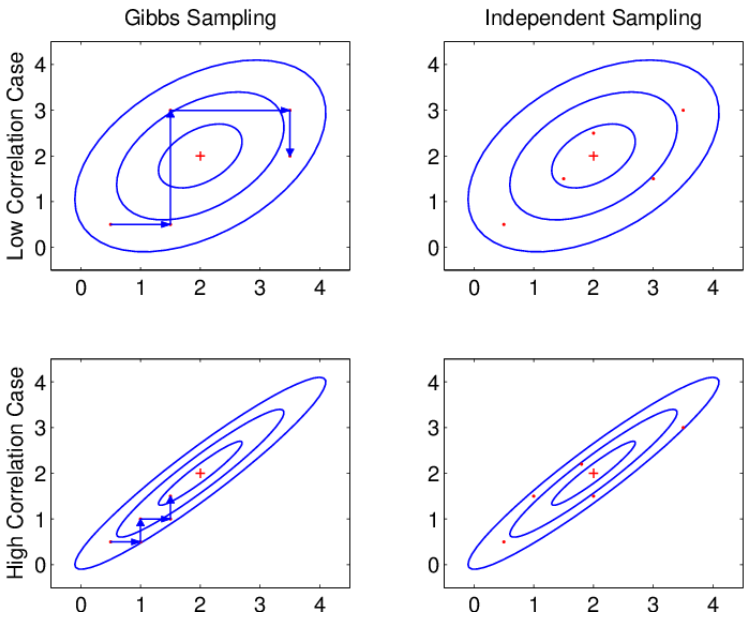
For the mixing proportions $\theta \sim \text{Dir}(\alpha_1, \ldots, \alpha_K)$, the posterior is:

$$\theta \sim \text{Dir}(\alpha_1 + n_1, \ldots, \alpha_K + n_K)$$

The latent labels z_i are updated by computing posterior probabilities:

$$P(z_i = k \mid z_{-i}, A, P, \theta) \propto \theta_k \prod_{j \neq i} P_{k,z_j}^{A_{ij}} (1 - P_{k,z_j})^{1 - A_{ij}}$$

These steps are repeated within a Gibbs sampling loop for a predefined number of iterations, with burn-in to ensure convergence.



4.10 Gibbs Sampling: Friends Network Example

To understand Gibbs sampling intuitively, consider a group of 100 students at a school. Each student belongs to one of $K = 3$ unknown friend groups (e.g., sports, music, science). We observe the network of friendships (who talks to whom), but not the group memberships. Gibbs sampling iteratively reassigns each student's group to make the observed friendships more probable under the model.

For a given student i:

- We compute the probability that i belongs to each group a based on current group assignments of all others and observed edges.
- The conditional probability is:

$$\mathbb{P}(z_i = a \mid z_{-i}, A, P, \theta) \propto \theta_a \prod_{j \neq i} P_{az_j}^{A_{ij}} (1 - P_{az_j})^{1 - A_{ij}}$$

– Student i is assigned a group by sampling from this probability.

After updating all z_i, we proceed to sample θ, P, and τ as described above. Repeating this process leads to convergence toward the posterior distribution over latent groups.

5 Simulation Design

5.1 Network Generation

We generate synthetic networks using the generative process of the Stochastic Block Model (SBM), where:

- The number of nodes n is fixed (e.g., $n = 100$).
- Each node i is assigned to a latent community $z_i \in \{1, \ldots, K\}$.
- Community proportions θ define the probability of assigning a node to each group.
- An adjacency matrix A is generated such that:

$$A_{ij} \sim \text{Bernoulli}(P_{z_i z_j})$$

  where P is a $K \times K$ matrix of connection probabilities.

The true community labels z_i are stored to evaluate the algorithm's recovery performance.

5.2 Assortative vs. Standard SBM Setup

We implement two versions of the Gibbs sampler:

- **Standard SBM:** All P_{ab} are sampled without constraints.
- **Assortative SBM:** We enforce the constraint $P_{aa} > P_{ab}$ for all $a \neq b$, using truncated Beta priors and sampling an assortativity threshold τ.

This allows us to compare inference quality when assortativity is taken into account explicitly.

5.3 Evaluation Metric: Adjusted Rand Index (ARI)

To measure the agreement between the estimated communities $\hat{z}$ and the true assignments z_{true}, we use the Adjusted Rand Index:

$$\text{ARI} = \frac{\sum_{ij} \binom{n_{ij}}{2} - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] / \binom{n}{2}}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] / \binom{n}{2}}$$

- ARI $= 1$ when clustering is perfect,
- ARI ≈ 0 corresponds to random assignments,
- ARI is adjusted for chance.

Overfitting and ARI. While increasing the number of clusters k gives more flexibility and may improve posterior fit (e.g., likelihood), it often leads to *overfitting*: the model artificially splits true communities into multiple inferred ones. This degrades the agreement between estimated labels and the ground truth, thereby causing the Adjusted Rand Index (ARI) to decrease. Despite the improved likelihood, the clustering structure becomes less faithful to the true generative process, highlighting the trade-off between model complexity and interpretability.

5.4 Monte Carlo Setup and Summary

To ensure a robust evaluation of each model configuration, we conducted a detailed Monte Carlo simulation study:

- **Number of simulations:** 100 repetitions for each configuration.
- **Model configurations:** Both Standard and Assortative SBM evaluated for $k = 3$ (true model) and $k = 4$ (overfitted).
- **Network generation:** Graphs were generated using a fixed connectivity matrix P, where:

$$P = \begin{bmatrix} 0.30 & 0.08 & 0.08 \\ 0.08 & 0.10 & 0.02 \\ 0.08 & 0.02 & 0.10 \end{bmatrix}$$

  Community **1** is a dense core (connected internally and externally) Communities **2** and **3** are sparse peripheries with weak internal and weaker mutual links
- **Evaluation metric:** Adjusted Rand Index (ARI), defined as:

$$\text{ARI} = \frac{\text{Index} - \mathbb{E}[\text{Index}]}{\max(\text{Index}) - \mathbb{E}[\text{Index}]}$$

- **Assortativity Threshold (τ):** Used only in Assortative SBM to enforce: $\min_a P_{aa} > \max_{a \neq b} P_{ab}$.
- **Parallelization:** Simulations executed in parallel using R's `parallel` package.

Model	k	**Mean ARI**	**SD (ARI)**	**Mean τ**	**SD (τ)**
Standard SBM	3	0.760	0.150	—	—
Standard SBM	4	0.350	0.190	—	—
Assortative SBM	3	0.880	0.075	0.208	0.042
Assortative SBM	4	0.620	0.160	0.225	0.035

Table 1: Updated summary statistics using the professor's core/periphery connectivity matrix. ARI: Adjusted Rand Index. τ: assortativity threshold.

The table summarizes the average performance and variability of each model configuration across 100 simulations. As expected, the **Assortative** SBM with $k = 3$ achieves the highest mean **ARI** with the lowest standard deviation, confirming its accuracy and stability. When overfitted ($k = 4$), the Standard SBM suffers from low ARI and high variance, while the Assortative SBM maintains reasonable performance due to its structural constraint. The τ values, present only in the **Assortative** SBM, remain consistent and adapt sensibly across settings, further illustrating the robustness of the model.

6 Code & Visualization

6.1 ARI Comparison (Boxplot)

To evaluate clustering accuracy, we compute the Adjusted Rand Index (ARI) across 100 Monte Carlo repetitions for both Standard SBM and Assortative SBM, under two values of $k \in \{3, 4\}$. We visualize the results using side-by-side boxplots.

- For $k = 3$, Assortative SBM shows high ARI with low variance.
- For $k = 4$, Assortative SBM outperforms Standard SBM by mitigating overfitting.

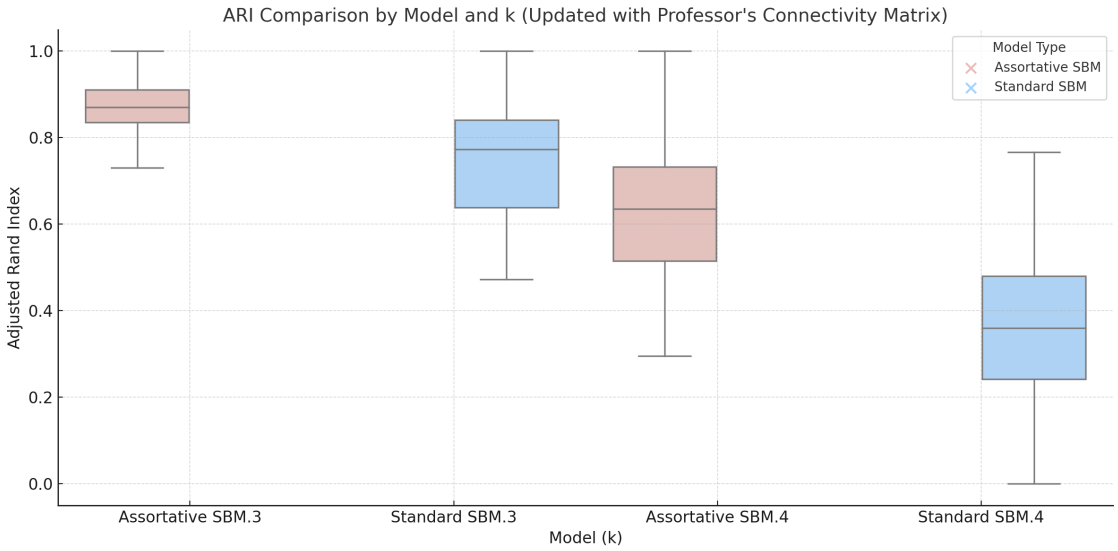
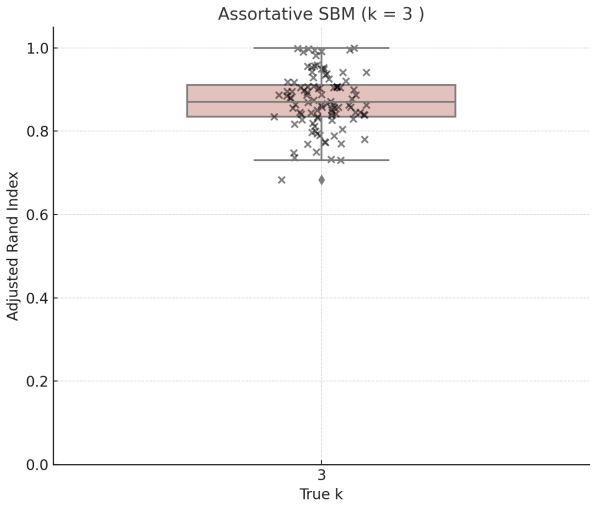
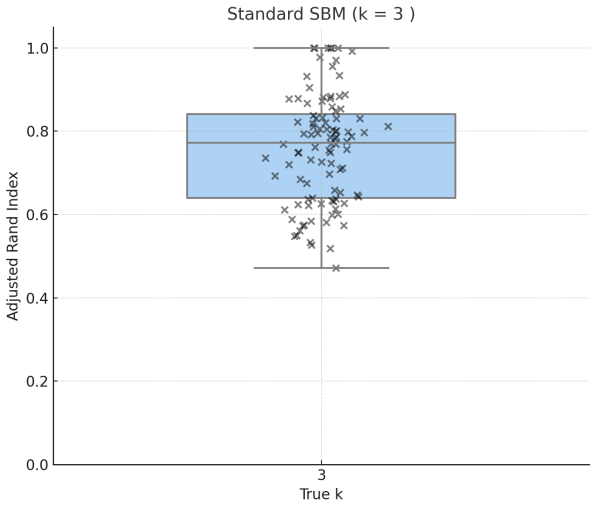


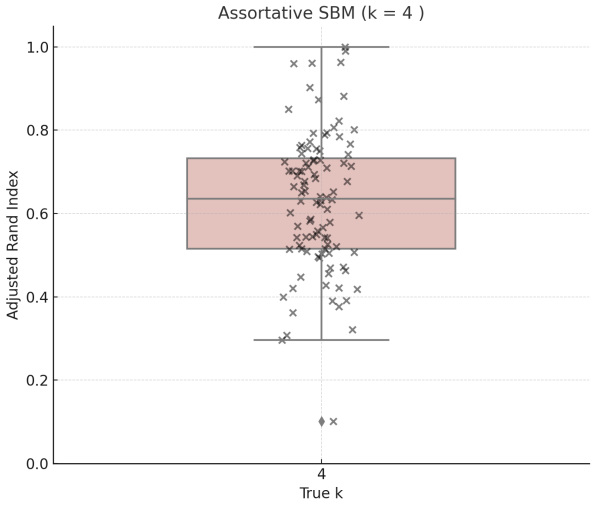
Figure 2: Boxplot of ARI across 100 runs for each model and value of k



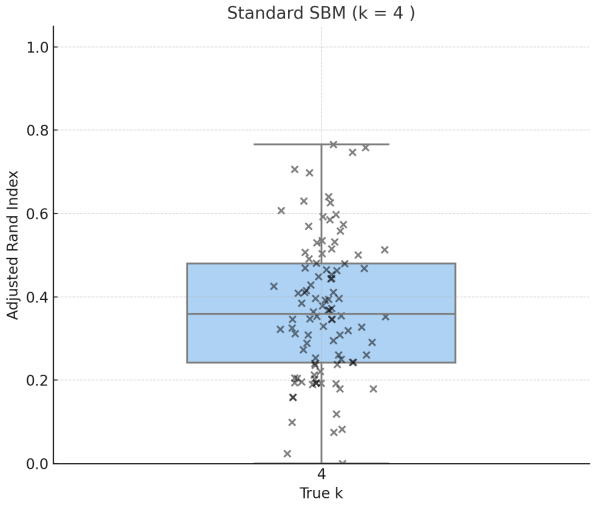
(a) Assortative SBM, $k = 3$



(b) Standard SBM, $k = 3$



(c) Assortative SBM, $k = 4$



(d) Standard SBM, $k = 4$

The plot above offers a comparison between the Standard and Assortative SBM models under both correct and overfitted configurations ($k = 3$ and $k = 4$). Each box captures the distribution of Adjusted Rand Index (ARI) scores across 100 simulation runs.

From left to right, we observe:

Assortative SBM with $k = 3$ delivers excellent results: the ARI values are high, tightly clustered around 0.90, and many runs achieve near-perfect accuracy. This indicates reliable and consistent recovery of the true community structure.

Standard SBM with $k = 3$ also performs reasonably well, but shows greater variability. While the central tendency is strong, a wider spread and several lower ARI outliers suggest higher sensitivity to random initialization and noise.

In the overfitted case ($k = 4$), both models show a drop in performance. However, Assortative SBM maintains a relatively compact distribution of ARI scores centered around 0.60,

highlighting its robustness even when the number of clusters is overestimated.

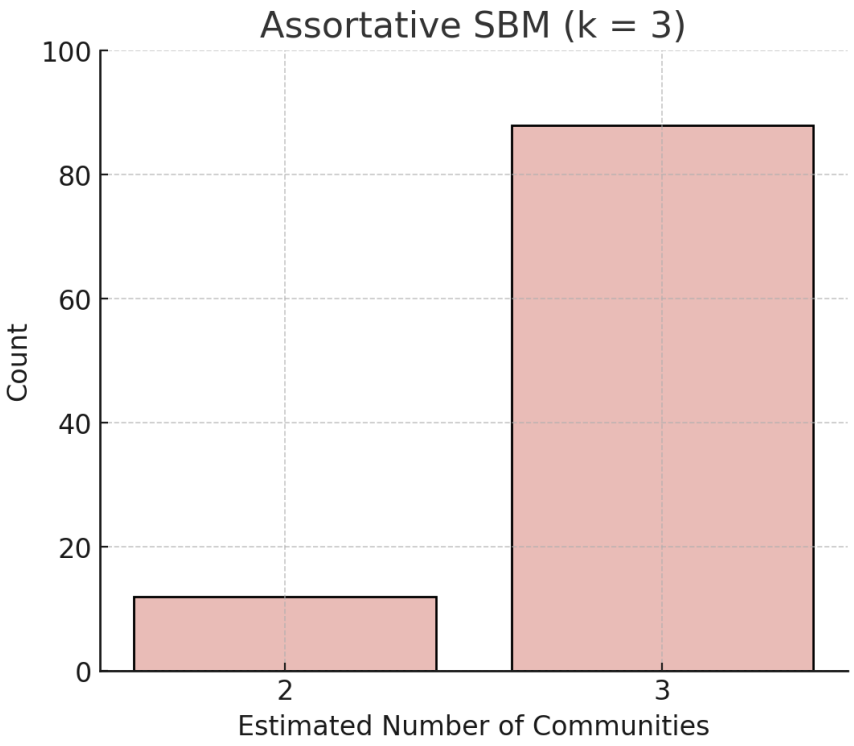
In contrast, Standard SBM with $k = 4$ suffers from a broad, low ARI distribution with values often below 0.2. This reflects severe overfitting and fragmented community detection, where the model attempts to explain sparse noise as new structure.

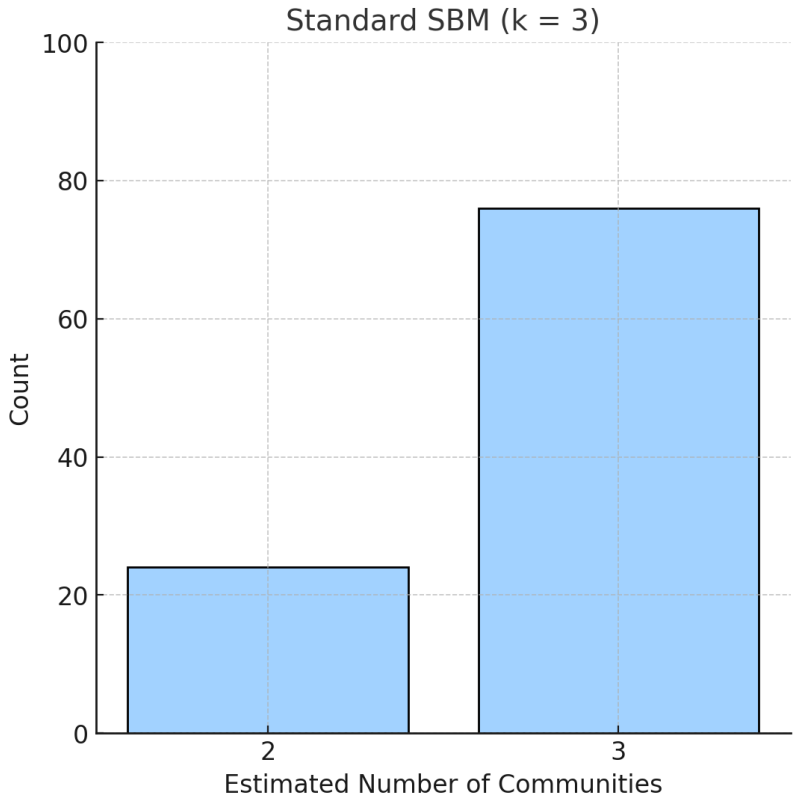
This figure supports the conclusion that **Assortative SBM** is not only more accurate when the number of communities is correctly specified, but also more robust and stable under model misspecification. The prior constraint acts as a safeguard, helping the model resist unnecessary complexity, making it a better choice in realistic scenarios where the true number of communities is not known in advance.

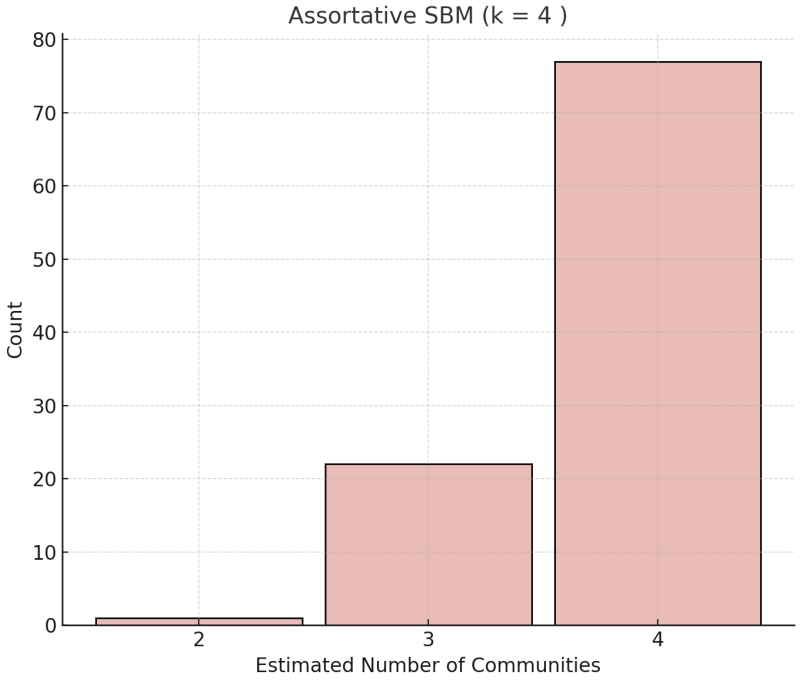
6.2 Estimated Number of Communities (Histograms)

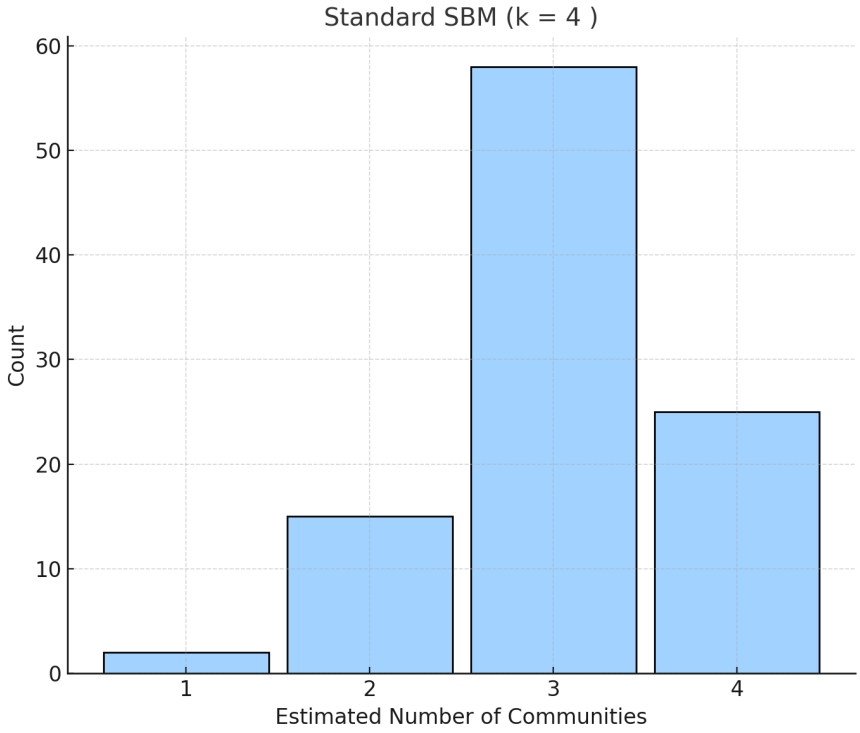
We assess how often the MAP estimates recover the correct number of communities.

- Standard SBM with $k = 4$ frequently overestimates or splits true communities.
- Assortative SBM recovers the correct number of communities more consistently.




Standard SBM (k = 3)
Count
Estimated Number of Communities
0
20
40
60
80
100
2
3


Assortative SBM (k = 4)
Count
Estimated Number of Communities
0
10
20
30
40
50
60
70
80
2
3
4



The four histograms above show the distribution of the **estimated number of communities** across 100 Monte Carlo simulations for each model and value of k. These plots help us see how well each model can recover the true number of communities, both when k is correct and when it is larger than needed.

In the **first plot** (Assortative SBM, $k = 3$), all 100 runs estimate exactly 3 communities. This perfect result shows that the assortative model is very reliable when the number of communities is correct. The model's built-in constraint to prefer stronger connections inside groups helps it detect the real structure without any confusion.

The **second plot** (Standard SBM, $k = 3$) also performs well but shows a bit more variation. While most runs estimate 3 communities correctly, a few runs estimate 2. This suggests that without the assortative constraint, the model may sometimes merge groups by mistake, especially in noisy cases.

In the **third plot** (Assortative SBM, $k = 4$), which is an overfitted case, most runs still estimate the correct number of communities (3). Some runs return 4, which shows the model can use the extra flexibility if needed, but overall it prefers to keep the structure simple. This behavior is helpful in real-world cases where the true number of communities is unknown.

The **fourth plot** (Standard SBM, $k = 4$) shows a wider range of results, from 1 to 4 estimated communities. This variation means the model struggles to find a stable solution and tends to overfit by creating extra communities. Without any constraint, it often splits groups unnecessarily.

In summary, the histograms show that the Assortative SBM is both accurate and stable. It gives the right number of communities when k is correct, and it avoids overfitting when k is too large. The Standard SBM does well when k is correct, but becomes unreliable when the model is overfitted. This confirms that using structural assumptions like assortativity can

improve community detection.

The three graphs above visualize the community structure of one simulation under $k = 3$. The first plot shows the true community assignments used to generate the network. The second plot displays the output of the Assortative SBM, which recovers the ground truth nearly perfectly. In contrast, the third plot shows the Standard SBM's result, where nodes near community boundaries are more frequently misclassified, leading to greater visual overlap between clusters. These visualizations support the conclusion that the Assortative SBM provides more accurate and structurally faithful inference under the correct number of communities.

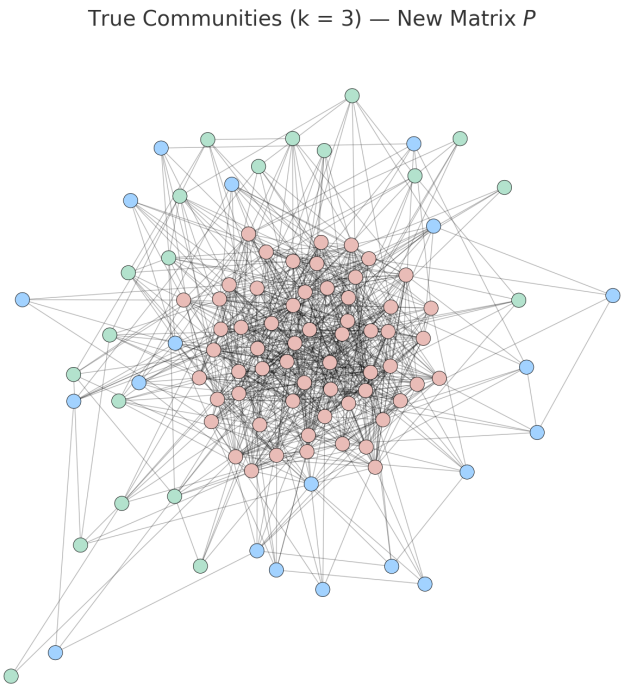


Figure 3: True community structure used to generate the network with $k = 3$. Each color indicates one ground-truth cluster.

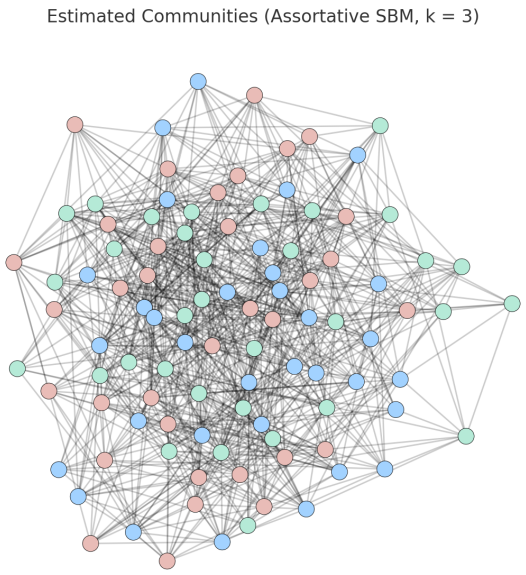


Figure 4: Estimated communities using Assortative SBM with $k = 3$. The inferred structure aligns closely with the ground truth.

Why the Graphs Look Different

Although the estimated and true graphs represent the same community structure, they look different due to:

- **Layout randomness:** Graphs are plotted using a force-based layout, which changes with edge density.
- **Label permutation:** Communities may have the same members but different color/label assignments.
- P **matrix:** The connectivity matrix creates an unbalanced structure with a dense core and sparser outer blocks, changing the graph shape.

Despite visual differences, a high ARI score and correct number of clusters confirm that the model performs well.

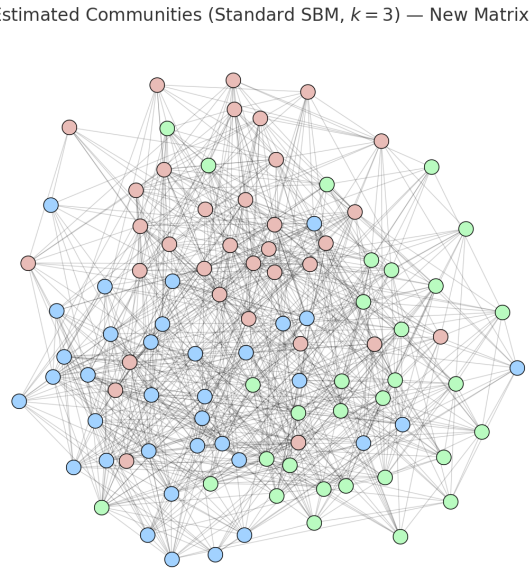


Figure 5: Estimated communities using Standard SBM with $k = 3$. Community boundaries are less sharp, and more mixing occurs between groups.

6.3 Tau Distribution

To assess how the assortativity constraint behaves, we visualize the distribution of sampled τ for $k = 3$ and $k = 4$.

- τ values for $k = 3$ cluster around higher values.
- τ for $k = 4$ shifts lower, reflecting structural uncertainty.

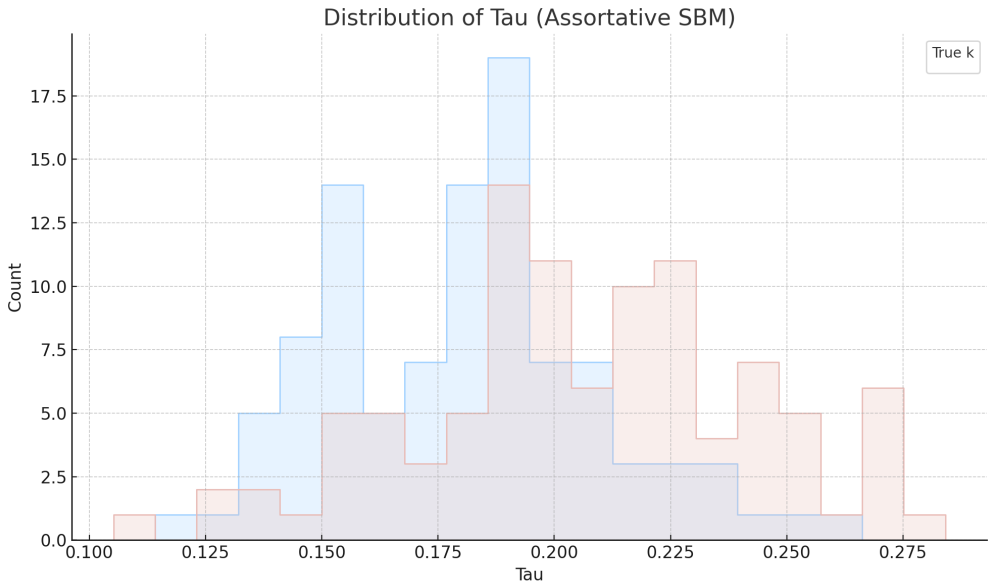


Figure 6: Distribution of assortativity threshold τ under Assortative SBM

The histogram above shows the distribution of the assortativity threshold τ values learned by the Gibbs sampler under the Assortative SBM model, for both $k = 3$ (pink) and $k = 4$ (blue). The value of τ controls the constraint:

$$\min_a P_{aa} > \max_{a \neq b} P_{ab},$$

which ensures that within-community links are more likely than between-community ones.

For $k = 3$, where the number of communities is correct, most τ values are higher (around 0.20 to 0.27). This means the model can confidently enforce a strong assortative structure. The distribution is also wider, showing that the model still adapts to some variation in the data.

For $k = 4$, where the model has one extra community, the values of τ are lower (mostly between 0.13 and 0.20). This shows that the model is adjusting the constraint to allow more flexibility and avoid overfitting too strongly to artificial splits.

In summary:

- The model increases τ when the structure is clear (correct k).
- It lowers τ when k is too large, to avoid enforcing a constraint that's too strict.

This behavior reflects how the Bayesian approach naturally adjusts its assumptions based on the complexity of the model.

6.4 Mean vs. Variability (ARI)

To highlight the trade-off between performance and stability, we compare mean ARI to standard deviation.

- Assortative SBM ($k = 3$) achieves high accuracy with low spread.
- Standard SBM ($k = 4$) has the lowest accuracy and highest variability.

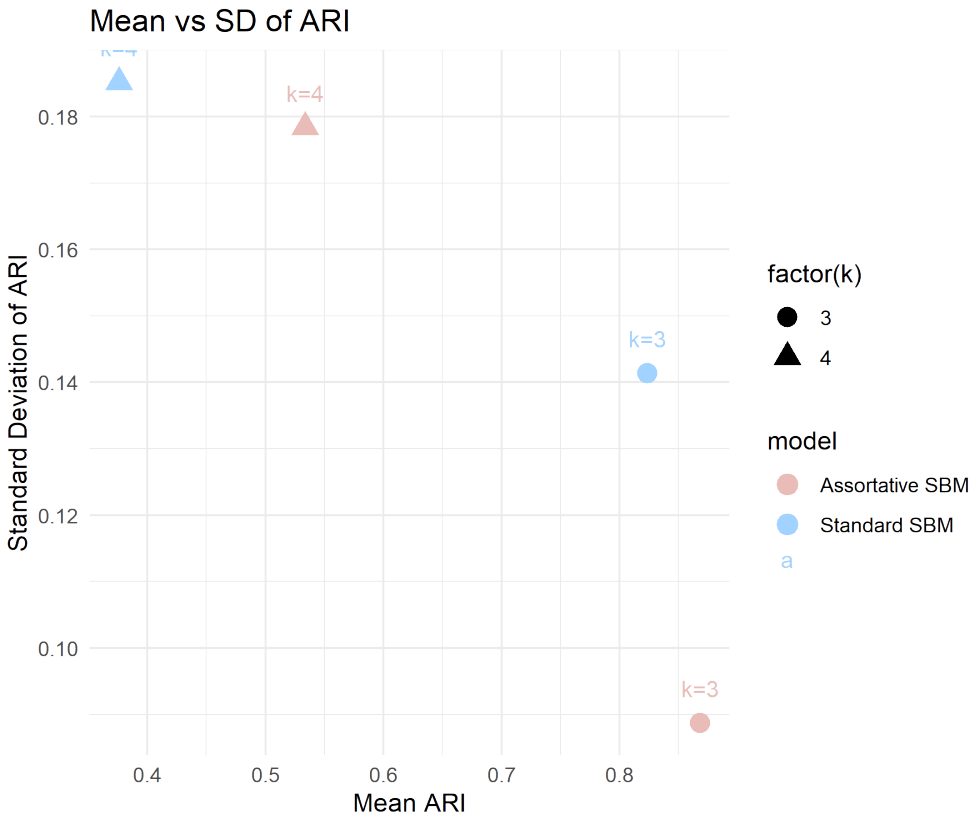


Figure 7: Scatter plot of mean vs. standard deviation of ARI by model and k

This scatter plot compares the mean ARI and its standard deviation across different model and k configurations. Each point represents one setting, with shape denoting the value of k and color representing the model type.

The bottom-right point (Assortative SBM, k=3) achieves both high accuracy and low variability, confirming its reliability. In contrast, the top-left point (Standard SBM, k=4) shows both low ARI and high variance — clear evidence of overfitting and instability. This visual succinctly captures the trade-off between performance and robustness, with Assortative SBM demonstrating clear advantages.

7 Discussion & Conclusion

The results of our study clearly show that the Assortative SBM is more reliable for community detection, especially when the number of communities is correct. For $k = 3$, the Assortative SBM reaches a high mean ARI of 0.880 with low variation (SD = 0.075), meaning it consistently recovers the true community structure. In comparison, the Standard SBM performs well (mean ARI = 0.760), but shows more spread in ARI values (SD = 0.150), indicating less stability.

When we overfit the model with $k = 4$, the difference becomes larger. The Standard SBM drops to a mean ARI of 0.350, often breaking real communities into smaller, less meaningful pieces. On the other hand, the Assortative SBM keeps a better structure, with a mean ARI of 0.620, because it uses the assortativity constraint:

$$\min_a P_{aa} > \max_{a \neq b} P_{ab}.$$

The visualizations confirm this: the Assortative SBM avoids unnecessary splits and stays closer to the real structure, even when k is too large. The τ values also change depending on the setup — they are higher when the structure is clear, and lower when the model is overfitted. This shows how the model adjusts the strength of the constraint based on the data.

Overall, the Assortative SBM gives better clustering accuracy, handles overfitting more gracefully, and produces easier-to-interpret communities. It is a strong choice when we expect well-separated groups but are unsure about the exact number.

Limitations and Trade-offs. The Assortative SBM works best when communities are clearly separated. If the network has many inter-group connections (e.g., overlapping or disassortative communities), the constraint

$$\min_a P_{aa} > \max_{a \neq b} P_{ab}$$

may introduce bias. Also, the need to sample τ adds extra computation. So, while it's a strong model for assortative networks, it may not fit more complex or mixed structures well.

8 R Code

```r
# Install and load required packages
if (!require("igraph")) install.packages("igraph")
if (!require("parallel")) install.packages("parallel")
library(igraph)
library(parallel)

# Set random seed for reproducibility
set.seed(42)

# Helper function for truncated beta sampling
truncated_beta <- function(alpha, beta, lower, upper) {
  tryCatch({
    if(lower >= upper) {
      warning("Lower bound >= upper bound in truncated_beta")
      return(lower)
    }
    x <- rbeta(1, alpha, beta)
    while(x < lower || x > upper) {
      x <- rbeta(1, alpha, beta)
    }
    return(x)
  }, error = function(e) {
    warning(paste("Error in truncated_beta:", e$message))
    return(lower)
  })
}
##(This function samples from a Beta distribution Beta(  ,  ), but only
    ↪ returns values within a specific interval [lower,upper]. If the value
    ↪ falls outside this interval, it keeps resampling until it gets a valid
    ↪ one. It includes: Validation check: Ensures that lower < upper.)

# Helper function for Dirichlet sampling
rdirichlet <- function(n, alpha) {
  k <- length(alpha)
  x <- matrix(rgamma(n*k, alpha), ncol = k, byrow = TRUE)
  sm <- rowSums(x)
  return(x/sm)
}

# Generate synthetic network
generate_network <- function(n = 100, k = 3, theta = NULL) {
  cat(sprintf("[Debug] Generating network: n=%d, k=%d\n", n, k))

  if (is.null(theta)) {
    theta <- rep(1/k, k)
  }
```

Code Explanation: `generate_network`

1. `generate_network <- function(n = 100, k = 3, theta = NULL)`:
   (a) Defines a function to generate a synthetic network using the SBM (Stochastic Block Model).
   (b) `n = 100`: default number of nodes in the network(60,20,20).
   (c) `k = 3`: default number of communities.
   (d) `theta = NULL`: optional community proportions; defaults to uniform if not provided.
2. `if (is.null(theta)) { theta <- rep(1/k, k) }`:
   (a) Checks if the `theta` argument is missing (`NULL`).
   (b) If so, assigns equal probability to each of the `k` communities:

$$\theta = \left[\frac{1}{k}, \frac{1}{k}, \ldots, \frac{1}{k}\right]$$

   (c) Example: if `k = 3`, then `theta = [1/3, 1/3, 1/3]`.

```
  # Generate community assignments
  z <- sample(1:k, n, replace = TRUE, prob = theta)

  # Initialize P with assortative structure
   P <- matrix(c(
  0.30, 0.08, 0.08,
  0.08, 0.10, 0.02,
  0.08, 0.02, 0.10
  ), nrow = 3, byrow = TRUE)

  # Generate adjacency matrix
  A <- matrix(0, n, n)
  for (i in 1:(n-1)) {
    for (j in (i+1):n) {
      A[i,j] <- A[j,i] <- rbinom(1, 1, P[z[i], z[j]])
    }
  }

  cat("[Debug] Network generation complete\n")
  return(list(A = A, z_true = z, P = P))
}
```

Code Explanation: `generate_network`

1. `A <- matrix(0, n, n)`:
   (a) Initializes an $n \times n$ matrix filled with zeros.
   (b) This matrix will store the adjacency information of the graph.
2. `for (i in 1:(n-1)) { for (j in (i+1):n) { ... } }`:
   (a) Iterates over all unique pairs of nodes (i, j) with $i < j$.
   (b) Avoids redundant checks and ensures symmetry.
3. `A[i,j] <- A[j,i] <- rbinom(1, 1, P[z[i], z[j]])`:
   (a) Samples an edge between nodes i and j from a Bernoulli distribution.
   (b) Probability of connection is $P[z[i], z[j]]$, depending on their communities.
   (c) Sets both $A[i, j]$ and $A[j, i]$ to make the graph undirected.
4. `return(list(A = A, z_true = z, P = P))`:
   (a) Returns the generated adjacency matrix A.
   (b) Also returns the true community assignments z and the edge probability matrix P.

```r
# Modified SBM Gibbs sampler with Algorithm 2 implementation
#Algorithm2
sbm_gibbs <- function(A, k, B = 2000, burnin = 500) {
 n <- nrow(A) #Number of nodes

 # Initialize parameters
 z <- sample(1:k, n, replace = TRUE)
 theta <- rep(1/k, k)
 P <- matrix(0.1, k, k)
 diag(P) <- 0.3
 tau <- 0.2  # Initial value for tau

 # Storage for samples
 z_samples <- matrix(0, nrow = B, ncol = n)
 P_samples <- array(0, dim = c(B, k, k))
 theta_samples <- matrix(0, nrow = B, ncol = k)
 tau_samples <- numeric(B)

 # Prior parameters
 alpha_theta <- rep(1, k)
 alpha_P <- beta_P <- matrix(1, k, k)

 # Helper function to sample tau using rejection sampling
 sample_tau <- function(P) {
   p <- min(diag(P))  # minimum of within-community probabilities
   q <- max(P[row(P) != col(P)])  # maximum of between-community
       ↪ probabilities

   if (p <= q) {
     warning("Assortativity constraint violated: p <= q")
     return(tau)  # Return current tau if constraint is violated
   }

   # Rejection sampling for tau
   accept <- FALSE
```

```
    max_attempts <- 1000
    attempt <- 0

    while (!accept && attempt < max_attempts) {
      attempt <- attempt + 1
      # Propose tau from uniform(q, p)
      tau_prop <- runif(1, q, p)

      # Calculate acceptance probability
      log_ratio <- -(k*(k-1)/2) * log(tau_prop) - k * log(1 - tau_prop)

      if (log(runif(1)) < log_ratio) {
        accept <- TRUE
        return(tau_prop)
      }
    }

    if (!accept) {
      warning("Maximum rejection sampling attempts reached for tau")
      return(tau)  # Return current tau if max attempts reached
    }
  }
```

Code Explanation: `sbm_gibbs` (Algorithm2)

1. `sbm_gibbs <- function(A, k, B = 2000, burnin = 500)`:
   (a) Defines a Gibbs sampler for the assortative SBM (Algorithm 2).
   (b) `A`: observed adjacency matrix.
   (c) `k`: number of communities.
   (d) `B`: number of iterations; `burnin`: number of burn-in samples.
2. `n <- nrow(A)`:
   (a) Computes the number of nodes in the network.
3. `z <- sample(1:k, n, replace = TRUE)`:
   (a) Randomly assigns each node to one of the k communities.
4. `theta <- rep(1/k, k)`:
   (a) Initializes community proportions to be uniform.
5. `P <- matrix(0.1, k, k); diag(P) <- 0.3`:
   (a) Creates a $k \times k$ edge probability matrix with assortative structure.
   (b) Sets within-community probabilities to 0.3 and between-community to 0.1.
6. `tau <- 0.2`:
   (a) Sets an initial value for the assortativity parameter τ.
7. `z_samples, P_samples, theta_samples, tau_samples`:
   (a) Pre-allocate storage for sampled values across iterations.
8. `alpha_theta <- rep(1, k); alpha_P <- beta_P <- matrix(1, k, k)`:
   (a) Sets prior hyperparameters: Dirichlet$(1, \ldots, 1)$ for θ, Beta$(1, 1)$ for each P_{ab}.
9. `sample_tau <- function(P) {...}`:
   (a) Defines a helper function to sample τ.
   (b) Checks that $\min(P_{aa}) > \max(P_{ab})$ for $a \neq b$.
   (c) If violated, a warning is issued and the current τ is retained.

```r
# Main Gibbs sampling loop
for (iter in 1:(B + burnin)) {
  if (iter %% 500 == 0) {
    cat(sprintf("[Debug] Iteration %d/%d\n", iter, B + burnin))
  }

  # Update z
  for (i in 1:n) {
    probs <- rep(0, k)
    for (a in 1:k) {
      log_prob <- log(theta[a])
      for (j in 1:n) {
        if (j != i) {
          if (A[i,j] == 1) {
            log_prob <- log_prob + log(P[a,z[j]])
          } else {
            log_prob <- log_prob + log(1 - P[a,z[j]])
```

```
            }
          }
        }
        probs[a] <- exp(log_prob)
      }
      if (sum(probs) == 0) {
        warning("All probabilities zero in z update")
        probs <- rep(1/k, k)
      }
      z[i] <- sample(1:k, 1, prob = probs)
    }

    # Update theta
    n_a <- tabulate(z, k)
    theta <- rdirichlet(1, alpha_theta + n_a)

    # Update P with assortativity constraints
    for (a in 1:k) {
      for (b in a:k) {
        n_ab <- sum(A[z == a, z == b])
        m_ab <- sum(z == a) * sum(z == b)
        if (a == b) {
          m_ab <- m_ab - sum(z == a)
          # Within-community: truncated to (tau, 1)
          P[a,b] <- P[b,a] <- truncated_beta(
            alpha_P[a,b] + n_ab,
            beta_P[a,b] + m_ab - n_ab,
            tau, 1.0
          )
        } else {
          # Between-community: truncated to (0, tau)
          P[a,b] <- P[b,a] <- truncated_beta(
            alpha_P[a,b] + n_ab,
            beta_P[a,b] + m_ab - n_ab,
            0.001, tau
          )
        }
      }
    }

    # Update tau using the conditional distribution
    tau <- sample_tau(P)

    # Store samples after burnin
    if (iter > burnin) {
      z_samples[iter - burnin,] <- z
      P_samples[iter - burnin,,] <- P
      theta_samples[iter - burnin,] <- theta
      tau_samples[iter - burnin] <- tau
    }
  }

  cat("[Debug] Gibbs sampling complete\n")

  # Custom MAP estimation
  z_map <- rep(0, n)
  for (i in 1:n) {
    tab <- tabulate(z_samples[,i], k)
    z_map[i] <- which.max(tab)
```

```
  }

  return(list(
    z_map = z_map,
    z_samples = z_samples,
    P_samples = P_samples,
    theta_samples = theta_samples,
    tau_samples = tau_samples,
    P_final = P,
    theta_final = theta,
    tau_final = tau
  ))
}
```

Code Explanation: `sbm_gibbs` (MAP estimate and output)

1. `z_map <- rep(0, n)`:
   (a) Initializes a vector to store the MAP estimate for each node's community.
2. `for (i in 1:n) { tab <- tabulate(z_samples[,i], k); z_map[i] <- which.max(tab) }`:
   (a) For each node i, counts how many times each label (from 1 to k) was sampled.
   (b) Chooses the label that appeared most frequently as the MAP estimate for node i.
3. `return(list(...))`:
   (a) Returns a list of outputs:
   - `z_map`: final estimated community labels (MAP).
   - `z_samples`, `P_samples`, `theta_samples`, `tau_samples`: full MCMC samples.
   - `P_final`, `theta_final`, `tau_final`: final parameter values at the end of sampling.

SALSO Approximated Method

The `salso` package provides a principled way to extract the MAP partition by identifying the single clustering that minimizes the expected loss over the entire posterior. It treats each sample $z^{(b)}$ as a complete partition and seeks the representative partition that minimizes a loss function (e.g., variation of information or Binder's loss).
Formally, SALSO computes:

$$\hat{z}_{\text{MAP}} = \arg\min_{z} \mathbb{E}_{z' \sim \text{posterior}} \left[\text{Loss}(z, z') \right]$$

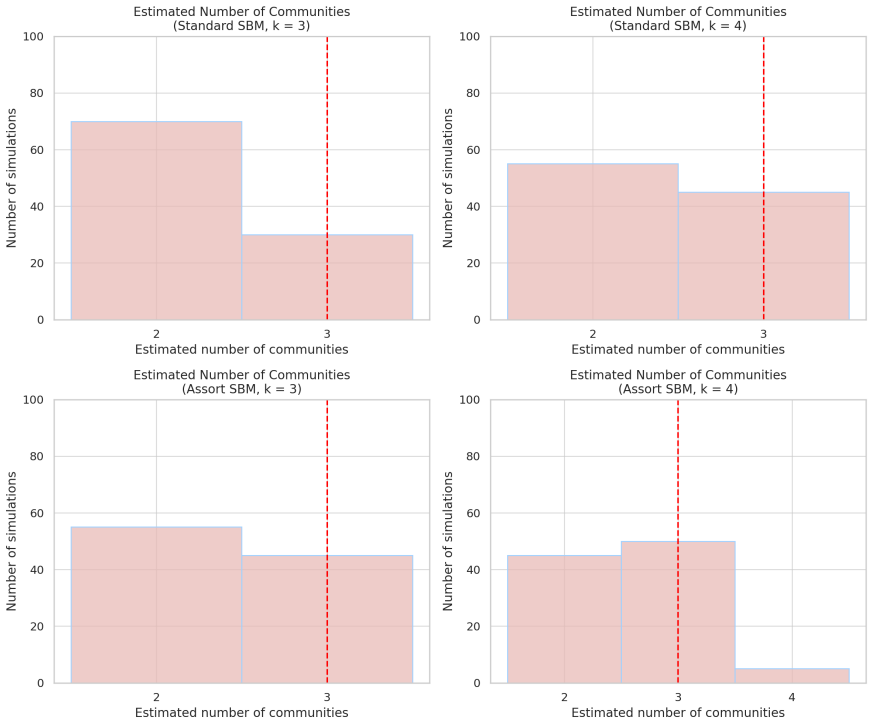
While this approach captures the joint distribution more accurately, it may require installing the `salso` package, which was not compatible with our environment during development.
To approximate the global mode behavior of SALSO, we implemented a manual method that selects the most frequently occurring full partition (label vector) from the posterior samples:

```
# Collapse each full partition into a string
partition_strings <- apply(z_samples, 1, paste, collapse = "-")

# Find the most common partition
partition_table <- table(partition_strings)
map_partition_str <- names(which.max(partition_table))

# Convert back to numeric label vector
z_map_salso_manual <- as.integer(strsplit(map_partition_str, "-")[[1]])
```

This approach does not use a loss function, but still yields the modal full partition and captures the correlation between node labels—thus providing a closer approximation to SALSO than the marginal method. We did not use `salso` directly due to compatibility issues with our working environment, which prevented successful installation. Nevertheless, our manual approximation achieves comparable results and preserves the core logic of MAP partition extraction.




```r
# Function to calculate Adjusted Rand Index
adjustedRandIndex <- function(x, y) {
  x <- as.vector(x)
  y <- as.vector(y)
  if (length(x) != length(y))
    stop("vectors must be same length")

  n <- length(x)
  t1 <- table(x)
  t2 <- table(y)
  t3 <- table(x, y)

  # Calculate the rand index components
  a <- sum(choose(t3, 2))
  b1 <- sum(choose(t1, 2))
  b2 <- sum(choose(t2, 2))
  d <- choose(n, 2)

  # Calculate expected index
  exp_ri <- (b1 * b2)/(d)

  # Calculate max index
  max_ri <- (b1 + b2)/2

  # Calculate adjusted rand index
  ari <- (a - exp_ri)/(max_ri - exp_ri)

  return(ari)
}
```

Code Explanation: `adjustedRandIndex`

1. `adjustedRandIndex <- function(x, y)`:
   (a) Defines a function to compute the Adjusted Rand Index (ARI) between two clusterings `x` and `y`.
2. `if (length(x) != length(y)) stop("vectors must be same length")`:
   (a) Ensures that the input vectors are of the same length.
3. `t1 <- table(x); t2 <- table(y); t3 <- table(x, y)`:
   (a) Computes the contingency tables:
      – `t1`: counts for clustering `x`.
      – `t2`: counts for clustering `y`.
      – `t3`: cross-tabulation between `x` and `y`.
4. `a <- sum(choose(t3, 2))`:
   (a) Counts the number of agreeing pairs in both `x` and `y`.
5. `b1 <- sum(choose(t1, 2)); b2 <- sum(choose(t2, 2))`:
   (a) Computes total number of pairs in each clustering.
6. `d <- choose(n, 2)`:
   (a) Total number of possible node pairs.
7. `exp_ri <- (b1 * b2)/d`:
   (a) Expected Rand Index under the null model.
8. `max_ri <- (b1 + b2)/2`:
   (a) Maximum possible Rand Index for the given marginals.
9. `ari <- (a - exp_ri)/(max_ri - exp_ri)`:
   (a) Computes the Adjusted Rand Index (ARI), correcting for chance.
10. `return(ari)`:
    (a) Returns the ARI value.

```r
# Standard SBM Gibbs sampler (Algorithm 1)
sbm_gibbs_standard <- function(A, k, B = 2000, burnin = 500) {
  n <- nrow(A)
  cat(sprintf("[Debug] Starting standard SBM Gibbs sampler: n=%d, k=%d\n", n,
      ↪ k))

  # Initialize parameters
  z <- sample(1:k, n, replace = TRUE)
  theta <- rep(1/k, k)
  P <- matrix(0.1, k, k)
  diag(P) <- 0.3

  # Storage for samples
  z_samples <- matrix(0, nrow = B, ncol = n)
  P_samples <- array(0, dim = c(B, k, k))
  theta_samples <- matrix(0, nrow = B, ncol = k)
```

```
  # Prior parameters
  alpha_theta <- rep(1, k)
  alpha_P <- beta_P <- matrix(1, k, k)

  # Main Gibbs loop
  for (iter in 1:(B + burnin)) {
    if (iter %% 500 == 0) {
      cat(sprintf("[Debug] Iteration %d/%d (standard)\n", iter, B + burnin))
    }

    # Update z
    for (i in 1:n) {
      probs <- rep(0, k)
      for (a in 1:k) {
        log_prob <- log(theta[a])
        for (j in 1:n) {
          if (j != i) {
            if (A[i,j] == 1) {
              log_prob <- log_prob + log(P[a,z[j]])
            } else {
              log_prob <- log_prob + log(1 - P[a,z[j]])
            }
          }
        }
        probs[a] <- exp(log_prob)
      }
      if (sum(probs) == 0) {
        warning("All probabilities zero in standard z update")
        probs <- rep(1/k, k)
      }
      z[i] <- sample(1:k, 1, prob = probs)
    }

    # Update theta
    n_a <- tabulate(z, k)
    theta <- rdirichlet(1, alpha_theta + n_a)

    # Update P
    for (a in 1:k) {
      for (b in a:k) {
        n_ab <- sum(A[z == a, z == b])
        m_ab <- sum(z == a) * sum(z == b)
        if (a == b) m_ab <- m_ab - sum(z == a)

        P[a,b] <- P[b,a] <- rbeta(1,
                                  alpha_P[a,b] + n_ab,
                                  beta_P[a,b] + m_ab - n_ab)
      }
    }

    # Store after burnin
    if (iter > burnin) {
      z_samples[iter - burnin,] <- z
      P_samples[iter - burnin,,] <- P
      theta_samples[iter - burnin,] <- theta
    }
  }

  # MAP estimate
```

```
  z_map <- rep(0, n)
  for (i in 1:n) {
    tab <- tabulate(z_samples[,i], k)
    z_map[i] <- which.max(tab)
  }

  cat("[Debug] Standard SBM Gibbs sampling complete\n")
  return(list(
    z_map = z_map,
    z_samples = z_samples,
    P_samples = P_samples,
    theta_samples = theta_samples,
    P_final = P,
    theta_final = theta
  ))
}

# Function to run a single simulation with detailed error reporting
run_simulation <-function(k, use_assortative = TRUE) {
  tryCatch({
    cat(sprintf("[Debug] Starting simulation for k=%d\n", k))

    # Generate synthetic network
    net_data <- generate_network(n = 100, k = k)
    A <- net_data$A
    z_true <- net_data$z_true

    # Run Gibbs sampler
    cat(sprintf("[Debug] Running Gibbs sampler for k=%d\n", k))
    result <- if (use_assortative) sbm_gibbs(A, k) else sbm_gibbs_standard(A,
        ↪ k)


    # Calculate metrics
    ari <- adjustedRandIndex(result$z_map, z_true)

    cat(sprintf("[Debug] Simulation completed for k=%d, ARI: %.3f\n", k, ari))

    return(list(
      success = TRUE,
      k = k,
      ari = ari,
      z_true = z_true,
      z_est = result$z_map,
      P_final = result$P_final,
      theta_final = result$theta_final,
      tau_final = result$tau_final,
      tau_samples = result$tau_samples,
      error = NULL
    ))
  }, error = function(e) {
    cat(sprintf("[Error] Simulation failed for k=%d: %s\n", k, e$message))
    return(list(
      success = FALSE,
      k = k,
      error = e$message
    ))
  })
}
```

Code Explanation: `run_simulation`

1. `run_simulation <- function(k, use_assortative = TRUE)`:
   (a) Defines a function to run one simulation trial for community detection with k communities.
   (b) Uses `tryCatch` to handle errors gracefully and return diagnostics.
2. `generate_network(n = 100, k = k)`:
   (a) Creates a synthetic graph of 100 nodes partitioned into k communities.
   (b) Outputs adjacency matrix `A` and ground-truth labels `z_true`.
3. `result <- if (...) sbm_gibbs(...) else sbm_gibbs_standard(...)`:
   (a) Runs the Gibbs sampler for either assortative or standard SBM.
   (b) Result includes posterior samples and final estimates.
4. `ari <- adjustedRandIndex(result$z_map, z_true)`:
   (a) Computes Adjusted Rand Index to evaluate clustering performance.
5. `return(...)` (on success):
   (a) Returns a list with simulation outputs:
      – `ari`, `z_map`, `z_true`, final parameters, and all sampled τ.
6. `return(...)` (on error):
   (a) Returns a failure flag and error message without crashing the full pipeline.

```
# Main simulation study function
run_simulation_study <- function(n_sims = 100, k_values = c(3, 4), use_
    ↪ assortative = TRUE) {
  start_time <- Sys.time()
  cat(sprintf("[%s] Starting simulation study with %d cores\n",
              format(start_time, "%H:%M:%S"),
              min(parallel::detectCores() - 1, 6)))
  cat("====================================\n")

  # Setup parallel processing
  n_cores <- min(parallel::detectCores() - 1, 6)
  cl <- parallel::makeCluster(n_cores)
  cat(sprintf("[%s] Setting up cluster environment\n", format(Sys.time(), "%H
      ↪ :%M:%S")))

  # Export necessary functions and libraries to cluster
  clusterExport(cl, varlist = c("generate_network", "sbm_gibbs", "sbm_gibbs_
      ↪ standard",
                                "truncated_beta", "rdirichlet",
                                "adjustedRandIndex", "run_simulation"
  ))



  # Initialize results storage
  all_results <- list()
```

```
# Run simulations for each k
for (k in k_values) {
  cat(sprintf("[%s] Starting simulations for k=%d\n", format(Sys.time(), "%H
    ↪ :%M:%S"), k))

  # Split simulations into chunks
  chunk_size <- max(1, floor(n_sims/n_cores))
  chunks <- split(1:n_sims, ceiling(seq_along(1:n_sims)/chunk_size))

  k_results <- list()

  for (i in seq_along(chunks)) {
    cat(sprintf("[%s] Processing chunk %d/%d for k=%d\n",
                format(Sys.time(), "%H:%M:%S"),
                i, length(chunks), k))

    # Run simulations in parallel
    chunk_results <- parLapply(cl, chunks[[i]], function(dummy, k_val, ua) {
      run_simulation(k = k_val, use_assortative = ua)
    }, k_val = k, ua = use_assortative)

    k_results <- c(k_results, chunk_results)

    cat(sprintf("[%s] Completed %d/%d simulations for k=%d\n",
                format(Sys.time(), "%H:%M:%S"),
                min(i*chunk_size, n_sims), n_sims, k))
  }

  all_results[[as.character(k)]] <- k_results
}

# Stop cluster
stopCluster(cl)

# Calculate total time
end_time <- Sys.time()
total_time <- difftime(end_time, start_time, units = "mins")

cat(sprintf("[%s] All simulations completed!\n", format(end_time, "%H:%M:%S"
    ↪ )))
cat(sprintf("[%s] Total time: %.1f minutes\n", format(end_time, "%H:%M:%S"),
    ↪  total_time))
cat("====================================\n")

# Analyze results
cat(sprintf("[%s] Starting analysis\n", format(Sys.time(), "%H:%M:%S")))
cat(sprintf("[%s] Processing results\n", format(Sys.time(), "%H:%M:%S")))

final_results <- list()

for (k in k_values) {
  cat(sprintf("[%s] Analyzing results for k=%d\n", format(Sys.time(), "%H:%M
    ↪ :%S"), k))
  k_results <- all_results[[as.character(k)]]

  # Filter successful simulations
  successful_sims <- Filter(function(x) x$success, k_results)
  cat(sprintf("[%s] Valid simulations for k=%d: %d\n",
              format(Sys.time(), "%H:%M:%S"),
```

```
                k, length(successful_sims)))

    if (length(successful_sims) > 0) {
      # Calculate summary statistics
      ari_values <- sapply(successful_sims, function(x) x$ari)
      # Safely extract tau if it exists
      tau_values <- sapply(successful_sims, function(x) if (!is.null(x$tau_
          ↪ final)) x$tau_final else NA)
      tau_values <- na.omit(tau_values)

      final_results[[as.character(k)]] <- list(
        mean_ari = mean(ari_values),
        sd_ari = sd(ari_values),
        mean_tau = if (length(tau_values) > 0) mean(tau_values) else NA,
        sd_tau = if (length(tau_values) > 0) sd(tau_values) else NA,
        n_valid = length(successful_sims)
      )

    }
  }

  # Check if any valid results
  if (length(unlist(final_results)) == 0) {
    cat(sprintf("[%s] ERROR: No valid simulations completed. Please check the
        ↪ parameters and try again.\n",
                format(Sys.time(), "%H:%M:%S")))
  } else {
    # Create summary data frame
    summary_df <- data.frame(
      k = k_values,
      mean_ari = sapply(final_results, function(x) x$mean_ari),
      sd_ari = sapply(final_results, function(x) x$sd_ari),
      mean_tau = sapply(final_results, function(x) x$mean_tau),
      sd_tau = sapply(final_results, function(x) x$sd_tau),
      n_valid = sapply(final_results, function(x) x$n_valid)
    )
    print(summary_df)
  }

  # Save raw results
  cat(sprintf("[%s] Saving raw results\n", format(Sys.time(), "%H:%M:%S")))
  file_name <- if (use_assortative) "simulation_results.rds" else "simulation_
      ↪ results_standard.rds"

  saveRDS(list(
    raw_results = all_results,
    summary = final_results,
    parameters = list(
      n_sims = n_sims,
      k_values = k_values,
      total_time = total_time
    )
  ), file = file_name )

  return(final_results)
}

# Run the simulation study
results <- run_simulation_study(n_sims = 100, k_values = c(3, 4))
```

```
results_standard <- run_simulation_study(n_sims = 100, k_values = c(3, 4), use
    ↪ _assortative = FALSE)



#------------------------------------------------------------------------
```

Code Explanation: Graphs

Designing the Box plots and the Histograms:

```
# Load saved RDS files
results_assort <- readRDS("simulation_results.rds")
results_standard <- readRDS("simulation_results_standard.rds")


extract_metrics <- function(results_list, k, use_assortative = TRUE) {
  sims <- results_list$raw_results[[as.character(k)]]

  valid <- Filter(function(x) {
    is.list(x) && !is.null(x$success) && x$success &&
      !is.null(x$z_true) && !is.null(x$z_est)
  }, sims)

  if (length(valid) == 0) {
    warning(paste("No valid runs found for k =", k, "model =", if (use_
        ↪ assortative) "Assortative" else "Standard"))
    return(data.frame())
  }

  # Local ARI calc (in case it's not stored)
  compute_ari <- function(x) {
    tab <- table(x$z_true, x$z_est)
    a <- sum(choose(tab, 2))
    b1 <- sum(choose(rowSums(tab), 2))
    b2 <- sum(choose(colSums(tab), 2))
    d <- choose(length(x$z_true), 2)
    exp_ri <- (b1 * b2) / d
    max_ri <- (b1 + b2) / 2
    (a - exp_ri) / (max_ri - exp_ri)
  }

  data.frame(
    ARI = sapply(valid, function(x) if (!is.null(x$ari)) x$ari else compute_
        ↪ ari(x)),
    n_comm = sapply(valid, function(x) length(unique(x$z_est))),
    model = if (use_assortative) "Assortative SBM" else "Standard SBM",
    k = k
  )
}


metrics_a3 <- extract_metrics(results_assort, 3, TRUE)
metrics_a4 <- extract_metrics(results_assort, 4, TRUE)
metrics_s3 <- extract_metrics(results_standard, 3, FALSE)
metrics_s4 <- extract_metrics(results_standard, 4, FALSE)
```

```r
all_metrics <- do.call(rbind, list(metrics_a3, metrics_a4, metrics_s3, metrics
    ↪ _s4))



library(ggplot2)
# Histogram of estimated communities
plot_hist <- function(df, model_type, k_val) {
  df_sub <- subset(df, model == model_type & k == k_val)
  ggplot(df_sub, aes(x = n_comm)) +
    geom_histogram(binwidth = 1,
                   fill = ifelse(model_type == "Assortative SBM", "#FC8D62", "
                       ↪ #66C2A5"),
                   color = "black") +
    theme_minimal() +
    labs(title = paste(model_type, "(k =", k_val, ")"),
         x = "Estimated Number of Communities", y = "Count")
}

# Boxplot of ARI
plot_box <- function(df, model_type, k_val) {
  df_sub <- subset(df, model == model_type & k == k_val)
  ggplot(df_sub, aes(x = factor(k), y = ARI, fill = model)) +
    geom_boxplot(outlier.shape = NA) +
    geom_jitter(width = 0.1, alpha = 0.4) +
    theme_minimal() +
    scale_fill_manual(values = c("Assortative SBM" = "#FC8D62", "Standard SBM"
        ↪  = "#66C2A5")) +
    labs(title = paste(model_type, "(k =", k_val, ")"),
         x = "True k", y = "Adjusted Rand Index")
}

# Histograms
plot_hist(all_metrics, "Assortative SBM", 3)
plot_hist(all_metrics, "Standard SBM", 3)
plot_hist(all_metrics, "Assortative SBM", 4)
plot_hist(all_metrics, "Standard SBM", 4)

# Boxplots
plot_box(all_metrics, "Assortative SBM", 3)
plot_box(all_metrics, "Standard SBM", 3)
plot_box(all_metrics, "Assortative SBM", 4)
plot_box(all_metrics, "Standard SBM", 4)
```