

A Bayesian Model for Committee Selection in Blockchain Consensus Protocols

Rob Jones

April 1, 2025

1 Introduction

This paper presents a Bayesian model for committee selection in blockchain consensus protocols, with a focus on estimating participant selection probabilities and optimizing committee composition. Key contributions include: (1) a hierarchical Bayesian framework using Dirichlet priors and multinomial sampling to model participant selection; (2) mathematical derivations for expected seat counts and their variances; (3) analysis of Byzantine Fault Tolerance (BFT) thresholds in the context of weighted participants; (4) an optimization approach for committee size determination based on stake distribution; and (5) application of Chernoff bounds to quantify the probability of Byzantine coalitions exceeding critical thresholds. The paper demonstrates how Bayesian inference can enhance our understanding of committee selection mechanics and improve security guarantees in blockchain systems where participants have heterogeneous influences based on stake.

Blockchain consensus algorithms often require committees of participants to validate transactions, contribute to governance, and ensure system reliability. In this context, Stake Pool Operators (SPOs) are grouped into committees of size k , drawn from a pool of n participants. Uniquely, participants can occupy multiple seats in the committee due to the sampling process, which introduces randomness into the selection mechanism. This randomness can be effectively modeled using hierarchical Bayesian inference, leveraging Dirichlet priors for the probability distribution of participants and posterior updates to refine predictions about committee seat assignments.

Committee members are chosen through random sampling with replacement from a pool of participants, and the probability of each participant being selected is governed by a probability distribution that is aligned with the multinomial distribution. Here's why:

- The **multinomial distribution** is an extension of the binomial distribution and deals with outcomes of experiments where there are more than two categories (or participants, in this case).
- Each “draw” or “trial” corresponds to a selection, and since sampling is done *with replacement*, the probability distribution over participants remains constant across trials.
- The result is that the counts of how many times each participant is chosen follow a multinomial distribution.

This paper presents a Bayesian model for estimating the selection probabilities of participants in a committee selection process, using Dirichlet priors and multinomial sampling. The model allows

for the incorporation of prior beliefs about participant selection probabilities and updates these beliefs based on observed data.

2 The Problem Setup

The committee selection process is described as follows:

1. **Participants (n):**
 - Represent distinct SPOs in the blockchain consensus algorithm.
 - Each participant has an associated probability of selection.
2. **Committee Size (k):**
 - Defines the number of seats in the committee.
 - Sampling is conducted *with replacement*, meaning a participant can occupy multiple seats.
3. **Stake Values:**
 - Each participant has a normalized stake value, influencing their probability of selection.

This process involves generating a probability distribution \mathbf{p} over the participants, sampling k seats based on \mathbf{p} , and iteratively refining our understanding of the probabilities \mathbf{p} using Bayesian inference.

3 Marginalizing Over \mathbf{p}

In the context of the multinomial distribution, we often have a prior belief about the probabilities \mathbf{p} of selecting each participant. This prior is typically modeled using a Dirichlet distribution, which is conjugate to the multinomial distribution. We want to treat the observed counts \mathbf{X} as a summary of the process and integrate over the unknown probabilities \mathbf{p} entirely. That makes sense if we're aiming for a fully Bayesian approach to marginalize out \mathbf{p} , rather than conditioning directly on it.

4 Model Specification

Given observed committee counts \mathbf{X} , we can perform posterior inference for the Dirichlet hyperparameters $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_n)$ by adopting a fully Bayesian approach.

In this case, the likelihood function is the multinomial probability mass function $P(\mathbf{X}|\mathbf{p})$, but our interest lies in the *marginal likelihood*, which integrates out \mathbf{p} by combining it with the prior distribution (Dirichlet). This results in a **Dirichlet-Multinomial** distribution, also known as the **Polya distribution**. The reasoning is as follows:

We place a prior on $\boldsymbol{\alpha}$ (for example, using a Gamma distribution for each α_i) and estimate it using Markov Chain Monte Carlo (MCMC) or variational Bayes inference. The MCMC approach can be implemented using the Python library PyMC. The problem is described mathematically with the following components:

1. Prior on Hyperparameters

In this framework, we assign each α_i a Gamma prior:

$$\alpha_i \sim \text{Gamma}(\text{shape}, \text{rate}),$$

where the "shape" and "rate" parameters reflect our prior beliefs about the concentration of each participant's selection probability.

2. Dirichlet Prior

A participant's probability of being selected is modeled as

$$\mathbf{p} = (p_1, p_2, \dots, p_n) \sim \text{Dirichlet}(\boldsymbol{\alpha}), \quad (1)$$

where

$$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_n)$$

are concentration parameters representing prior beliefs about the selection probabilities.

3. Sampling from the Committee

The number of times each participant occupies a seat in the committee is modeled as

$$\mathbf{X} = (X_1, X_2, \dots, X_n) \sim \text{Multinomial}(k, \mathbf{p}), \quad (2)$$

where \mathbf{X} represents the counts for each participant and $k = \sum_{i=1}^n X_i$ is the total committee size.

4. Bayesian Updating

After observing committee compositions \mathbf{X} , the prior $\boldsymbol{\alpha}$ is updated to a posterior distribution:

$$p(\boldsymbol{\alpha} \mid \mathbf{X}) \propto p(\mathbf{X} \mid \boldsymbol{\alpha}) \cdot p(\boldsymbol{\alpha}). \quad (3)$$

5. Posterior Inference

With these conjugate priors¹ and updates in place, MCMC methods are used to sample from the posterior distribution $p(\boldsymbol{\alpha} \mid \mathbf{X})$. This sampling process allows us to estimate the full posterior distributions for $\boldsymbol{\alpha}$ and, consequently, enables the computation of expected committee seat counts and their variances. In particular, the expected count for participant i is given by:

$$E[X_i] = k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha_j},$$

where k is the total number of committee seats. This Bayesian procedure thoroughly captures the uncertainty in the selection probabilities, providing a robust foundation for further inference and decision-making.

¹Conjugate priors simplify Bayesian inference by ensuring that the posterior distribution is in the same family as the prior, which makes analytical and computational procedures more tractable.

5 Mathematical Insights

Expected Seat Count

The expected number of seats assigned to participant i is computed as:

$$E[X_i] = k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}. \quad (4)$$

Variance of Seat Count

The variance of the seat count for participant i is given by:

$$\begin{aligned} \text{Var}(X_i) = k \left(\frac{\alpha_i}{\sum_{j=1}^n \alpha_j} - \frac{\alpha_i(\alpha_i + 1)}{\left(\sum_{j=1}^n \alpha_j\right) \left(\sum_{j=1}^n \alpha_j + 1\right)} \right) \\ + k^2 \cdot \frac{\alpha_i \left(\sum_{j=1}^n \alpha_j - \alpha_i\right)}{\left(\sum_{j=1}^n \alpha_j\right)^2 \left(\sum_{j=1}^n \alpha_j + 1\right)}. \end{aligned} \quad (5)$$

6 Implementation

The following script outlines the steps for Bayesian Committee Selection using Dirichlet priors and multinomial sampling. The algorithm is designed to be flexible, allowing for the incorporation of prior beliefs about participant selection probabilities and updating these beliefs based on observed data.

```
1 import pymc as pm
2 import numpy as np
3 import arviz as az
4
5 # Observed data (committee seat counts)
6 # where each instance sums to the total committee size, 50
7 observed_counts = np.array(
8     [[5, 10, 7, 13, 15],
9      [5, 10, 8, 12, 15],
10     [8, 12, 15, 8, 7]]
11 ) # Replace with actual data
12
13 k = observed_counts.sum(axis=1) # total committee size
14
15 # Bayesian Model
16 with pm.Model() as model:
17     # Gamma prior on Dirichlet parameters (alpha)
18     alpha = pm.Gamma(
19         "alpha", alpha=2.0, beta=1.0, shape=n.categories
20     )
21     # Dirichlet distribution for probabilities
22     p = pm.Dirichlet("p", a=alpha)
23
24     # Multinomial likelihood for observed data
25     observed = pm.Multinomial(
```

```

26     "observed", k, p=p, observed=observed_counts
27 )
28 # MCMC sampling
29 trace = pm.sample(
30     2000, tune=1000, return_inferencedata=True
31 )
32
33 # Posterior Analysis
34 posterior_mean_alpha = (
35     trace.posterior["alpha"]
36     .mean(dim=("chain", "draw"))
37     .values
38 )
39 print("Posterior Mean of Alpha:", posterior_mean_alpha)
40 expected_seat_counts = committee_size * \
41     (posterior_mean_alpha / posterior_mean_alpha.sum())
42 print("Expected Seat Counts:", expected_seat_counts)
43
44 # Visualize posterior distributions
45 az.plot_trace(trace, var_names=["alpha"])

```

Listing 1: Bayesian Model for Committee Seat Counts

7 Results and Discussion

The complete implementation of the algorithm is provided in Appendix D and is available in the repository as `bayes_stake_dist.py`. The code uses the PyMC library for Bayesian modeling and sampling, and ArviZ for posterior analysis and visualization. The model allows for flexible prior specifications and can be adapted to different committee selection scenarios.

The model's output provides a posterior distribution of the Dirichlet parameters, which can be interpreted as the expected number of times each participant is selected to the committee. The posterior mean of the Dirichlet parameters gives us the expected seat counts for each participant, which can be used to inform decisions about committee composition and selection mechanisms. The results of the Bayesian model can be summarized in Appendix E, Table E, which provides insights into the expected seat counts for each participant based on their prior beliefs and observed data. The table lists the mean probability distribution, $\mathbf{m} = [m_1, m_2, \dots, m_n]$, for columns $1, 2, \dots, n$ over the probabilities of committee selection, \mathbf{p} , for different stake sizes and group sizes, n . These parameters can be multiplied by the committee size k to calculate the expected number of times the associated participant is selected to the committee.

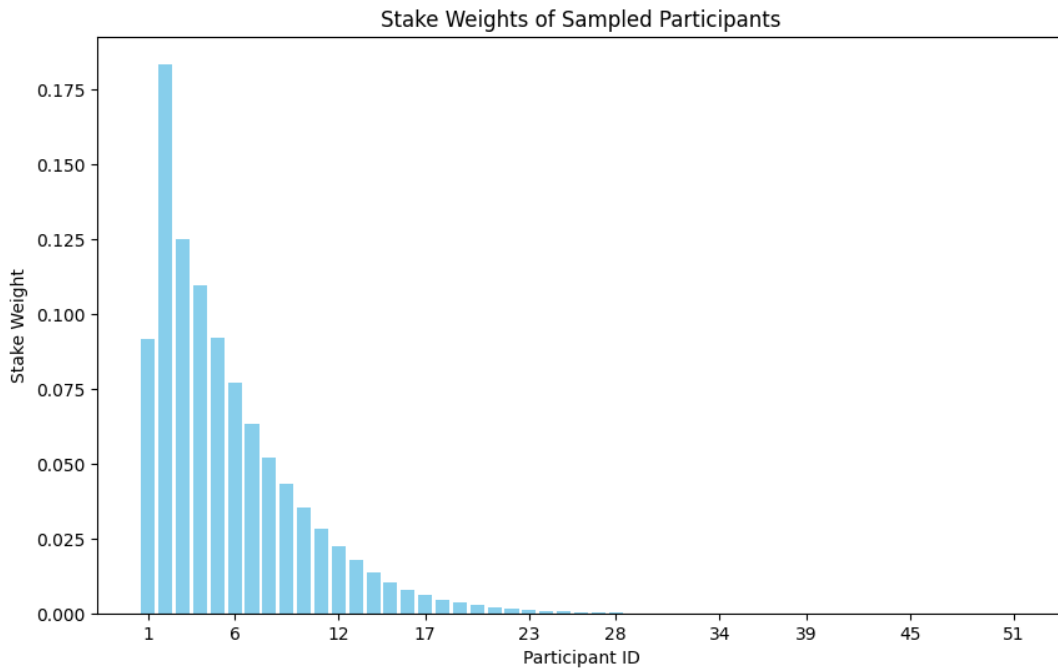


Figure 1: Distribution of stake weights across participants in a typical stake pool group. The x-axis represents individual participants, while the y-axis shows their normalized stake weight. This visualization demonstrates the heterogeneity in stake distribution, which directly influences the probability of selection to the committee according to our Bayesian model.

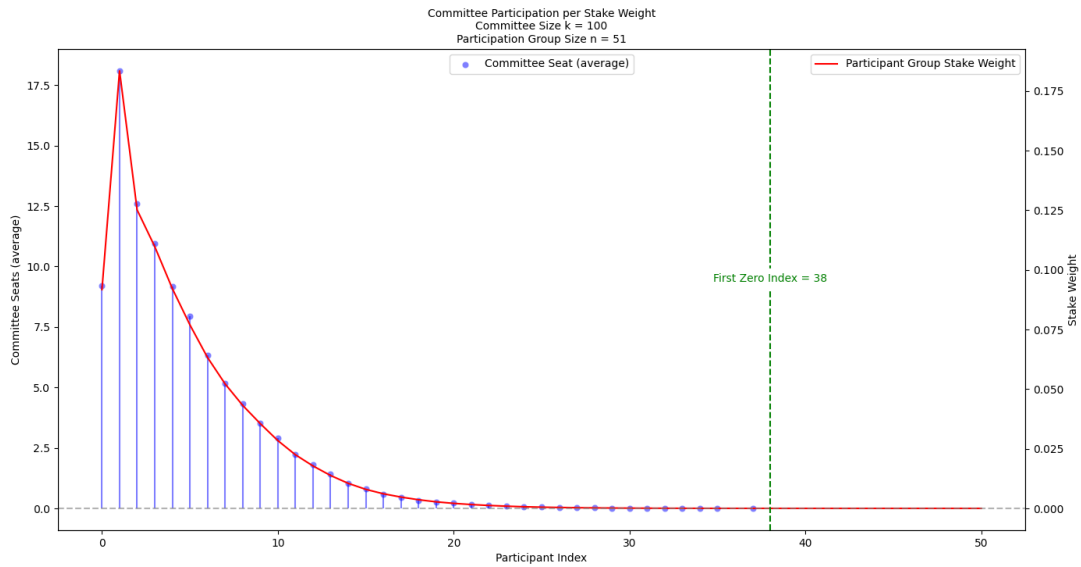


Figure 2: Committee participation of size $k = 100$ per stake weight sampled from a typical group of 50. The x-axis represents the stake weight of each participant, while the y-axis shows the number of times each participant is selected to the committee. The plot illustrates how the stake weight influences the likelihood of selection, with higher stake weights leading to more frequent committee participation.

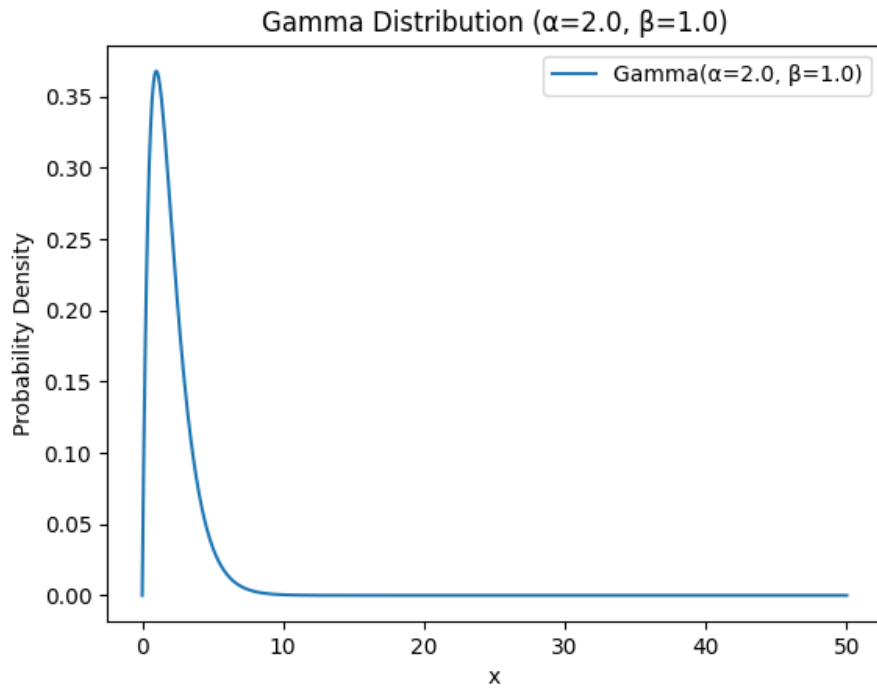


Figure 3: Gamma distribution used as the prior on the Dirichlet concentration parameters. The plot illustrates how the Gamma distribution with shape parameter $\alpha = 2.0$ and rate parameter $\beta = 1.0$ controls the prior beliefs about the concentration parameters of the Dirichlet distribution. This hyperprior influences the expected variation in selection probabilities among participants, with higher concentration values leading to more uniform distributions.

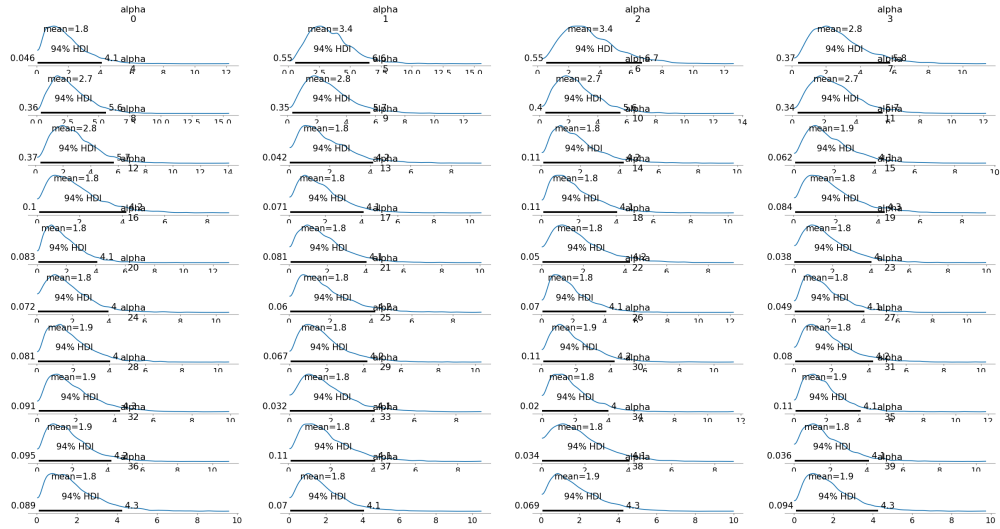


Figure 4: Bayesian update of Dirichlet parameters based on observed committee selection data. The plot illustrates how the prior distribution (blue) is updated through the likelihood of observed data to form the posterior distribution. This visualization demonstrates the core principle of our approach: as committee selections are observed, our belief about the underlying selection probabilities evolves, leading to more refined estimates of participant influence. The updated (posterior) Dirichlet parameters provide the foundation for calculating expected seat counts and their variances in future committee selections.

TODO: make a better plot of this.

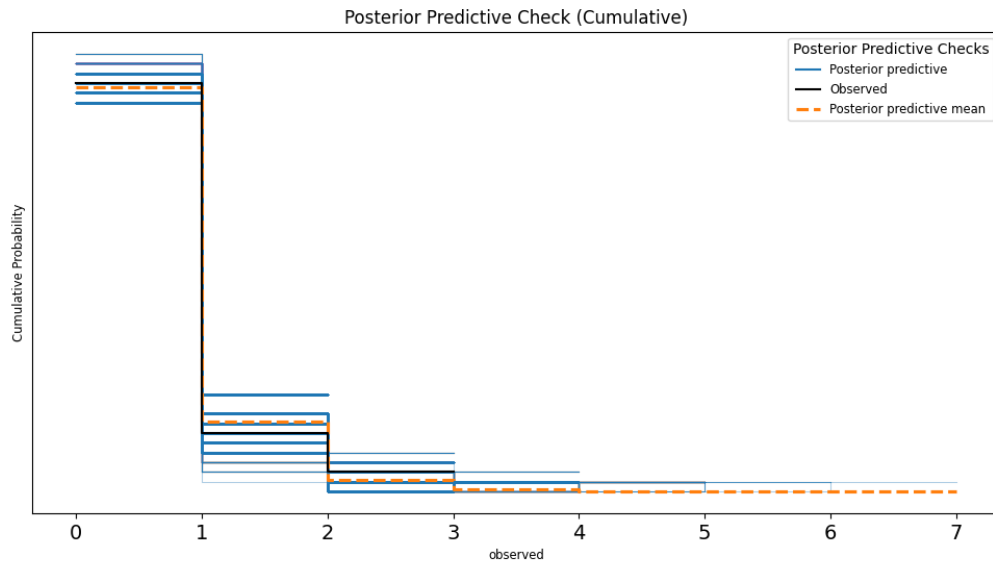


Figure 5: Posterior predictive check of the Bayesian model. This visualization compares data simulated from the posterior distribution of the model against the observed data, allowing assessment of model fit. The densities represent the distribution of committee seat allocations under the model's posterior (blue and orange lines), while the observed data (black line) should fall within the high-density regions if the model is capturing the underlying process well. Good alignment between simulated and observed data indicates that the model effectively captures the stake-based selection dynamics in the committee formation process.

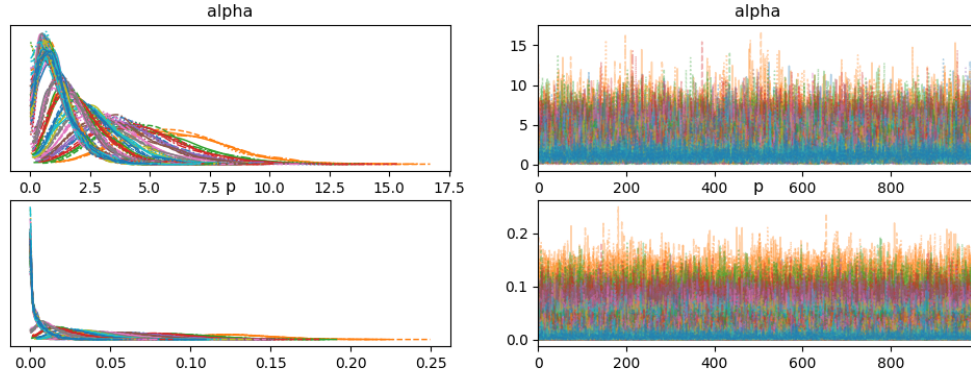


Figure 6: Bayesian inferencing process for the Dirichlet distribution parameters. The figure illustrates the Markov Chain Monte Carlo (MCMC) simulation used to calculate the α vector parameters. Left panels show the trace plots of individual α_i parameters across MCMC iterations, demonstrating convergence of the chains to the posterior distribution. Right panels display the posterior distributions of these parameters after convergence. This visualization captures how our model updates beliefs about participant selection probabilities through iteration, starting from prior assumptions and incorporating observed committee selection data to form reliable posterior estimates of the Dirichlet concentration parameters.

8 Deeper Dive on the Fully Bayesian Approach

8.1 Why Fully Bayesian?

In the context of the multinomial distribution, a fully Bayesian approach allows us to treat the selection probabilities \mathbf{p} as random variables rather than fixed parameters. This means we can incorporate prior beliefs about these probabilities and update them as we observe data. This is particularly useful when we have limited data or when we want to account for uncertainty in our estimates. By using a Dirichlet prior on \mathbf{p} , we can model the selection probabilities as a distribution rather than point estimates. This allows us to capture the uncertainty in our estimates and make probabilistic statements about the selection process. The Dirichlet distribution is a conjugate prior for the multinomial distribution, meaning that if we assume a Dirichlet prior on \mathbf{p} , the posterior distribution of \mathbf{p} given observed counts \mathbf{X} will also be a Dirichlet distribution. This property simplifies the Bayesian updating process and allows us to easily compute posterior distributions.

This means that the concentration parameters α_i can be used in the multinomial model to calculate probabilities \mathbf{p} interpreted as prior counts or pseudo-observations for each participant. For example, if we have $\alpha_1 = 2$, $\alpha_2 = 3$, and $\alpha_3 = 1$, this implies that we have prior beliefs of having observed 2, 3, and 1 counts for participants 1, 2, and 3, respectively. This allows us to incorporate prior knowledge or beliefs about the selection probabilities into our model. The posterior distribution of \mathbf{p} given observed counts \mathbf{X} will be a Dirichlet distribution with updated concentration parameters:

$$\alpha_{\text{post}} = \alpha_{\text{prior}} + \mathbf{X} = (\alpha_1 + x_1, \alpha_2 + x_2, \dots, \alpha_n + x_n). \quad (6)$$

and then the posterior becomes the prior for the next iteration (because of the conjugate prior

property):

$$\boldsymbol{\alpha}_{\text{prior}} \leftarrow \boldsymbol{\alpha}_{\text{post}} \quad (7)$$

This means that the posterior distribution of \mathbf{p} will be a Dirichlet distribution with parameters $\boldsymbol{\alpha}_{\text{post}}$, which can be interpreted as having observed x_i additional counts for each participant. This Bayesian updating process allows us to refine our estimates of the selection probabilities as we observe more data. The posterior distribution can be used to make probabilistic statements about the selection process, such as predicting the likelihood of selecting a particular participant in future trials. This is particularly useful in scenarios where we want to make decisions based on uncertain probabilities, such as in committee selection processes or other decision-making contexts.

Once we infer the Dirichlet distribution parameters $\boldsymbol{\alpha}$ we can marginalize out \mathbf{p} and compute the marginal likelihood of the observed data \mathbf{X} . This is done by integrating over the Dirichlet distribution, which leads to the Dirichlet-Multinomial distribution. The steps are as follows:

1. **Prior:**

$$\mathbf{p} \sim \text{Dirichlet}(\boldsymbol{\alpha}), \quad \text{with } \boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_n).$$

2. **Likelihood:** Given \mathbf{p} , the likelihood for \mathbf{X} (committee counts) is

$$P(\mathbf{X} | \mathbf{p}) = \frac{k!}{x_1!x_2!\dots x_n!} \prod_{i=1}^n p_i^{x_i}, \quad (8)$$

where $\mathbf{X} = (x_1, x_2, \dots, x_n)$ and $k = \sum_{i=1}^n x_i$ is the total number of selections.

3. **Integrate Over \mathbf{p} :** Combine the prior and likelihood, then integrate \mathbf{p} out:

$$P(\mathbf{X} | \boldsymbol{\alpha}) = \int_{\mathbf{p}} P(\mathbf{X} | \mathbf{p}) P(\mathbf{p} | \boldsymbol{\alpha}) d\mathbf{p}. \quad (9)$$

The resulting marginal likelihood is given by the Dirichlet-Multinomial distribution:

$$P(\mathbf{X} | \boldsymbol{\alpha}) = \frac{k!}{x_1!x_2!\dots x_n!} \frac{\Gamma(\sum_{i=1}^n \alpha_i)}{\Gamma(k + \sum_{i=1}^n \alpha_i)} \prod_{i=1}^n \frac{\Gamma(x_i + \alpha_i)}{\Gamma(\alpha_i)}. \quad (10)$$

This marginal likelihood captures the uncertainty in the selection probabilities and allows us to make probabilistic statements about the committee selection process. The Dirichlet-Multinomial distribution is particularly useful in scenarios where we want to make decisions based on uncertain probabilities, such as in committee selection processes or other decision-making contexts.

Key Advantages of This Approach

- **No Direct Dependence on \mathbf{p} :** By marginalizing \mathbf{p} , the model directly considers the uncertainty in \mathbf{p} through the Dirichlet prior.
- **Closed-Form Marginal Likelihood:** The Dirichlet-Multinomial distribution gives a tractable form for the marginal likelihood, avoiding the need for numerical integration.
- **Flexibility for Hyperparameter Estimation:** If we're uncertain about $\boldsymbol{\alpha}$, we can place a prior on it (e.g., $\alpha_i \sim \text{Gamma}$) and use MCMC to estimate it from the data.

To recap, why do we need a fully Bayesian approach for our multinomial committee selection model? Here are several compelling reasons:

- **Handling Parameter Uncertainty:** In most real-world scenarios, the true selection probabilities, represented by \mathbf{p} , are unknown. A Bayesian framework allows us to specify a prior belief (via a Dirichlet distribution) about these probabilities and update this belief as data accumulates.
- **Propagation of Uncertainty:** Rather than providing just point estimates, the fully Bayesian approach produces a posterior distribution over the probabilities. This distribution captures the uncertainty inherent in the estimation process and informs decisions under uncertainty.
- **Hierarchical Modeling Capabilities:** In complex scenarios, probabilities might themselves be generated from higher-level processes. A Bayesian framework permits the specification of hyperpriors on the parameters (e.g., through a hierarchical Dirichlet model), enabling a richer, multi-level understanding of uncertainty.
- **Robust Predictive Inference:** By integrating over the uncertainties in \mathbf{p} , the marginal likelihood (or predictive distribution) derived in the Dirichlet-Multinomial model is more robust. This integration accounts for all plausible values of \mathbf{p} , leading to more reliable probabilistic predictions about future observations.

9 Application to Blockchain Consensus

By modeling the committee selection process using Dirichlet priors, one can:

- Reflect the stake-based influence of participants on their likelihood of selection.
- Incorporate observed data to refine the understanding of the selection probabilities.
- Use the posterior estimates of α to predict future committee compositions and ensure fairness, or adjust selection mechanisms.

This approach combines both flexibility and rigor, providing a robust framework for modeling the inherent randomness in blockchain consensus protocols.

Customizations and Extensions

- Adjust the prior distributions for α_i if we have specific beliefs (e.g., using narrower Gamma priors or a uniform distribution).
- Modify the number of trials n in the multinomial likelihood according to how the normalized data were scaled.
- Consider adding prior predictive checks or posterior predictive sampling to validate the model's performance against observed data.

10 Byzantine Fault Tolerance and Dirichlet Priors

10.1 Byzantine Fault Tolerance (BFT) in Standard Systems

In traditional (uniform-weight) BFT, a system can tolerate up to f faulty nodes out of a total of n nodes as long as

$$f < \frac{n}{3}, \quad (11)$$

which ensures that the honest nodes control more than two-thirds of the votes. When decisions require a $2/3$ supermajority, this also ensures liveness.

10.2 Translating to the Weighted Case

In a weighted setting, the analogous condition is that the *total voting power* of the faulty participants (in the worst-case scenario) must be less than $1/3$ of the total committee weight, or the number of seats out of the total of k seats in the committee. If we assume that an adversary can choose the most influential participants, then we sort participants in descending order by their $E[X_i]$.

Weighted Committee Model

Each participant i in a committee of size k has an expected number of seats (or voting power)

$$E[X_i] = k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}, \quad (12)$$

where the α_i come from our Dirichlet model (and higher α_i indicate higher stake).

Define the Cumulative Measure

$$C(f) = \sum_{i=1}^f E[X_{(i)}], \quad (13)$$

where f is the number of faulty participants and $E[X_{(i)}]$ is the expected number of seats for the i -th participant in the sorted order. The notation $X_{(i)}$ indicates the i -th order statistic, which is the i -th largest value in a sorted list. $E[X_{(1)}] \geq E[X_{(2)}] \geq \dots \geq E[X_{(n)}]$.

10.3 The Critical Thresholds

Safety (Agreement)

To avoid conflicting decisions (i.e. to maintain safety), the total voting power controlled by the faulty participants should be less than one-third of the committee weight. In other words, if f is the maximum number of faulty participants that can be tolerated, then the worst-case cumulative voting power must satisfy:

$$C(f) < \frac{k}{3}. \quad (14)$$

If the coalition of the top f participants exceeds $k/3$ in weight, then they could, in principle, prevent the honest nodes (which would then have less than $2k/3$) from reaching consensus on a unique decision.

Liveness (Progress)

Liveness requires that the honest participants be able to make progress, which in many protocols is ensured by requiring at least a $2/3$ majority for decisions. If the faulty nodes control less than $k/3$ of the weight, then the honest nodes naturally control more than $2k/3$, which supports liveness.

Thus, instead of associating one threshold with safety and the other separately with liveness, the fundamental requirement is that the adversary's total weight remains under one-third of the total voting power. In a uniform system, this is equivalent to $f < n/3$, but with heterogeneous weights it becomes:

$$C(f) = \sum_{i=1}^f E[X_{(i)}] < \frac{k}{3}. \quad (15)$$

Then, automatically, the remaining (honest) weight is more than $\frac{2k}{3}$.

10.4 Maximum Faulty Participants

If we define f to be the **maximum number of faulty participants** that our system can tolerate, then a correct interpretation is that the system will remain *both safe and live* if the worst-case (i.e. the highest weighted) f participants collectively hold *less than* $k/3$ of the total vote weight. Conversely, if their cumulative influence reaches or exceeds $k/3$, then the system is **at risk**—safety (agreement) may be compromised and liveness (ability to progress) may stall.

In practice, we can determine the maximum tolerable f by computing the cumulative sums $C(1), C(2), \dots$ until we find the largest f such that:

$$C(f) < \frac{k}{3}. \quad (16)$$

Applying This to Our Bayesian Model with Dirichlet Priors

Recall that in our model, the expected number of seats for participant i is given by

$$E[X_i] = k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}, \quad (17)$$

which serves as a proxy for that participant's influence (or voting power).

Sorting by Influence

Since the α_i may vary and so do $E[X_i]$, we want to rank the participants in descending order by $E[X_i]$. This order tells us which participants have the most influence.

Cumulative Voting Power

Define the cumulative voting power of the top m participants as

$$C(m) = \sum_{i=1}^m E[X]_{(i)}, \quad (18)$$

where $E[X]_{(1)} \geq E[X]_{(2)} \geq \dots \geq E[X]_{(n)}$ are the ordered expected seat counts.

Fault Tolerance Threshold

The system will be secure (safe and live) as long as no colluding adversarial coalition can muster at least $1/3$ of the total voting power. In other words, if we can identify the smallest integer m for which

$$C(m) \geq \frac{k}{3}, \quad (19)$$

then if those top m participants were malicious, the Byzantine voting weight would meet or exceed the critical threshold for breaking the consensus guarantees.

Thus, while the decision rule in many protocols is to require $2/3$ of votes to commit (ensuring that honest voting power is above that level), the ultimate fault-tolerance limit is based on keeping the Byzantine fraction below $1/3$ of the total weight. our intuition is exactly right: by applying order statistics (ranking by stake/ α), we can quantify precisely how many—or rather, which—participants would need to be malicious before compromising the system.

10.5 Algorithm for BFT Analysis

The following algorithm outlines the steps to compute the maximum tolerable Byzantine participants for safety and liveness in a weighted committee selection process. The algorithm takes as input the Dirichlet parameters (stake distribution), committee size, and number of participants, and outputs the maximum tolerable Byzantine participants for both safety and liveness.

10.6 Implications

Byzantine Resilience

If m_s is small, even a small number of adversaries with high stake can compromise safety. Conversely, a large m_s indicates that the system is resilient to faults spread across many participants.

Stake Distribution Sensitivity

Systems with concentrated stake (few participants dominating $E[X_i]$) are more vulnerable compared to systems with evenly distributed influence.

Final Thoughts

Incorporating order statistics is crucial to determine the resilience of the system because the impact of faults depends on both the relative and cumulative influence of participants. This approach gives a precise framework for analyzing fault tolerance thresholds in heterogeneous systems like blockchain consensus protocols.

11 Optimal Committee Size Approximation

It is possible to optimize k given α , n , and f . By considering order statistics and iteratively evaluating cumulative voting power thresholds, we can derive the smallest k that ensures desired BFT guarantees. While exact optimization depends on factors like computational constraints and nuances of the blockchain protocol, we can derive an approximate method to balance system resilience and efficiency.

Algorithm 1 Byzantine Fault Tolerance Analysis

```
1: Input:
2:    $\alpha$ : Dirichlet parameters (stake distribution)
3:    $k$ : Committee size
4:    $n$ : Number of participants
5: Output:  $m_s, m_l$ : Maximum tolerable Byzantine participants for safety and liveness
6: Step 1: Calculate expected seat counts
7: for each participant  $i \in \{1, 2, \dots, n\}$  do
8:   Compute  $E[X_i] \leftarrow k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}$ 
9: end for
10: Step 2: Sort participants by expected seat counts
11: Sort  $\{E[X_i]\}_{i=1}^n$  in descending order to get  $E[X]_{(1)} \geq E[X]_{(2)} \geq \dots \geq E[X]_{(n)}$ 
12: Step 3: Compute cumulative voting power
13: for  $m = 1$  to  $n$  do
14:   Compute  $C(m) \leftarrow \sum_{i=1}^m E[X]_{(i)}$ 
15: end for
16: Step 4: Determine fault thresholds
17:  $m_s \leftarrow 0$ 
18:  $m_l \leftarrow 0$ 
19: for  $m = 1$  to  $n$  do
20:   if  $C(m) \geq \frac{k}{3}$  and  $m_s = 0$  then
21:      $m_s \leftarrow m$  ▷ Safety threshold reached
22:   end if
23:   if  $C(m) \geq \frac{2k}{3}$  and  $m_l = 0$  then
24:      $m_l \leftarrow m$  ▷ Liveness threshold reached
25:   end if
26: end for
27: return  $m_s - 1$  as maximum Byzantine participants for safety
28: return  $m_l - 1$  as maximum Byzantine participants for liveness
```

11.1 Key Quantities

Inputs:

1. **BFT Threshold** (f): Desired fraction of tolerated Byzantine faults (e.g., $f = 1/3$ for traditional safety and liveness guarantees).
2. **Group Size** (n): Total number of participants (SPOs) eligible for committee selection.
3. **Dirichlet α** : A vector representing stake distribution among n participants. Participants with larger α_i have higher expected influence.

Outputs:

- **Optimal Committee Size** (k): The number of seats in the committee ensuring desired BFT guarantees while minimizing computational and communication overhead.

11.2 Decision Criteria

The optimal k must satisfy:

1. **Safety Constraint**: The system should tolerate Byzantine participants controlling up to f of total committee influence. This requires:

$$C(m_s) < f \cdot k, \quad (20)$$

where $C(m_s)$ is the cumulative voting power of the top m_s participants.

2. **Liveness Constraint**: Honest participants must hold at least $(1 - f)$ of the voting power:

$$C(m_h) \geq (1 - f) \cdot k, \quad (21)$$

where $C(m_h)$ is the cumulative influence of the top m_h honest participants.

3. **Stake Distribution Sensitivity**: A larger k dilutes the relative weight of high-stake participants (reducing the influence of top-ranked α_i), improving fairness and resilience.
4. **Efficiency**: Smaller k minimizes computational and communication overhead. k should balance security and efficiency.

11.3 Optimum k Approximation Algorithm

We aim to determine the smallest k meeting the BFT thresholds. The algorithm iteratively computes expected seat counts, sorts participants by influence, and checks cumulative voting power against the safety and liveness constraints. The process continues until both constraints are satisfied. The algorithm is as follows:

11.4 Factors Affecting Optimization

Impact of α

- **Highly unequal α** : (few dominant participants): Requires larger k to reduce adversarial influence and achieve fairness.
- **Uniform α** : Smaller k suffices, as influence is evenly distributed.

Algorithm 2 Optimal Committee Size Determination

```
1: Input:
2:    $\alpha$ : Dirichlet parameters (stake distribution)
3:    $n$ : Number of participants
4:    $f$ : Byzantine fraction threshold
5:    $k_{\min}$ : Minimum committee size
6: Output:  $k_{\text{optimal}}$ : Optimal committee size
7:  $k \leftarrow k_{\min}$ 
8: constraints_satisfied  $\leftarrow$  false
9: while not constraints_satisfied do
10:   for  $i = 1$  to  $n$  do
11:      $E[X_i] \leftarrow k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}$ 
12:   end for
13:   Sort participants in descending order of  $E[X_i]$  to get  $E[X]_{(1)} \geq E[X]_{(2)} \geq \dots \geq E[X]_{(n)}$ 
14:    $C(m_s) \leftarrow \sum_{i=1}^{m_s} E[X]_{(i)}$   $\triangleright$  Cumulative voting power for safety
15:    $C(m_h) \leftarrow \sum_{i=1}^{m_h} E[X]_{(i)}$   $\triangleright$  Cumulative voting power for liveness
16:   if  $C(m_s) < f \cdot k$  and  $C(m_h) \geq (1 - f) \cdot k$  then
17:     constraints_satisfied  $\leftarrow$  true
18:   else
19:      $k \leftarrow k + 1$ 
20:   end if
21: end while
22: return  $k$  as  $k_{\text{optimal}}$ 
```

Group Size n

- Larger n : Requires larger k to accommodate diverse participants while maintaining fault tolerance.
- Smaller n : Can tolerate smaller k , as fewer participants contribute to total influence.

Desired Byzantine Fraction f

- Lower f : Requires larger k to dilute adversarial influence.
- Higher f : Smaller k might suffice, but with reduced fault tolerance.

11.5 Approximation Formula

While iterative computation is more precise, a heuristic formula for k can be derived as:

$$k_{\text{optimal}} \approx \frac{\max(C_{\text{top}}, C_{\text{honest}})}{f}, \quad (22)$$

where:

- C_{top} : Influence of top-ranked participants representing $f \cdot k$.
- C_{honest} : Influence of honest participants needed for $(1 - f)$ resilience.

Below is an algorithm outline of how we might approximately “optimize” the committee size given a target number of Byzantine faults f (that is, the committee should tolerate up to f adversaries), a total number n of participants (with associated Dirichlet parameters α) and the ensuing expected vote shares.

Algorithm 3 OptimalCommitteeSize

```

1: Input:
2:    $\alpha$ : Dirichlet parameters (stake distribution)
3:    $n$ : Group size
4:    $f$ : Byzantine Fault Tolerance fraction
5:    $k_0$ : Initial guess for committee size
6: Output:  $k_{\text{optimal}}$ : Optimal committee size
7: Initialize:
8:   Sort  $\alpha\_vector$  in descending order
9:    $k \leftarrow k_0$ 
10: while True do
11:   for each participant  $i$  do
12:      $E[X_i] \leftarrow k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha\_vector_j}$ 
13:   end for
14:   Compute  $C_{\text{top}}$ : sum of  $E[X_i]$  for top  $m_s$  participants ▷ Safety constraint
15:   Compute  $C_{\text{honest}}$ : sum of  $E[X_i]$  for top  $m_h$  honest participants ▷ Liveness constraint
16:   if  $C_{\text{top}} < f \cdot k$  and  $C_{\text{honest}} \geq (1 - f) \cdot k$  then
17:     break
18:   end if
19:    $k \leftarrow k + 1$ 
20: end while
21: return  $k$  as  $k_{\text{optimal}}$ 

```

Explanation of the Algorithm

The algorithm iteratively computes the expected seat counts for each participant based on the current committee size k . It then checks if the cumulative voting power of the top m_s participants (representing the worst-case Byzantine coalition) is less than $f \cdot k$, and if the cumulative voting power of the top m_h honest participants is greater than or equal to $(1 - f) \cdot k$. If both conditions are satisfied, it breaks out of the loop and returns the optimal committee size.

In many traditional BFT systems, when every participant is equally weighted, the well-known bound is that a committee (or network) of size k can tolerate up to f Byzantine faults provided that

$$k \geq 3f + 1. \tag{23}$$

When weights are unequal—say, derived from stake or (in our Bayesian model) proportional to α_i —the situation is more subtle. Now each participant’s influence is given by its expected seat count

$$E[X_i] = k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}. \tag{24}$$

A coalition that comprises the f most influential (highest α_i or equivalently highest $E[X_i]$) participants is our worst-case Byzantine adversary. In that case, we want their cumulative weight not to

exceed $\frac{k}{3}$ (so that honest participants hold more than two-thirds of the voting power).

Because the Dirichlet model yields expected seat counts that scale linearly with k , the relative weights are fixed. If we took only expectations, the condition for safety (in expectation) would be

$$\sum_{i=1}^f \frac{\alpha_{(i)}}{\sum_{j=1}^n \alpha_j} < \frac{1}{3}, \quad (25)$$

where $\alpha_{(1)} \geq \alpha_{(2)} \geq \dots \geq \alpha_{(n)}$ is the sorted order. Notice that if this inequality is satisfied, then the weighted adversary's share is below the threshold for compromise on average. In an “idealized” setup, then, any k would do.

However, real committee selection via multinomial sampling introduces random variation. Even if the expected fraction of voting power held by the worst-case f nodes is below $\frac{1}{3}$, fluctuations (variance) may produce an outcome in which that fraction exceeds the critical threshold. In other words, a larger committee size k means that the relative fluctuations (by the law of large numbers) shrink, and the actual vote shares concentrate more tightly around their expectations.

Optimization Strategy

Our goal is to choose k as the (approximately) minimal committee size so that with high probability the Byzantine (worst-case) faction does not exceed $\frac{k}{3}$ of the total committee votes.

1. Define the “Adversary Weight”

Let

$$q = \sum_{i=1}^f \frac{\alpha_{(i)}}{\sum_{j=1}^n \alpha_j}. \quad (26)$$

Then the expected cumulative seats for the worst-case f is:

$$\mu = k \cdot q. \quad (27)$$

2. Safety Condition (in the Presence of Sampling Variance)

We require that the probability that the actual number of seats X_b held by these f participants exceeds $\frac{k}{3}$ is below a small error threshold ϵ . That is,

$$P\left(X_b \geq \frac{k}{3}\right) \leq \epsilon. \quad (28)$$

(Here X_b is the random sum of the seats for the adversarial set, and its expectation is μ .)

3. Using Concentration Inequalities

Because X_b is a sum of multinomial counts (which, under mild conditions, can be approximated by a sum of independent (or weakly correlated) bounded variables), we can use Chernoff-type bounds to say:

$$P\left(X_b \geq (1 + \delta)\mu\right) \leq \exp\left(-\frac{\delta^2 \mu}{2 + \delta}\right). \quad (29)$$

We wish to set

$$(1 + \delta)\mu = \frac{k}{3}. \quad (30)$$

Since $\mu = kq$, this equation becomes:

$$1 + \delta = \frac{k}{3kq} = \frac{1}{3q}, \quad (31)$$

or

$$\delta = \frac{1}{3q} - 1. \quad (32)$$

Then, requiring

$$\exp\left(-\frac{\delta^2 kq}{2 + \delta}\right) \leq \epsilon, \quad (33)$$

we solve for k :

$$k \geq \frac{(2 + \delta)(-\ln \epsilon)}{\delta^2 q}. \quad (34)$$

In words, for a given tolerance level ϵ (say, $\epsilon = 0.01$ for 99% confidence) and the “adversarial expected fraction” q (which is determined from our sorted α vector and our target f), we obtain a lower bound on k .

4. Trade-Off

- **Smaller k :** Lower overhead, but higher relative variance in seat allocation, which might allow unlikely—but possible—fluctuations that violate the $\frac{1}{3}$ safety rule.
- **Larger k :** Reduces variance so that the actual fraction of Byzantine control will concentrate near q , thus enforcing security with high probability. However, k cannot be made arbitrarily large due to practical constraints (communication overhead, latency, etc.).

Putting It Together

The following algorithm summarizes the steps to compute the minimum committee size k_{\min} that satisfies the safety condition with high probability, given the Dirichlet parameters α , the number of participants n , the number of Byzantine participants f , and the error tolerance ϵ .

Algorithm 4 Committee Size Optimization using Chernoff Bounds

- 1: **Input:**
 - 2: α : Dirichlet parameters (stake distribution)
 - 3: n : Number of participants
 - 4: f : Number of Byzantine participants to tolerate
 - 5: ϵ : Error tolerance (e.g., 0.01)
 - 6: **Output:** k_{\min} : Minimal committee size
 - 7: Sort α in descending order to get $\alpha_{(1)} \geq \alpha_{(2)} \geq \dots \geq \alpha_{(n)}$
 - 8: Compute $q \leftarrow \sum_{i=1}^f \frac{\alpha_{(i)}}{\sum_{j=1}^n \alpha_j}$ ▷ Adversary weight fraction
 - 9: Compute $\delta \leftarrow \frac{1}{3q} - 1$ ▷ Deviation parameter
 - 10: Compute $k_{\min} \leftarrow \lceil \frac{(2+\delta)(-\ln \epsilon)}{\delta^2 q} \rceil$ ▷ Ceiling function ensures integer result
 - 11: **return** k_{\min}
-

11.6 Discussion

Model Assumptions

(a) We assume that the expected values are a good proxy for actual vote shares when k is large, and (b) that the Chernoff bound gives a reasonable approximation to the tail probability of the multinomial sum.

Optimization Complexity

In a real implementation, we might want to simulate the committee selection process (using the full hierarchical Bayesian model) to validate these approximations. We can also adjust k iteratively until simulation results show that the probability of crossing the $1/3$ threshold falls below ϵ .

Balancing Other Factors

This procedure optimizes k with respect to fault tolerance. In a complete system design, we may also need to consider liveness conditions and trade-offs concerning communication complexity and latency in addition to safety.

12 Conclusion

This paper presented a comprehensive Bayesian framework for modeling committee selection in blockchain consensus protocols with heterogeneous participant influence. Through a Dirichlet-Multinomial approach, we derived closed-form expressions for the expected seat counts and their variances, enabling precise analysis of committee composition dynamics.

We established a rigorous methodology for Byzantine fault tolerance analysis in weighted committee settings, demonstrating how order statistics can identify worst-case adversarial coalitions. By examining cumulative voting power thresholds, we determined the maximum number of Byzantine participants a system can tolerate while maintaining both safety and liveness guarantees.

A key contribution is our algorithm for optimizing committee size based on stake distribution. Using Chernoff bounds, we derived a principled approach to calculate the minimum committee size needed to ensure that with high probability $(1 - \epsilon)$, the cumulative weight of Byzantine participants remains below critical thresholds. This approach balances security requirements with operational efficiency, as larger committees provide stronger security guarantees but introduce additional communication overhead.

The framework developed here is directly applicable to proof-of-stake blockchain systems where committee selection is weighted by stake. By applying our optimization algorithm, designers can determine the optimal committee size that provides robust Byzantine fault tolerance based on specific stake distributions, desired security levels, and performance constraints.

This approach represents a significant advancement over traditional uniform-weight BFT analysis by accounting for the heterogeneous influence of participants in realistic blockchain deployments, offering a mathematical foundation for secure and efficient committee-based consensus mechanisms.

A Bayesian Statistics Calculations

Let's recap this step by step to compute the expected value $E[X_i]$ for a fixed i in the context of the hierarchical model. In the context of the hierarchical model, we want to compute the expected value of X_i , which represents the number of times participant i is selected for the committee. The expected value of X_i is a crucial quantity in the context of the Dirichlet-Multinomial model. It provides insight into how many times we expect participant i to be selected based on their latent probability p_i and the total number of draws k . This expected value is particularly useful for understanding the distribution of committee members and can inform decision-making processes regarding participant selection. The expected value of X_i can be derived from the properties of the Dirichlet-Multinomial distribution. The expected value of a random variable X_i that follows a multinomial distribution with parameters k and \mathbf{p} is given by:

- X_i : The number of times participant i is selected for the committee.
- $\mathbf{p} = (p_1, p_2, \dots, p_n)$: The latent probabilities of selecting each participant, drawn from a Dirichlet distribution:

$$\mathbf{p} \sim \text{Dirichlet}(\boldsymbol{\alpha}). \quad (35)$$

- $\mathbf{X} = (X_1, X_2, \dots, X_n)$: The committee composition follows a multinomial distribution:

$$\mathbf{X} \sim \text{Multinomial}(k, \mathbf{p}), \quad (36)$$

where

$$k = \sum_{i=1}^n X_i \quad (37)$$

is the total number of committee seats.

Expected Value of X_i

The expected value $E[X_i]$ is given by

$$E[X_i] = E[E[X_i \mid \mathbf{p}]], \quad (38)$$

where the outer expectation integrates over the distribution of \mathbf{p} , and the inner expectation is conditional on \mathbf{p} .

Step 1: Conditional Expectation

Given the multinomial distribution, the expected value of X_i conditioned on \mathbf{p} is

$$E[X_i \mid \mathbf{p}] = k \cdot p_i, \quad (39)$$

where p_i is the probability of selecting participant i .

Step 2: Marginalize Over \mathbf{p}

Since \mathbf{p} follows a Dirichlet distribution

$$\mathbf{p} \sim \text{Dirichlet}(\boldsymbol{\alpha}), \quad (40)$$

the marginal expectation of p_i is

$$E[p_i] = \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}. \quad (41)$$

Now, combining these,

$$E[X_i] = E[E[X_i | \mathbf{p}]] = k \cdot E[p_i] = k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}. \quad (42)$$

Final Result

The expected value of X_i is

$$E[X_i] = k \cdot \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}. \quad (43)$$

Intuition

This result makes sense intuitively:

- k is the total number of draws (committee seats).
- $\frac{\alpha_i}{\sum_{j=1}^n \alpha_j}$ represents the proportion of "weight" assigned to participant i in the Dirichlet distribution, which acts as a prior belief about their likelihood of being selected.

Let's calculate the variance $\text{Var}(X_i)$ for a fixed i in our model, building on the expected value we derived earlier.

B Variance Calculations

The variance of X_i in the context of the Dirichlet-Multinomial model can be derived using the law of total variance. The variance captures the uncertainty in the number of times participant i is selected for the committee, accounting for both the multinomial distribution and the uncertainty in the selection probabilities p_i . The variance of X_i is given by:

$$\text{Var}(X_i) = E[\text{Var}(X_i | \mathbf{p})] + \text{Var}(E[X_i | \mathbf{p}]). \quad (44)$$

This equation states that the total variance of X_i can be decomposed into two components:

- The expected value of the conditional variance of X_i given \mathbf{p} .
- The variance of the conditional expectation of X_i given \mathbf{p} .

This decomposition is useful because it allows us to separately analyze the variability due to the multinomial distribution and the variability due to the uncertainty in the selection probabilities p_i . The variance of X_i can be derived from the properties of the Dirichlet-Multinomial distribution.

The variance of a random variable X_i that follows a multinomial distribution with parameters k and \mathbf{p} is given by:

$$\text{Var}(X_i | \mathbf{p}) = k \cdot p_i \cdot (1 - p_i). \quad (45)$$

This equation captures the variability of X_i given the selection probabilities p_i . The term $k \cdot p_i \cdot (1 - p_i)$ reflects the variance of a binomial distribution, scaled by the total number of draws k .

We now integrate over the Dirichlet distribution to calculate the expected conditional variance and the variance of the conditional mean.

First Term: $E[\text{Var}(X_i | \mathbf{p})]$: Substitute p_i into the Dirichlet expectation $E[p_i]$:

$$E[\text{Var}(X_i | \mathbf{p})] = k \cdot (E[p_i] - E[p_i^2]).$$

Using properties of the Dirichlet distribution: - $E[p_i] = \frac{\alpha_i}{\sum_{j=1}^n \alpha_j}$, - $E[p_i^2] = \frac{\alpha_i(\alpha_i+1)}{(\sum_{j=1}^n \alpha_j)(\sum_{j=1}^n \alpha_j + 1)}$.

Thus:

$$E[\text{Var}(X_i | \mathbf{p})] = k \cdot \left(\frac{\alpha_i}{\sum_{j=1}^n \alpha_j} - \frac{\alpha_i(\alpha_i + 1)}{\left(\sum_{j=1}^n \alpha_j\right) \left(\sum_{j=1}^n \alpha_j + 1\right)} \right).$$

Second Term: $\text{Var}(E[X_i | \mathbf{p}])$: The conditional mean $E[X_i | \mathbf{p}] = k \cdot p_i$. The variance of p_i under the Dirichlet distribution is:

$$\text{Var}(p_i) = \frac{\alpha_i \left(\sum_{j=1}^n \alpha_j - \alpha_i \right)}{\left(\sum_{j=1}^n \alpha_j \right)^2 \cdot \left(\sum_{j=1}^n \alpha_j + 1 \right)}.$$

Multiply by k^2 to get:

$$\text{Var}(E[X_i | \mathbf{p}]) = k^2 \cdot \text{Var}(p_i).$$

Substitute $\text{Var}(p_i)$:

$$\text{Var}(E[X_i | \mathbf{p}]) = k^2 \cdot \frac{\alpha_i \left(\sum_{j=1}^n \alpha_j - \alpha_i \right)}{\left(\sum_{j=1}^n \alpha_j \right)^2 \cdot \left(\sum_{j=1}^n \alpha_j + 1 \right)}.$$

Final Formula: Combining both terms, the total variance is:

$$\text{Var}(X_i) = k \cdot \left(\frac{\alpha_i}{\sum_{j=1}^n \alpha_j} - \frac{\alpha_i(\alpha_i + 1)}{\left(\sum_{j=1}^n \alpha_j\right) \left(\sum_{j=1}^n \alpha_j + 1\right)} \right) + k^2 \cdot \frac{\alpha_i \left(\sum_{j=1}^n \alpha_j - \alpha_i \right)}{\left(\sum_{j=1}^n \alpha_j \right)^2 \cdot \left(\sum_{j=1}^n \alpha_j + 1 \right)}.$$

Intuition Behind the Variance

- The first term reflects the variability within the multinomial distribution when p_i is fixed.
- The second term accounts for the uncertainty in p_i due to its Dirichlet prior, which adds additional variance.

C Chernoff Bounds in the Context of BFT

Chernoff bounds are a powerful tool in probability theory used to bound the tail probabilities of random variables. They provide a way to quantify how far a sum of random variables is likely to deviate from its expected value. Below is a detailed explanation in the context of our Bayesian model and BFT optimization.

C.1 Context of Chernoff Bounds

In our model:

- The total number of seats k is fixed, and these seats are assigned to participants according to a Multinomial distribution.
- Let X_b represent the total number of seats held by the worst-case f participants (the colluding Byzantine adversaries).
- The expected value of X_b is given by

$$\mu = E[X_b] = k \cdot q, \quad (46)$$

where

$$q = \sum_{i=1}^f \frac{\alpha_{(i)}}{\sum_{j=1}^n \alpha_j} \quad (47)$$

is the expected cumulative fraction of influence of the adversarial participants.

While X_b is expected to stay close to μ , there is some probability that it deviates significantly due to the randomness of the sampling process. Chernoff bounds help us bound the probability of X_b exceeding a critical threshold, such as $\frac{k}{3}$, which would compromise safety.

Chernoff bounds apply to the sum of independent (or weakly dependent) random variables. For our case, the Multinomial distribution can be viewed as assigning k independent, weighted “votes” to the participants, distributed according to the probabilities

$$\mathbf{p} \sim \text{Dirichlet}(\boldsymbol{\alpha}). \quad (48)$$

Let

$$X_b = \sum_{i=1}^f X_i, \quad (49)$$

where X_i is the number of seats held by the i -th participant. For $\mu = E[X_b]$, Chernoff bounds provide a way to calculate

$$P(X_b \geq (1 + \delta)\mu), \quad (50)$$

which is the probability of X_b exceeding $(1 + \delta)\mu$ by a multiplicative factor $1 + \delta$. This represents the *upper tail probability*.

C.2 The Chernoff Bound Formula

The bound can be written as

$$P(X_b \geq (1 + \delta)\mu) \leq \exp\left(-\frac{\delta^2\mu}{2 + \delta}\right). \quad (51)$$

Here,

- $\delta > 0$ quantifies the degree of deviation above μ ,
- $\mu = kq$, the expected weight of the adversaries.

This inequality tells us that the probability of a large deviation decreases exponentially with the size of μ (and hence k).

C.3 Safety Requirement

To ensure safety, we require that

$$P(X_b \geq \frac{k}{3}) \leq \epsilon, \quad (52)$$

where ϵ is a small failure tolerance (for example, $\epsilon = 0.01$, corresponding to 99% confidence).

Let

$$\frac{k}{3} = (1 + \delta)\mu. \quad (53)$$

Then,

$$1 + \delta = \frac{\frac{k}{3}}{\mu} = \frac{\frac{k}{3}}{kq} = \frac{1}{3q}. \quad (54)$$

This yields

$$\delta = \frac{1}{3q} - 1. \quad (55)$$

Substituting δ into the Chernoff bound gives:

$$P(X_b \geq \frac{k}{3}) \leq \exp\left(-\frac{\delta^2\mu}{2 + \delta}\right). \quad (56)$$

Rewriting the inequality in terms of k , we obtain:

$$k \geq \frac{(2 + \delta)(-\ln \epsilon)}{\delta^2 q}. \quad (57)$$

C.4 Interpretation

This result tells us how large the committee size k must be to ensure that the probability of a Byzantine coalition controlling more than $\frac{k}{3}$ seats is below ϵ . Larger k reduces the variability in seat allocations, making the likelihood of exceeding the threshold exponentially small.

C.5 Key Properties of Chernoff Bounds

1. Exponentially Decaying Tail Probabilities:

The probability of large deviations decreases exponentially with μ , meaning that as k increases the variance in the random sampling diminishes and the actual seat distribution concentrates around its expected value.

2. Applicability to Independent or Weakly Dependent Variables:

While Chernoff bounds strictly apply to sums of independent random variables, they are often good approximations for weakly dependent variables, such as those from a Multinomial distribution.

3. Confidence Tuning (ϵ):

We can adjust ϵ to control the likelihood of an adversarial coalition exceeding the threshold. A smaller ϵ requires a larger k , improving fault tolerance at the cost of larger committees.

In optimizing k for our BFT analysis, Chernoff bounds allow us to:

- Quantify the probability of unsafe configurations.
- Translate a safety margin (represented by the failure tolerance ϵ) into a concrete requirement for the committee size k .
- Ensure that the actual voting power distribution aligns closely with the expectations derived from the α vector and the desired BFT properties.

D Computer Code

The following code implements the Bayesian model for committee selection and the optimization of committee size based on the Dirichlet distribution. The code is structured to load data, sample from the Dirichlet distribution, and visualize the results. It also includes functions for saving and loading models and traces. The code is designed to be modular and reusable, allowing for easy integration into larger projects or systems. It includes functions for loading data, sampling from the Dirichlet distribution, and visualizing the results. The code is written in Python and uses the PyMC library for probabilistic modeling and ArviZ for visualization. It includes functions for loading data, sampling from the Dirichlet distribution, and visualizing the results. The code is designed to be modular and reusable, allowing for easy integration into larger projects or systems.

```
1
2 #!/usr/bin/env python
3 # -*- coding: utf-8 -*-
4 """
5 Module: bayes-stake-dist.py
6
7 Created on 2025-03-23 by Rob Jones <robert.jones@shielded.io>
8
9 This script calculates the group stake distribution using a Bayesian approach.
10 It uses a Dirichlet distribution as a generative model of the probabilities of
11 different participants being selected for a committee, based on their stake
12 sizes. The model is implemented using PyMC and ArviZ for sampling and
13 visualization.
14
15 The script includes:
```

```

16 - Bayesian model definition using PyMC
17 - Sampling from the posterior distribution
18 - Posterior predictive checks
19 - Visualization of the results
20 - Saving and loading the model and trace
21 - Saving and loading posterior predictive samples
22
23 """
24 import random
25 import matplotlib.pyplot as plt
26 from scipy.stats import gamma
27 import pymc as pm
28 import arviz as az
29 from data import load_data
30 from participation_lib import get_stake_distribution, assign_committee
31 from typing import List, Dict, Any, Union
32 import numpy as np
33 import pandas as pd
34 import argparse
35 from pathlib import Path
36 from joblib import Parallel, delayed
37
38
39 def load_population(input_data_file):
40     """
41     Load and clean the data from the specified CSV file.
42
43     Args:
44         input_data_file (Path or str): The path to the CSV data file.
45
46     Returns:
47         pd.DataFrame: The cleaned DataFrame containing SPO information.
48     """
49     print("Loading data...")
50     population = load_data(input_data_file)
51     print(population.info())
52     print(population.describe())
53     return population
54
55 # The get_stake_distribution function is used to simulate the stake
56 # distribution
57 # of the population. It takes the following parameters:
58 # - 'population': The DataFrame containing the population data.
59 # - 'group_size': The size of the group to be sampled.
60 # - 'num_iter': The number of iterations for the simulation.
61 # - 'plot_it': A boolean flag indicating whether to plot the distribution.
62 # The function returns a DataFrame containing the simulated stake distribution
63 # .
64 # The function simulates the stake distribution by randomly sampling from the
65 # population and calculating the stake weights for each participant.
66 # The stake weights are normalized to sum to 1, representing the relative
67 # stake
68 # of each participant in the group.

```

```

67
68
69 def sample_group(
70     population: pd.DataFrame,
71     stake_size: List[int],
72     group_size: int,
73     plot_it: bool = False,
74     num_ticks: int = 10,
75 ) -> pd.DataFrame:
76     """Sample a group of participants ensuring
77     one has the given stake size(s).
78
79     Args:
80         population (pd.DataFrame): The population of registered SPOs.
81         stake_size (List[int]): The stake size(s) to include in the group.
82         group_size (int): The number of participants in the group.
83         plot_it (bool): Whether to plot the distribution of stake weights.
84         num_ticks (int): The number of ticks to display on the x-axis.
85
86     Returns:
87         pd.DataFrame: A DataFrame containing the group of participants.
88     """
89     # Sample a group of participants with the chosen stake size
90     # at the beginning of the group (starting from index 1)
91     group = pd.DataFrame()
92     for stake in stake_size:
93         # Find rows with the given stake value
94         subset = population[population["stake"] == stake]
95         if not subset.empty:
96             # If present, sample one matching record
97             selected = subset.sample(1, replace=False)
98         else:
99             # If not present, sample one record and set
100             # its stake values to the given stake size
101             selected = population.sample(1, replace=False).copy()
102             selected["stake"] = stake
103             selected["stake_weight"] = stake
104             # Add the selected record to the group
105             group = pd.concat([group, selected], ignore_index=True)
106     if len(group) == 0:
107         # No participants found with stake value, so add it artificially
108         group = population.sample(1, replace=False)
109         group["stake"] = stake
110         group["stake_weight"] = stake
111         group.index = [0]
112     # Add to that group the remaining participants
113     # sampled from the population, excluding the selected participant
114     # and sort the group by stake value from highest to lowest
115     other_participants = get_stake_distribution(
116         population,
117         group_size=group_size - 1,
118         num_iter=1000, # 1 for exact instance, >> 1 for smoothed
119         plot_it=False,
120     ).sort_values(by="stake", ascending=False)

```

```

121
122     # Concatenate the selected participant with the other participants
123     group = pd.concat([group, other_participants], ignore_index=True)
124     # Reindex the group to start from 1
125     group.index = group.index + 1
126     # and sort the group by stake value from highest to lowest,
127     # save for the selected stake in the first position and finally
128     # normalize the stake values as "stake_weight"
129     group["stake_weight"] = group.stake / group.stake.sum()
130
131     # Plot the distribution of the stake weights
132     if plot_it:
133         plt.figure("Distribution of the Stake Weights", figsize=(10, 6))
134         plt.bar(group.index, group.stake_weight, color="skyblue")
135         # Choose a fixed number of ticks (e.g., 10) based on domain size
136         tick_positions = np.linspace(
137             group.index.min(),
138             group.index.max(),
139             num_ticks,
140             dtype=int,
141         )
142         plt.xticks(tick_positions)
143         plt.xlabel("Participant ID")
144         plt.ylabel("Stake Weight")
145         plt.title("Stake Weights of Sampled Participants")
146         plt.show()
147
148     return group
149
150
151 def predictive_checks(
152     trace: az.InferenceData,
153     model: pm.Model,
154     plot_it: bool = False,
155 ) -> pd.DataFrame:
156     """Perform posterior predictive checks and return the results.
157
158     Args:
159         trace (az.InferenceData): The posterior samples.
160         model (pm.Model): The PyMC model.
161         plot_it (bool): Whether to plot the posterior predictive checks.
162             Default is False.
163
164     Returns:
165         pd.DataFrame: A DataFrame containing the posterior predictive samples.
166     """
167     with model:
168         ppc = pm.sample_posterior_predictive(
169             trace=trace,
170             var_names=["observed"],
171             random_seed=42,
172         )
173     # The arviz library is used to visualize the posterior trace and summarize
174     # the results (posterior means, credible intervals, etc.).

```

```

175 print(pm.summary(trace, hdi_prob=0.95))
176 if plot_it:
177     pm.plot_trace(trace)
178     pm.plot_posterior(trace)
179
180 # Posterior predictive checks
181 with model:
182     ppc = pm.sample_posterior_predictive(
183         trace=trace,
184         var_names=["observed"],
185         random_seed=42,
186     )
187 print(az.summary(ppc, hdi_prob=0.95))
188
189 if plot_it:
190     # Visualize posterior predictive checks
191     fig = plt.figure("Posterior Predictive Check", figsize=(12, 6))
192     ax = fig.add_subplot(111)
193     az.plot_ppc(ppc, kind="kde", ax=ax)
194     # az.plot_ppc(ppc, kind="scatter", ax=ax)
195     plt.title("Posterior Predictive Check (Cumulative)")
196     plt.xlabel("observed", fontsize="small")
197     plt.ylabel("Cumulative Probability", fontsize="small")
198     plt.legend(title="Posterior Predictive Checks", fontsize="small")
199     plt.show()
200
201 return ppc
202
203
204 def save_trace(trace: az.InferenceData, filename: str) -> None:
205     """
206     Save the posterior trace to a NetCDF file.
207
208     Args:
209         trace (az.InferenceData): The trace from posterior sampling.
210         filename (str): The filename for saving the trace.
211
212     Returns:
213         None
214     """
215     # Save the trace to a NetCDF file
216     trace_filename = Path(filename).with_suffix(".nc")
217     az.to_netcdf(trace, trace_filename)
218
219
220 def load_trace(filename: str) -> az.InferenceData:
221     """
222     Load the trace from a NetCDF file.
223
224     Args:
225         filename (str): Filename for the trace data.
226
227     Returns:
228         az.InferenceData: The loaded trace.
229     """

```

```

229     trace_filename = Path(filename).with_suffix(".nc")
230     # Load the trace from a NetCDF file
231     trace = az.from_netcdf(trace_filename)
232     return trace
233
234
235 def compute_and_adjust_seat_counts(
236     group: pd.DataFrame,
237     committee_size: int,
238     plot_it: bool = False,
239 ) -> np.ndarray:
240     """
241     Compute and adjust seat counts for a given group and committee size.
242
243     This function assigns committee seats using the assign_committee function,
244     rounds the seat counts using floor, allocates any remaining seats based on
245     the decimal remainders, and verifies that the total equals the committee
246     size.
247
248     Args:
249         group (pd.DataFrame): DataFrame containing the group of participants.
250         committee_size (int): Total number of committee seats.
251         plot_it (bool): Whether to plot the seat assignment (default is False)
252
253     Returns:
254         np.ndarray: Adjusted seat counts as an integer array.
255     """
256     # Observed data (counts of committee seats for each participant)
257     seats = assign_committee(
258         group,
259         committee_size=committee_size,
260         plot_it=plot_it,
261     )
262     seat_counts = seats["seat_counts"]
263
264     # Initial integer allocation using floor
265     rounded_seats = np.floor(seat_counts).astype(int)
266     remainder = committee_size - rounded_seats.sum()
267
268     if remainder > 0:
269         # Compute decimal remainders
270         decimals = seat_counts - np.floor(seat_counts)
271         # Get indices sorted by descending order of remainders
272         indices = decimals.sort_values(ascending=False).index
273         # Allocate the remaining seats to those with highest decimals
274         for idx in indices[:remainder]:
275             rounded_seats[idx] += 1
276     elif remainder < 0:
277         # If over-allocated, subtract seats from those with smallest
278         remainders
279         decimals = seat_counts - np.floor(seat_counts)
280         indices = decimals.sort_values(ascending=True).index
281         for idx in indices[:abs(remainder)]:

```

```

280         rounded_seats[idx] -= 1
281
282     seat_counts = rounded_seats
283     # Confirm the sum equals committee_size
284     assert (
285         seat_counts.sum() == committee_size
286     ), f"Sum of seats {seat_counts.sum()} does not equal committee size {
committee_size}"
287
288     print("Sum of seats:", seat_counts.sum())
289     return seat_counts
290
291
292 def plot_gamma_prior(
293     group_size: int,
294     alpha_param: float = 2.0,
295     beta_param: float = 1.0,
296     num_points: int = 1000,
297 ):
298     """
299     Visualize the Gamma distribution used as a prior for the Dirichlet
300     hyperparameter.
301
302     Args:
303         group_size (int): Maximum x-axis value (e.g., size of the group).
304         alpha_param (float): Shape parameter of the Gamma distribution (
305         default: 2.0).
306         beta_param (float): Rate parameter of the Gamma distribution (default:
307         1.0).
308         In scipy's gamma, scale = 1/beta.
309         num_points (int): Number of points to generate for the x-axis (default
310         : 1000).
311     """
312     scale_param = 1.0 / beta_param # convert beta to scale for scipy's gamma
313     x = np.linspace(0, group_size, num_points)
314     y = gamma.pdf(x, a=alpha_param, scale=scale_param)
315     plt.figure("Gamma Distribution", figsize=(10, 6))
316     plt.plot(x, y, label=f"Gamma($\alpha$={alpha_param}, $\beta$={beta_param})
317     ")
318     plt.title(f"Gamma Distribution ($\alpha$={alpha_param}, $\beta$={
319     beta_param}) ")
320     plt.xlabel("x")
321     plt.ylabel("Probability Density")
322     plt.legend()
323     plt.show()
324
325 def run_bayesian_model(
326     committee_size: int,
327     seat_counts: np.ndarray,
328     group: pd.DataFrame,
329     target_accept: float = 0.98,
330     gamma_alpha=2.0,
331     gamma_beta=1.0,

```

```

327 ) -> Dict[str, Any]:
328     """
329     Run the hierarchical Bayesian model using a Dirichlet-Multinomial
    framework.
330
331     Args:
332         committee_size (int): Number of committee seats (k).
333         seat_counts (np.ndarray): Observed counts of seats per participant.
334         group (pd.DataFrame): DataFrame containing the group of participants.
335         target_accept : float in [0, 1]. The step size is tuned such that we
336             approximate this acceptance rate. Higher values like 0.9 or
    0.98
337             often work better for problematic posteriors. This argument is
338             passed directly to sample. Default is 0.98.
339         gamma_alpha (float): Shape hyper-parameter of the Gamma distribution,
340             which is used as the hyper-prior on the Dirichlet distribution.
341             Default is 2.0.
342         gamma_beta (float): Rate hyper-parameter of the Gamma distribution.
343             Default is 1.0.
344
345     Returns:
346         Dict[str, Any]: A dictionary containing the the following key:values
347         - "concentration": The concentration of the Dirichlet distribution.
348         - "mean_prob": The mean probability of each participant.
349         - "model": The PyMC model object.
350         - "trace": The posterior samples obtained from the MCMC sampling.
351     """
352     k: int = committee_size # number of committee seats
353     n: int = len(group) # number of participants in the group
354     X: np.ndarray = (
355         seat_counts # observed data: counts of committee seats per
    participant
356     )
357     with pm.Model() as model:
358         # Priors for Dirichlet hyperparameters (Gamma prior)
359         alpha = pm.Gamma("alpha", alpha=2.0, beta=1.0, shape=n)
360
361         # Dirichlet distribution for probabilities (p)
362         p = pm.Dirichlet("p", a=alpha)
363
364         # Multinomial likelihood for observed counts
365         pm.Multinomial("observed", n=k, p=p, observed=X)
366
367         # Sampling from the posterior
368         trace = pm.sample(
369             return_inferencedata=True,
370             target_accept=target_accept,
371         )
372         # Analyze Posterior Samples of Alpha
373         posterior_mean_alpha: np.ndarray = (
374             trace.posterior["alpha"].mean(dim=("chain", "draw")).values
375         )
376         # Concentration of the Dirichlet distribution:
377         concentration = posterior_mean_alpha.sum()

```

```

378
379     # Mean probability of each participant, which normalizes
380     # the posterior mean of alpha by the concentration so it sums to 1:
381     mean_prob = posterior_mean_alpha / concentration
382
383     # Extend the mean probabilities to include the concentration
384     # at the first position (index 0) of the array
385     # and the mean probabilities at the remaining positions
386     # (index 1 to n).
387     extended_array = np.empty(mean_prob.shape[0] + 1)
388     extended_array[0] = concentration
389     extended_array[1:] = mean_prob
390
391     return {
392         "mean_prob": extended_array,
393         "model": model,
394         "trace": trace,
395     }
396
397
398 def main_model(
399     population: pd.DataFrame,
400     stake_size: Union[int, List[int]],
401     group_size: int = 50,
402     committee_size: int = 10,
403     plot_it: bool = False,
404     debug_it: bool = False,
405     verbose: bool = False,
406 ) -> Dict[str, Any]:
407     """
408     Test the Bayesian model by running it and performing posterior
409     predictive checks.
410
411     Args:
412         population (pd.DataFrame): The population of registered SPOs.
413         stake_size (Union[int, List[int]]): The stake size(s) to include
414             in the group. If an integer, it will be converted to a list.
415         group_size (int): The number of participants in the group. Default
416             is 50 for testing.
417         committee_size (int): The number of committee seats. Default is 10
418             for testing.
419         plot_it (bool): Whether to plot the distribution of stake weights.
420             Default is False.
421         debug_it (bool): Whether to debug the model. Default is False.
422         verbose (bool): Whether to print debug information. Default is False.
423
424     Returns:
425         Dict[str, Any]: A dictionary containing the the following key:values
426         - "concentration": The concentration of the Dirichlet distribution.
427         - "mean_prob": The mean probability of each participant.
428         - "model": The PyMC model object.
429         - "trace": The posterior samples obtained from the MCMC sampling.
430     """
431     print("Running the Bayesian model for stake sizes:", stake_size)

```

```

432 print("Group size:", group_size)
433 print("Committee size:", committee_size)
434
435 # Ensure stake_size is a list even if an integer is provided
436 if isinstance(stake_size, int):
437     stake_size = [stake_size]
438
439 # Sample a group of participants with the chosen stake size
440 # at the beginning of the group (at index 1)
441 group = sample_group(
442     population,
443     stake_size,
444     group_size,
445     plot_it=plot_it,
446 )
447 if verbose:
448     print("Sampled group of participants:")
449     print(group.head(10))
450
451 seat_counts = compute_and_adjust_seat_counts(
452     group,
453     committee_size=committee_size,
454     plot_it=plot_it,
455 )
456
457 # Plot the Gamma distribution used as a prior distribution
458 # on the Dirichlet hyperparameter  $\alpha$ 
459 if plot_it:
460     plot_gamma_prior(group_size=group_size)
461
462 # Run the Bayesian model
463 result = run_bayesian_model(
464     committee_size=committee_size,
465     seat_counts=seat_counts,
466     group=group,
467 )
468 model = result["model"]
469
470 # Optional: Debug the model (if debugging is needed)
471 if debug_it:
472     # Perform posterior predictive checks
473     print("Performing posterior predictive checks...")
474     trace = result["trace"]
475     result["ppc"] = predictive_checks(trace, model, plot_it)
476
477 return result
478
479
480 def main(
481     population: pd.DataFrame,
482     stake_size: List[int],
483     committee_size: int = 200,
484     group_size: List[int] = [100, 200, 300],
485     downsample: float = 1.0,

```

```

486     debug_it: bool = False,
487     plot_it: bool = False,
488     verbose: bool = False,
489 ) -> Dict[int, Dict[int, np.ndarray]]:
490     """
491     Main function to run the Bayesian model for different stake sizes
492     and group sizes.
493
494     Args:
495         population (pd.DataFrame): The population of registered SPOs.
496         stake_size (List[int]): The stake size(s) to include in the group.
497             If empty list, a random selection of a stake size is chosen
498             from the observed data.
499         committee_size (int): The number of committee seats. Default is 200.
500         group_size (List[int]): List of group sizes to sample. Default is
501             [100, 200, 300].
502         downsample (float): Fraction of the observed stake sizes to sample.
503             Default is 1.0 (100%).
504         debug_it (bool): Whether to debug the model. Default is False.
505         plot_it (bool): Whether to plot the distribution of stake weights.
506             Default is False.
507         verbose (bool): Whether to print debug information. Default is False.
508
509     Note:
510         Without loss of generality, the committee size can be set to the
511         group size to ensure that the model is not sensitive to the committee
512         size and avoid overfitting. We are inferring the pseudo counts of
513         committee seats for each participant based on their stake size (weight
514         ).
515         Once inferred, the pseudo counts can be used to compute the
516         expected number of committee seats for each participant in the group
517         by normalizing the pseudo counts to sum to 1 and multiplying by
518         the committee size, k.
519
520     Returns:
521         dict: A dictionary containing the psuedo counts for different
522         stake sizes and group sizes.
523     """
524     print("Simulating model for different stake and group sizes...")
525     print("Committee size:", committee_size)
526     print("Group sizes:", group_size)
527     print("Stake sizes:", stake_size)
528     print("Downsample:", downsample)
529     # =====
530     # Get the unique stake sizes in the population and and sort in
531     # descending order. If stake_size is not provided, sample from
532     # the population
533     if len(stake_size) > 0:
534         # Use the provided stake size list
535         unique_stakes = np.array(stake_size)
536     else:
537         # Get the stake size from the population
538         unique_stakes = population["stake"].unique()
539     # Downsample the stake sizes if needed

```

```

539 num_stakes = int(len(unique_stakes) * downsample)
540 assert num_stakes > 0, "No stake size found in the population."
541 # Randomly sample the stake size from the population
542 stake_sizes = np.random.choice(unique_stakes, num_stakes, replace=False)
543 stake_sizes = np.sort(stake_sizes)[::-1] # sort in descending order
544 print(
545     "Number of unique stake sizes in the population:\n"
546     f"\t{num_stakes} out of {len(unique_stakes)} "
547     f"({num_stakes/len(unique_stakes):.1%}) "
548 )
549 # =====
550 # Parallelize the computation using joblib and run the model
551 # for each combination of stake and group size
552
553 def process_stake_group(s, n):
554     """Function to execute for each combination of stake and group size"""
555     print(f"Stake size: {s}")
556     print(f"Group size: {n}")
557     result = main_model(
558         population,
559         stake_size=[s],
560         group_size=n,
561         committee_size=committee_size,
562         plot_it=plot_it,
563         debug_it=debug_it,
564         verbose=verbose,
565     )
566     return s, n, result
567
568 print("Running the model in parallel...")
569 results = Parallel()(
570     # Process each combination of stake size and group size
571     delayed(process_stake_group)(s, n) for s in stake_sizes for n in
group-size
572 )
573 # =====
574 # Store the results in a dictionary
575 # where the keys are the stake sizes and group sizes
576 # and the values are the result dictionaries
577 result_dict = {}
578 for stake, group, data in results:
579     if stake not in result_dict:
580         result_dict[stake] = {}
581     result_dict[stake][group] = data
582
583 return result_dict
584
585
586 def convert_results_to_dataframe(result: dict) -> pd.DataFrame:
587     """
588     Convert a nested results dictionary into a DataFrame with a MultiIndex for
rows.
589     Each row corresponds to a (stake size, group size, data key) triple.
590

```

```

591     Args:
592         result (dict): Nested dictionary with structure:
593             {stake-size: {group-size: {data-key: data-value, ...}, ...}, ...}
594
595     Returns:
596         pd.DataFrame: DataFrame with MultiIndex rows ("Stake Size", "Group
597         Size", "Data").
598         The DataFrame columns are set to range based on the number
599         of pseudo counts.
600     """
601     rows = []
602     cols = []
603     data = []
604     for stake_size_key, groups in result.items():
605         for group_size_key, result_dict in groups.items():
606             for data_key, data_val in result_dict.items():
607                 if data_key in ["model", "trace", "ppc"]:
608                     continue
609                 rows.append((stake_size_key, group_size_key, data_key))
610                 cols.append(data_key)
611                 data.append(data_val)
612     multi_index = pd.MultiIndex.from_tuples(
613         rows, names=["Stake Size", "Group Size", "Data"]
614     )
615     results_df = pd.DataFrame(data, index=multi_index)
616     # Set the column names to [0, 1, ..., max_index] based on the length of
617     # data.
618     results_df.columns = range(results_df.shape[1])
619     return results_df
620
621 # %%
622 # Example usage during testing:
623 if __name__ == "__main__":
624     # =====
625     # Parse command line arguments
626     parser = argparse.ArgumentParser(
627         description=(
628             "Run the group stake distribution calculation with "
629             "optional input/output file paths."
630         )
631     )
632     default_data_dir = Path(__file__).parent.parent / "data"
633     parser.add_argument(
634         "--input-data-file",
635         type=Path,
636         default=default_data_dir / "pooltool-cleaned.csv",
637         help="Path to the input CSV data file (default: %(default)s)",
638     )
639     parser.add_argument(
640         "--output-data-file",
641         type=Path,
642         default=default_data_dir / "bayes_stake_dist_output.csv",

```

```

642         help="Path to the output CSV file (default: %(default)s)",
643     )
644     parser.add_argument(
645         "--downsample",
646         type=float,
647         default=1.0,
648         help="Percentage of the population to sample (0.0-1.0).",
649     )
650     parser.add_argument(
651         "--group-size",
652         type=int,
653         nargs="+",
654         default=100,
655         help="Number of participants in the group (default: 100)",
656     )
657     parser.add_argument(
658         "--committee-size",
659         type=int,
660         default=50,
661         help="Number of seats in the committee (default: 50)",
662     )
663     parser.add_argument(
664         "--stake-size",
665         type=int,
666         nargs="+",
667         default=None,
668         help="List of stake sizes to analyze (default: None)",
669     )
670     parser.add_argument(
671         "--all-in",
672         action="store_true",
673         default=False,
674         help=(
675             "Process all stake sizes specified in '--stake-size' "
676             "simultaneously instead of individually."
677         ),
678     )
679     parser.add_argument(
680         "--plot",
681         action="store_true",
682         default=False,
683         help="Enable plotting of data (default: False)",
684     )
685     parser.add_argument(
686         "--test",
687         action="store_true",
688         default=False,
689         help="Run in test mode",
690     )
691     parser.add_argument(
692         "--seed",
693         type=int,
694         default=None,
695         help="Random number generator seed (default: None)",

```

```

696 )
697 parser.add_argument(
698     "--verbose",
699     action="store_true",
700     default=False,
701     help="Enable verbose output (default: False)",
702 )
703 args = parser.parse_args()
704 # =====
705 # Initialize variables
706 INPUT_DATA_FILE = args.input_data_file
707 OUTPUT_DATA_FILE = args.output_data_file
708 TEST_IT = args.test
709 PLOT_IT = args.plot
710 VERBOSE = args.verbose
711 DOWNSAMPLE = args.downsample
712 GROUP_SIZE = args.group_size
713 COMMITTEE_SIZE = args.committee_size
714 STAKE_SIZE = args.stake_size
715 ALL_IN = args.all_in
716 # =====
717 # Set the random seed for reproducibility
718 if args.seed is not None:
719     random.seed(args.seed)
720     np.random.seed(args.seed)
721 # =====
722 # Load the population data
723 print(f"Input data file: {INPUT_DATA_FILE}")
724 population = load_population(INPUT_DATA_FILE)
725 if DOWNSAMPLE < 1.0:
726     print(f"Downsampling population to {DOWNSAMPLE:.1%} of original size.")
727 )
728     population_size = int(len(population) * DOWNSAMPLE)
729 else:
730     print("Using full population size.")
731     population_size = len(population)
732 print(f"Population size: {population_size}")
733 # Get the smoothed stake distribution of the population
734 # and downsample and plot it if if desired
735 population = get_stake_distribution(
736     population,
737     group_size=population_size,
738     num_iter=1, # 1 for exact instance, >> 1 for smoothed
739     plot_it=PLOT_IT,
740 )
741 # =====
742 # Prepare the stake sizes for processing
743 if isinstance(STAKE_SIZE, list):
744     # If stake size is a list, convert to a list of integers
745     stake_size = [int(s) for s in STAKE_SIZE]
746 elif STAKE_SIZE is None:
747     stake_size = []
748 if ALL_IN:
749     # If all-in mode is enabled, convert to a list of lists so that

```

```

749     # all stake sizes are processed at once in a batch
750     stake_size = [stake_size]
751     # =====
752     if TEST_IT:
753         # Test the model with a specific stake, group, and committee sizes
754         print(
755             f"Testing with stake size {stake_size}",
756             f"and group size {GROUP_SIZE}.",
757         )
758         # Run the model for each combination of stake and group size
759         # and accumulate the psuedo counts.
760         result = {}
761         for stake in stake_size:
762             for group in GROUP_SIZE:
763                 result_sg = main_model(
764                     population=population,
765                     stake_size=stake,
766                     group_size=group,
767                     committee_size=COMMITTEE_SIZE,
768                     plot_it=PLOT_IT,
769                     debug_it=TEST_IT,
770                     verbose=VERBOSE,
771                 )
772                 # Append the result dictionary to the result
773                 if isinstance(stake, list):
774                     for s in stake:
775                         result[s] = {group: result_sg}
776                 else:
777                     result[stake] = {group: result_sg}
778             # =====
779         else:
780             # Main function to run the model
781             result = main(
782                 population,
783                 stake_size=stake_size,
784                 committee_size=COMMITTEE_SIZE,
785                 group_size=GROUP_SIZE,
786                 downsample=DOWNSAMPLE,
787                 debug_it=TEST_IT,
788                 plot_it=PLOT_IT,
789                 verbose=VERBOSE,
790             )
791             # =====
792             # Convert the results to a DataFrame and save to a CSV file
793             results_df = convert_results_to_dataframe(result)
794
795             OUTPUT_DATA_FILE = Path(args.output_data_file)
796             OUTPUT_DATA_FILE.parent.mkdir(parents=True, exist_ok=True)
797             # Ensure the output file has the correct suffix
798             if OUTPUT_DATA_FILE.suffix != ".csv":
799                 OUTPUT_DATA_FILE = OUTPUT_DATA_FILE.with_suffix(".csv")
800             try:
801                 # Save the DataFrames to a CSV file
802                 results_df.to_csv(OUTPUT_DATA_FILE, index=True)

```

```

803         print(f"Pseudo counts prediction saved to file {OUTPUT_DATA_FILE}.")
804
805     except Exception as e:
806         print(f"Error saving result: {e}")
807     if VERBOSE:
808         print("Predictions DataFrame:")
809         print(results_df.head(10))
810     print("Done.")
811     # =====
812     # End of script

```

Listing 2: Computer Code in Python using PyMC

The usage of the script is as follows:

```

1  usage:
2
3  bayes_stake_dist.py [-h] [--input-data-file INPUT_DATA_FILE]
4  [--output-data-file OUTPUT_DATA_FILE] [--downsample DOWNSAMPLE]
5  [--group-size GROUP_SIZE [GROUP_SIZE ...]] [--committee-size COMMITTEE_SIZE]
6  [--stake-size STAKE_SIZE [STAKE_SIZE ...]] [--all-in] [--plot] [--test]
7  [--seed SEED] [--verbose]
8
9  Run the group stake distribution calculation with optional
10 input/output file paths.
11
12 options:
13 -h, --help            show this help message and exit
14
15 --input-data-file INPUT_DATA_FILE
16 Path to the input CSV data file
17 (default: /Users/rjones/Desktop/Work/Shielded/Midnight/midnight-architecture/
18   consensus/risk-modeling/data/pooltool-cleaned.csv)
19
20 --output-data-file OUTPUT_DATA_FILE
21 Path to the output CSV file
22 (default: /Users/rjones/Desktop/Work/Shielded/Midnight/midnight-architecture/
23   consensus/risk-modeling/data/bayes_stake_dist_output.csv)
24
25 --downsample DOWNSAMPLE
26 Percentage of the population to sample (0.0-1.0).
27
28 --group-size GROUP_SIZE [GROUP_SIZE ...]
29 Number of participants in the group (default: 100)
30
31 --committee-size COMMITTEE_SIZE
32 Number of seats in the committee (default: 50)
33
34 --stake-size STAKE_SIZE [STAKE_SIZE ...]
35 List of stake sizes to analyze (default: None)
36
37 --all-in              Process all stake sizes specified in '--stake-size'
38                       simultaneously instead of individually.
39 --plot                Enable plotting of data (default: False)
40 --test               Run in test mode
41 --seed SEED          Random number generator seed (default: None)
42 --verbose             Enable verbose output (default: False)

```

Listing 3: Command to run the script

A helper script to run the Bayesian model with various parameters is provided below. This script can be executed in a terminal or command line interface.

```

1 #!/bin/bash
2 # Script to execute the participation_gen.py script
3 # with various parameters for committee size, group size, and stake.
4
5 ./bayes_stake_dist.py \
6     --committee-size 350 \
7     --group-size 100 200 300 400 500 600 \
8     --stake-size 10 100 1_000 10_000 100_000 1_000_000 10_000_000 100_000_000 \
9     --seed 123 \
10    --verbose
11
12 echo "Participation generation completed."

```

Listing 4: Script to execute the Bayesian model with various parameters

E Computation Results

The results of the Bayesian model for different stake sizes and group sizes are stored in a nested dictionary. The keys are the stake sizes and group sizes, and the values are the result dictionaries containing the posterior samples, model, and other relevant data.

The table lists the mean probability distribution, $\mathbf{m} = [m_1, m_2, \dots, m_n]$, for columns $1, 2, \dots, n$ over the probabilities of committee selection, \mathbf{p} , for different stake sizes and group sizes, n . These parameters can be multiplied by the committee size k to calculate the expected number of times the associated participant is selected to the committee.

The role of the parameter m_0 at column 0 can be interpreted as a measure of the *sharpness* (or precision) of the distribution characterized by parameters \mathbf{m} . That is, while the $m_i : i \in \{1, 2, \dots, n\}$ parameters represent the expected probabilities of selection for each participant, such that

$$\sum_{i=1}^n m_i = 1 \quad (58)$$

the m_0 measures how different we expect typical samples \mathbf{p} to be from the mean \mathbf{m} . A large value of m_0 produces a distribution over \mathbf{p} that is sharply peaked around \mathbf{m} .

Table 1: Mean Probability by Stake Size, Group Size, and Posterior Predictions

Parameters			Mean Concentration of Probability						
Stake Size	Group Size		0	1	2	3	4	5	6
10 ⁸	600		1176.23	0.0046	0.0038	0.0038	0.0038	0.0038	0.0038
10 ⁷	600		1176.97	0.0021	0.0038	0.0038	0.0038	0.0038	0.0035
10 ⁶	600		1176.56	0.0014	0.0038	0.0038	0.0038	0.0038	0.0034
10 ⁵	600		1176.04	0.0014	0.0037	0.0038	0.0038	0.0038	0.0038
10 ⁴	600		1175.74	0.0014	0.0039	0.0038	0.0038	0.0039	0.0038
10 ³	600		1175.24	0.0014	0.0038	0.0038	0.0038	0.0038	0.0038