Efficient Evaluation of Multinomial Probabilities in Differentially Private Synthetic Data Verification Servers

Tom Balmat, Andrés F. Barrientos, Jerome P. Reiter*

May 31, 2018

Abstract

As private data sets are made accessible to the public, methods to limit the risk of identifying individuals represented in the data become increasingly necessary. These include construction and release of representative synthetic data and implementation of verification servers.[1] In both of these treatments authentic data are withheld from public view and either alternative, statistically transformed data are released that are designed to maintain important distributional and covariate relationships while limiting opportunity of individual identification (synthetic data) or no data are released, but a system of inquiry is implemented to fit statistical models to synthetic and/or authentic data while reporting results that are statistically adjusted to balance integrity of results (accuracy of coefficient estimates, standard errors, etc.) with limiting risk of individual disclosure. This paper presents a hierarchical Laplace-Dirichlet-Multinomial model used in statistical adjustment of results by generating a differentially private posterior distribution of the proportion of random data partitions that have a given verification property.[2] [3] [4] The model is used in the authors' research and, although having high utility, is found to be computationally intensive to evaluate, especially when high numbers of iterative evaluations are required for complete distributional representation.[5] The analytical form of the posterior distribution is derived, efficiency inhibitors are diagnosed, and two alternative numeric algorithms are presented: one that makes straightforward, but inefficient, use of standard R functions, and a more efficient implementation that requires additional processing, but replaces redundant evaluation of computationally intensive multinomial probabilities with a compute-once-and-recycle strategy. When used with typical partitioning schemes, the improved algorithm reduces computation time to 1/50th of that exhibited by the standard R algorithm.

1 Introduction

To limit the risk of disclosure of sensitive information regarding individuals or subjects that participate in a data set, private data can be to transformed into representative synthetic data using a variety of controlled randomization and perturbation techniques that balance utility and retention of key covariate relationships with risk of individual identification. A fully random data set would possess low utility and a high degree of protection, while unaltered data achieves maximum utility, but with no potential for privacy protection beyond what the original data possess. To further reduce risk of individual identification, synthetic data can be withheld from direct public access and a verification server can be configured to fit analyst specified models to both authentic and synthetic data then report a summarized comparison of resulting parameter estimates

*Author contact info, grant support info

[1] Reference SDL and verification server resources

[2] Differentially privacy is discussed in greater detail later.

[3] In this case, the data are randomly partitioned by groups of individuals, partitions are balanced, and each individual appears strictly in one partition.

[4] As an example, verification could imply that the proportion of partitions with a given model parameter estimate within 0.01 of the "true" estimate (as measured in the entire data set) is above a specified level.

[5] reference race verification paper

using algorithms that satisfy ϵ-differential privacy.[6] Although probability mass and density functions used to generate differentially private analyses may involve useful, closed form prior distributions such as the Beta, binomial, Dirichlet, and multinomial, when combined with the Laplace (double exponential) distribution, posterior probability distributions may lack any apparent closed form, requiring iterative simulation methods for evaluation. Such methods may require thousands of iterations, each computing hundreds to thousands of binomial or multinomial coefficients and probability function evaluations. This article presents a hierarchical ϵ-DP posterior distribution for proportions that involves Dirichlet prior and multinomial probabilities. Two evaluation scripts are provided: one using the standard R `dmultinom()` function for multinomial probabilities and an improved one that eliminates redundant computation of multinomial coefficients, by using a single instance of a constructed coefficient table, while computing posterior probabilities within a C function compiled using `Rcpp()`.[7] [8] A comparison of the numerical values generated by both scripts is made across the possible input domain, algorithm efficiency opportunities are explored, and execution times required to solve typical problems are presented, with the improved algorithm demonstrating superior performance.

Explain requirement to simulate verification measure results for sensitivity analysis that prompted development of alternative computational algorithm.

2 ϵ-Differential Privacy

- Introduce concepts of differential privacy with references
- Explain role of ϵ

3 Data Partitioning, Regression Models, Threshold Verification Measure, and Partition Counts

- Explain partitioning rationale and role in protecting privacy (*references*)
- Explain regression threshold parameter verification measure (*references*)
- Define M, S_1, S_0, and S_{err}
- Define S_{n_1}, S_{n_0}, and $S_{n_{err}}$ (noisy substitutes for observed S_1, S_0, and S_{err})

4 ϵ-DP Laplace-Dirichlet-Multinomial Posterior Distribution of Partition Proportions

Let S be a vector of S_1, S_0, and S_{err} partitions, such that their sum is the total number of partitions M, and let $p = (p_1, p_0, p_{err})^{\mathrm{T}}$ be a vector of probabilities of simultaneously observing S_1, S_0, S_{err}, respectively (note that $p_1 + p_0 + p_{err} = 1.0$). The objective is to generate a simulated posterior distribution of p given observed M and S using a Dirichlet(α) prior distribution for p, where α is a vector of α_1, α_0, and α_{err} parameters, and a multinomial(M, p) distribution for S. If ϵ-DP results were not an objective, then the resulting posterior distribution for p would have closed form and be relatively simple to evaluate.[9] However, as seen in equations (1) through (4), the use of Laplace probabilities to perturb S and achieve ϵ-DP produces a posterior distribution that requires numeric, as opposed to analytic, evaluation.[10] Derivation of the posterior distribution of $p|S$ follows.

How does introduction of the Laplace guarantee DP? Do we need to prove this, or is there a reference that establishes this?

[6] ϵ-differential privacy implies an upper bound of e^{ϵ} on the ratio of each value in the image of an algorithm A evaluated with data for a given individual present in the data to the corresponding value in the image of A with data for the individual removed. In what follows, ϵ-DP will denote either ϵ-differential privacy or ϵ-differentially private.

[7] Reference `dmultinom()` resources.

[8] Reference `Rcpp` resources.

[9] Cite a reference on the Dirichlet-Multinomial distribution.

[10] The Laplace, or double exponential, distribution has pdf $f(x, x_0, \epsilon) = \frac{\epsilon}{2} e^{-\epsilon|x-x_0|}$. We will employ simultaneous Laplace probabilities for three independent random values, giving $f(x, y, z, x_0, y_0, z_0, \epsilon) = \frac{\epsilon^3}{8} e^{-\epsilon(|x-x_0|+|y-y_0|+|z-z_0|)}$.

Given M, S, p, and α as described above, a value of ϵ to achieve a desired upper bound on privacy loss, and an initial vector $S_n = (S_{n_1}, S_{n_0}, S_{n_{err}})^{\mathrm{T}}$ of Laplace perturbed S values, the pdf of $S|p$ is the product of the Laplace and multinomial pdfs evaluated at S and p divided by the sum of these products for all S, to maintain cumulative density of 1.0. This appears in equation (1).

$$f_{S|p}(S|p) = \frac{\frac{\epsilon^3}{8} e^{-\epsilon(\Sigma|S_i - S_{n_i}|)} \binom{M}{S_1} \binom{M-S_1}{S_0} p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}}{\frac{\epsilon^3}{8} \sum_S e^{-\epsilon(\Sigma|S_i - S_{n_i}|)} \binom{M}{S_1} \binom{M-S_1}{S_0} p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}} \tag{1}$$

The prior (Dirichlet) distribution of $p|\alpha$ is given as

$$f_{p|\alpha}(p|\alpha) = \frac{\Gamma(\sum \alpha_i)}{\prod \Gamma(\alpha_i)} \prod p_i^{\alpha_i - 1} \tag{2}$$

Using the conditional relationship f(S, p)=$f_{p|\alpha}(p|\alpha) f_{S|p}(S|p)$, the joint distribution of S and p is

$$\begin{aligned} f(S,p) &= f_{p|\alpha}(p|\alpha) f_{S|p}(S|p) \\ &= \frac{\Gamma(\sum \alpha_i)}{\prod \Gamma(\alpha_i)} \prod p_i^{\alpha_i - 1} \frac{\frac{\epsilon^3}{8} e^{-\epsilon(\Sigma|S_i - S_{n_i}|)} \binom{M}{S_1} \binom{M-S_1}{S_0} p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}}{\frac{\epsilon^3}{8} \sum_S e^{-\epsilon(\Sigma|S_i - S_{n_i}|)} \binom{M}{S_1} \binom{M-S_1}{S_0} p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}} \\ &= \frac{\Gamma(\sum \alpha_i)}{\prod \Gamma(\alpha_i)} \frac{e^{-\epsilon(\Sigma|S_i - S_{n_i}|)} \binom{M}{S_1} \binom{M-S_1}{S_0} p_1^{S_1+\alpha_1-1} p_0^{S_0+\alpha_0-1} p_{err}^{S_{err}+\alpha_{err}-1}}{\sum_S e^{-\epsilon(\Sigma|S_i - S_{n_i}|)} \binom{M}{S_1} \binom{M-S_1}{S_0} p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}} \end{aligned} \tag{3}$$

The posterior distribution of $p|S$ is then $f_{p|S}(p|S) = \frac{f(S,p)}{f_S(S)}$, where $f_S(S)$ is the marginal distribution of S, given by

$$f_S(S) = \frac{\Gamma(\sum \alpha_i)}{\prod \Gamma(\alpha_i)} e^{-\epsilon(\Sigma|S_i - S_{n_i}|)} \binom{M}{S_1} \binom{M-S_1}{S_0} \int_0^1 \frac{p_1^{S_1+\alpha_1-1} p_0^{S_0+\alpha_0-1} p_{err}^{S_{err}+\alpha_{err}-1}}{\sum_S e^{-\epsilon(\Sigma|S_i - S_{n_i}|)} \binom{M}{S_1} \binom{M-S_1}{S_0} p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}} dp \tag{4}$$

Although the coefficient preceding the integral in equation (4) is canceled when dividing $f(S,p)$ and $f_S(S)$, the integral remains and it is this component of the posterior distribution of $p|S$ that necessitates numeric evaluation. Note that in the derivation of $f_{p|S}(p|S)$ above, S_n was treated as non-stochastic when, in practice, to achieve ϵ-DP, S_n is randomly drawn from a Laplace(ϵ) distribution. This further complicates the resulting form of $f_{p|S,S_n}(p|S,S_n)$, making numeric evaluation methods realistically unavoidable.

5 Simulation of Posterior Distribution of p

Provide rationale for the method employed:

- Alternating evaluation of Dirichlet-multinomial probabilities behaves like a Markov process (*reference*)
- Retaining generated Dirichlet distributed p values simulates $p|S$

5.1 Algorithm

1. Compute total number of partitions, M, as sum of observed $(S_1, S_0, S_{err})^{\mathrm{T}}$ passed in S parameter
2. Construct matrix S_3 of multinomial triplets, with columns S_{3_1}, S_{3_0}, and $S_{3_{err}}$ and one row for each permutation of integers $S_1, S_0, S_{err} \in \{0, ..., M\}$ such that $S_1 + S_0 + S_{err} = M$
3. Generate Laplace(ϵ) distributed $S_n = (S_{n_1}, S_{n_0}, S_{n_{err}})^{\mathrm{T}}$ using values of $(S_1, S_0, S_{err})^{\mathrm{T}}$, and ϵ passed in S and *epsilon* parameters
4. Compute vector of Laplace(ϵ) densities, one for each row of S_m, offset from S_n

   $$f_l(S_3) = \frac{\epsilon^3}{8} e^{-\epsilon(|S_{3_1} - S_{n_1}| + |S_{3_0} - S_{n_0}| + |S_{3_{err}} - S_{n_{err}}|)}$$

5. Generate initial Dirichlet(α) distributed value for p, where $\alpha =$ observed $(S_1, S_0, S_{err})^{\mathrm{T}}$ passed in S
6. Execute the following for 5,000 iterations

   6.1 Compute vector of multinomial(p) probabilities, one for each row of S_3, using current value of p

   $$f_3(S_3) = \binom{M}{S_{3_1}} \binom{M - S_{3_1}}{S_{3_0}} p_1^{S_{3_1}} p_0^{S_{3_0}} p_{err}^{S_{3_{err}}}$$

   6.2 Compute S_3 row sampling weights w_i as product of multinomial $f_3(S_3)$ row probabilities and Laplace $f_l(S_3)$ row densities

   6.3 Randomly sample one row S_j from S_3 using w_i weights

   6.4 Generate new Dirichlet(α) distributed value for p, where $\alpha = (S_{j_1}, S_{j_0}, S_{j_{err}})^{\mathrm{T}}$

   6.5 Save p

7. Compute mode of final 4,000 generated p values
8. Return vector of final 4,000 p values along with corresponding mode

5.2 Implementation of Posterior p Algorithm Using Standard R Functions

Following is an R script that implements the preceding Laplace-Dirichlet-Multinomial posterior p differential privacy algorithm. *Discuss features. Explain function parameters and return values.*

```
library(LaplacesDemon)

# Differential Private Laplace-Dirichlet-Multinomial Algorithm

DP.threshold <- function (S, epsilon, alpha) {

  M = sum(S)              # Number of subgroups

  # Range of S
  RangeS = matrix(0,ncol=3,nrow=(M+1)^3)
  j=0
  for(j1 in 0:M)
    for(j2 in 0:M)
      for(j3 in 0:M) {
        j=j+1
        RangeS[j,]=c(j1,j2,j3)
      }
  RangeS = RangeS[apply(RangeS,1,sum)==M,]

  nS = S + rlaplace(3,0,2/epsilon) # noisy S

  dslap <- dlaplace(nS[1],RangeS[,1],2/epsilon,log = T) +
           dlaplace(nS[2],RangeS[,2],2/epsilon,log = T) +
           dlaplace(nS[3],RangeS[,3],2/epsilon,log = T)

  # Initial value of p (multinomial parameter)
  p = rdirichlet(1,c(S[1], S[2], S[3]))

  # Gibbs interaction
  f.iter = function(j) {
    prob = apply(RangeS, 1, function(k, p) dmultinom(k, prob=p, log=T), p=p)
    + dslap
    prob = exp(prob - max(prob))
    s <- RangeS[sample(1:nrow(RangeS),1,prob=prob),]
    # Update p
    # Note the addition of alpha - this prevents non-stochastic return of 0 for S
    # positions that are 0, or return of 1 for S when other two are 0
    p <<- rdirichlet(1,alpha+s)
    c(s,p)
  }

```

```
42   # MCMC evaluation of posterior distribution
43   pDP <- t(sapply(1:5000, f.iter)[,1001:5000])
44
45   # Return S1, S0, Se, p1, p0, and pe vectors and modes of p1, p0, pe
46   # Note that p1 and p0 are computed as conditional on not(pe)
47   pDP[,4:5] <- pDP[,4:5]/(pDP[,4]+pDP[,5])
48   list("Mode"=c("p1"=frequencyMode(pDP[,4]),
49                 "p0"=frequencyMode(pDP[,5]),
50                 "pe"=frequencyMode(pDP[,6])),
51        "pDP"=pDP)
52
53 }
```

5.3 Example Posterior Distributions from Standard R Algorithm

Describe features of example posterior p distributions in figure 1. Explain the role of the mode.

5.4 Standard R Implementation Efficiency Diagnosis

Inspection of the algorithm and corresponding R instructions reveals computation of multinomial probabilities for each permutation of S_1, S_0, S_{err} that sum to M (each row of S_3 in step 6.1 of the algorithm and each row of `RangeS` passed to `dmultinom()` on line 30 of the R script) in each of 5,000 iterations (part 6 of the algorithm and lines 30 through 40 of the R script). It is easy to establish that the computationally intensive portion of this is in evaluating the corresponding multinomial coefficients[11] and, since S_3 and `RangeS` do not change once they are constructed (in step 2 of the algorithm and lines 10 through 17 of the R script), their corresponding multinomial coefficients do not change. Recomputing these computationally expensive values that, in fact, do not change is inefficient. For a common value of M=50, there are 1,326 permutations of S_1, S_0 and S_{err} that sum to M and, in 5,000 iterations, this gives 1,326 × 5,000 = 6,630,000 coefficient computations, each involving several factorials on the order of $M!$. Only 1/5,000 (0.02%) of these are unique and necessary yet, in the R script presented, all are computed.

6 Improved Posterior p Algorithm

Having identified redundant computation of multinomial coefficients as the primary inefficiency of the algorithm presented in section 5.1, we now develop a means of computing each coefficient once to be referenced as needed in subsequent computations. Two conceptually simple modifications to the algorithm are to insert line 2.1, to compute the multinomial coefficient for each permutation of S_1, S_0, S_{err}, and modify line 6.1, replacing redundant computation of coefficients with retrieval of those that were previously computed, as follows:

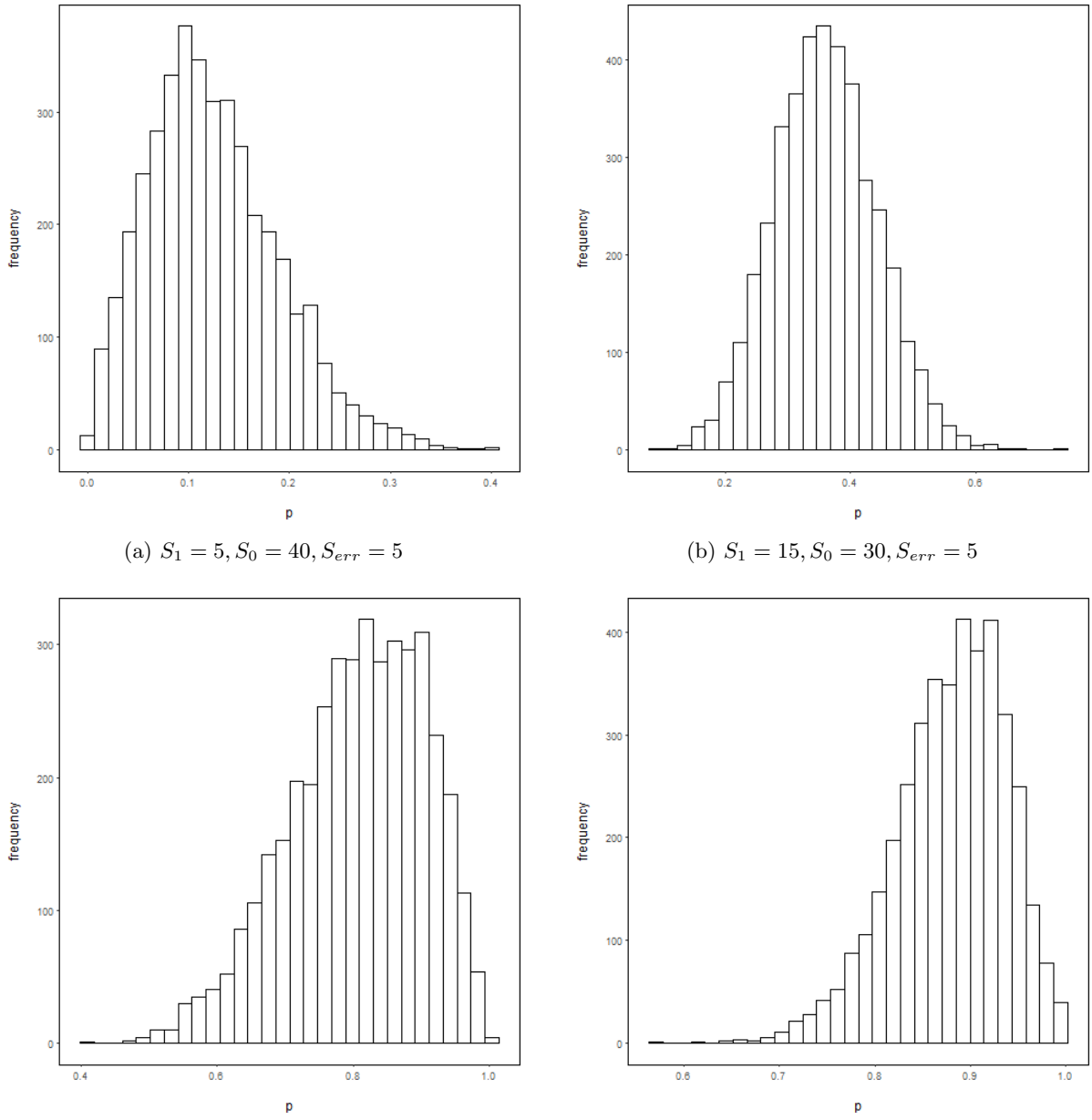
2.1 Compute the multinomial coefficient $\binom{M}{S_{3_1}}\binom{M-S_{3_1}}{S_{3_0}}$ for and append to each row of S_3

6.1 $f_3(S_3) = computed\_S_3\_coefficient \times p_1^{S_{3_1}} p_0^{S_{3_0}} p_{err}^{S_{3_{err}}}$

6.1 Construction of the Multinomial Coefficient Table

Construction of S_3 was presented in section 5.1. S_3 has three columns, one for each S_1, S_0, and S_{err} partition count ($S_1, S_0, S_{err} \in \{0...M\}$), and one row for each permutation of S_1, S_0, S_{err} such that $S_1+S_0+S_{err} = M$. The objective is to efficiently compute, for each row of S_3, the multinomial coefficient corresponding to the S_1, S_0, S_{err} entries of that row, that is $\binom{M}{S_1}\binom{M-S_1}{S0}$, and save the result in an appended, fourth column of S_3. Note that, for each S_1, S_0, S_{err}, $\binom{M}{S_1}\binom{M-S_1}{S0} = \frac{M!}{S_1!(M-S_1)!}\frac{(M-S_1)!}{S_0!(M-S_1-S_0)!} = \frac{M!}{S_1!S_0!S_{err}!}$. Arranging permutations of S_1, S_0, S_{err} as in table 1 and assigning the trivial multinomial coefficient value for row 1

[11] For instance, by comparing execution times of 5,000 iterations each of 1.) `dmultinom(x=c(10, 20, 30), prob=c(0.20, 0.30, 0.50)` and 2.) `0.20**10 * 0.30**20 * 0.50**30`, the latter being equivalent to the multinomial probability divided by, or lacking, its coefficient.



(a) $S_1 = 5, S_0 = 40, S_{err} = 5$

(b) $S_1 = 15, S_0 = 30, S_{err} = 5$

(c) $S_1 = 30, S_0 = 15, S_{err} = 5$

(d) $S_1 = 40, S_0 = 5, S_{err} = 5$

Figure 1: Example posterior distributions of proportion S_1 partitions. $M = 50, \epsilon = 1$, observed S_1, S_0, and S_{err} as indicated in captions.

permits replacement of expensive multinomial coefficient computation with simple multiplication of prior computed coefficients by the ratio of two integers, each a function of corresponding row number. It is seen in the *computation* column that the multinomial coefficient of the S_1, S_0, S_{err} values in row i is the multinomial coefficient in row $i-k$ times $(M-k+1)/k$, where k is the second term of $M-k$ and appears in the column labeled k. It is the *computation* column that is appended to S_3.

Table 1: Permutations of S_1, S_0, S_{err} partition counts ($S_1 + S_0 + S_{err} = M$) with corresponding multinomial coefficients, in order of computational efficiency.

i	k	$S_1 = M - k$	S_0	S_{err}	mc=$\binom{M}{S_1}\binom{M-S_1}{S_0}$	computation
1	0	M	0	0	$\binom{M}{M}\binom{0}{0}$	$\frac{M!}{M!0!0!} = 1$
2	1	$M-1$	0	1	$\binom{M}{M-1}\binom{1}{0}$	$\frac{M!}{(M-1)!0!1!} = M$
3	1	$M-1$	1	0	$\binom{M}{M-1}\binom{1}{1}$	$\frac{M!}{(M-1)!1!0!} = M$
4	2	$M-2$	0	2	$\binom{M}{M-2}\binom{2}{0}$	$\frac{M!}{(M-2)!0!2!} = \text{mc}[2](M-1)/2 = \text{mc}[i-k](M-k+1)/k$
5	2	$M-2$	1	1	$\binom{M}{M-2}\binom{2}{1}$	$\frac{M!}{(M-2)!0!2!} = \text{mc}[3](M-1)/1 = \text{mc}[i-k](M-k+1)/(k-1)$
6	2	$M-2$	2	0	$\binom{M}{M-2}\binom{2}{2}$	$\frac{M!}{(M-2)!2!0!} = \text{mc}[i-k]$
7	3	$M-3$	0	3	$\binom{M}{M-3}\binom{3}{0}$	$\frac{M!}{(M-3)!0!3!} = \text{mc}[4](M-2)/3 = \text{mc}[i-k](M-k+1)/k$
8	3	$M-3$	1	2	$\binom{M}{M-3}\binom{3}{1}$	$\frac{M!}{(M-3)!1!2!} = \text{mc}[5](M-2)/2 = \text{mc}[i-k](M-k+1)/(k-1)$
9	3	$M-3$	2	1	$\binom{M}{M-3}\binom{3}{2}$	$\frac{M!}{(M-3)!2!1!} = \text{mc}[6](M-2)/1 = \text{mc}[i-k](M-k+1)/(k-2)$
10	3	$M-3$	3	0	$\binom{M}{M-3}\binom{3}{3}$	$\frac{M!}{(M-3)!3!0!} = \text{mc}[i-k]$
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Although our problem involves counts of data partitions in three categories (verification measure criteria met, verification criteria not met, and model fitting error or indeterminate verification result), the above method can be extended to any number of categories by considering (from induction) that, for integers $x_1, x_2, \ldots, x_k$ such that $x_1 + x_2 + \cdots + x_k = M$, if

$$\binom{M-x_k}{x_1}\binom{M-x_k-x_1}{x_2}\binom{M-x_k-x_1-x_2}{x_3}\cdots = \frac{(M-x_k)!}{x_1!x_2!x_3!\ldots x_{k-1}!}$$

then

$$\binom{M}{x_k}\binom{M-x_k}{x_1}\binom{M-x_k-x_1}{x_2}\binom{M-x_k-x_1-x_2}{x_3}\cdots = \frac{M!}{x_k!(M-x_k)!}\frac{(M-x_k)!}{x_1!x_2!x_3!\ldots x_{k-1}!} = \frac{M!}{x_1!x_2!\ldots x_k!}.$$

Since the left hand expression is the multinomial coefficient for a k-variable problem, the expression on the right can be used to construct a corresponding k-category partition coefficient table. Beginning with the trivial results $\frac{M!}{M!0!\ldots0!} = 1$ in row 1 and $\frac{M!}{(M-1)!1!0!\ldots0!} = M, \ldots, \frac{M!}{(M-1)!0!\ldots0!1!} = M$ in rows 2 through k, construction of subsequent rows proceeds using products of values from previously computed rows and the ratio of two integers, both simple linear functions of row index.

6.2 Validation of Multinomial Coefficients and Probabilities

Before using the multinomial coefficients constructed in section 6.1 to make the proposed modifications to the algorithm presented in section 5.1, it is important to verify that they can be used to produce accurate multinomial probabilities. To do this, we will compare products of the generated coefficients and random, exponentiated values of p_1, p_0, and p_{err} with corresponding probabilities given by the R `dmultinom()` function. Specifically, we are interested in the number of significant digits in agreement between $\binom{M}{S_1}\binom{M-S_1}{S_0}p_1^{S_1}p_0^{S_0}p_{err}^{S_{err}}$ and `dmultinom(x=c(`S_1, S_0, S_{err}`), prob=c(`p_1, p_0, p_{err}`))` for all permutations of S_1, S_0, and S_{err}.[12] Figure 2 plots, for 10,000 jointly random values of p_1, p_0, and p_{err} and each S_1S_0, S_{err} permutation, the mini-

[12]Significant digits, the mantissa, are the leading, non-zero digits of a floating point number. Significant digits in agreement between probabilities p and q is computed as $\log_{10}(p) - \log_{10}(|p-q|)$.

mum number of significant digits in agreement between multinomial probabilities computed using coefficients and those returned by `dmultinom()`.[13] Although there exists one multinomial probability for each $S_1, S_0, S_{err}, p_1, p_0$, and p_{err}, differences are plotted against individual p_1, p_0, and p_{err} values to identify potential bias by probability level. There does appear to be slight bias of approximately 0.2 digits near the 0 and 1 endpoints. However, all differences are beyond the twelfth significant digit, indicating good agreement between multinomial probabilities computed using these two methods. Another feature apparent in figure 2 is a reduction in density of points with increase in p. This is a result of p_1, p_0, and p_{err} being constrained to a sum of 1.0. For each triplet with one p greater than 0.5, the remaining two must be less than 0.5; if one is greater than 0.75, the remaining two must be less than 0.25; etc.

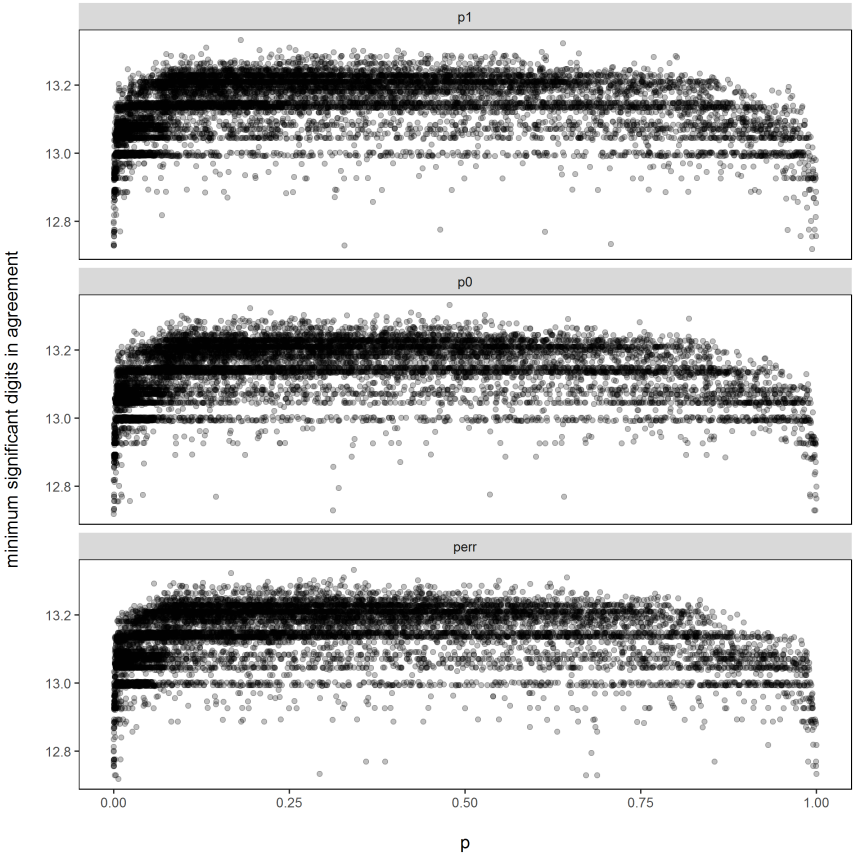


Figure 2: Minimum number of significant digits in agreement between multinomial probabilities computed with tabulated coefficients and those returned by `dmultinom()`. 10,000 iterations using all permutations of S_1, S_0, S_{err} and random values of p_0, p_1, p_{err}.

6.3 Efficient Computation of $p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}$

The coefficients composed in section 6.1 provide the combinatorial factor, $\binom{M}{S_1}\binom{M-S_1}{S_0}$, required to compute multinomial probabilities. Before assessing the efficiency of these coefficients, we first assess the efficiency of computing the remaining multinomial factor, $p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}$. Given that `S` and `p` are each vectors of length three, where $\texttt{S}=(S_1, S_0, S_{err})^{\mathrm{T}}$ and $\texttt{p}=(p_1, p_0, p_{err})^{\mathrm{T}}$, three immediate methods of computation are apparent,

[13] Random p_1, p_0, p_{err} triplets were generated with three stage uniform intervals where one p value is is randomly drawn from [0,1], a second p is drawn from [1-*first_p*, *first_p*], and the third is set to 1-*first_p*-*second_p*. The resulting p values sum to 1.0 and are randomly assigned to p_1, p_0, and p_{err}.

two using standard R instructions and one developed in C:[14]

- Method 1 (R): `result <- p[1]**S[1] * p[2]**S[2] * p[3]**S[3]`
- Method 2 (R): `result <- exp(S[1]*log(p[1]) + S[2]*log(p[2]) + S[3]*log(p[3]))`
- Method 3 (C): `result = exp(S[0]*log(p[0]) + S[1]*log(p[1]) + S[2]*log(p[2]));`[15]

Table 2 compares the time to execute 100,000 iterations of each method using all permutations of S_1, S_0, and S_{err}, with a sum of M=50, and randomly generated triplets of p_1, p_0, and p_{err} that sum to 1.0.[16] Since comparison of performance is of interest, computation time is given as a proportion of the time required for method 1, the most intuitive and standard of the three. It is clear from the results that use of logarithms improves efficiency of computing products of exponentiated probabilities and method 3, employing a compiled C function, outperforms both R methods tested.

Table 2: Mean time to compute 1,000,000 $p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}$ values. Results in proportion to those of Method 1.

Method	Total Compute Time (proportional to method 1)
1	1.00
2	0.47
3	0.29

6.4 Implementation of Multinomial Coefficient Table with C Function Posterior p Evaluation

Having established the efficiency and accuracy of computing multinomial probabilities using coefficients as composed in section 6.1 and the efficiency of evaluating products of exponentiated probabilities using a compiled C function, we now combine these methods to implement the improved posterior p algorithm. The following script creates an R function, `DP.threshold.multinomial()`, which constructs the multinomial permutation and coefficient table corresponding to $M = S_1 + S_0 + S_{err}$, where S_1, S_0, and S_{err} are supplied in the `S` (vector) parameter, and iteratively computes 5,000 Dirichlet(s_1, s_0, s_{err}) distributed posterior p values, where s_1, s_0, s_{err} is a permutation sampled using weights corresponding to joint Laplace, multinomial probabilities, where multinomial probabilities are computed in the compiled C function `pMultinomPermS()`.[17]

Explain function parameters and return values. Explain `epsilon` *and* `alpha`. *Note the conditional probabilities of* p_1 *and* p_0 *given not* p_{err}.

Overview:

- Line 51: Load `LaplacesDemon` package for Dirichlet and Laplace probability functions[18]
- Line 52: Load `CommonFunctions` library for `mode()` function[19]
- Lines 57 through 103: Construct multinomial coefficient table, `permS`, using S_1, S_0, and S_{err} supplied in `S` parameter
- Line 106: Generate random Laplace(ϵ) distributed S_{n_1}, S_{n_0}, and $S_{n_{err}}$ centered at S_1, S_0, and S_{err}, respectively
- Line 111: Compute vector of Laplace densities, `dslap`, one density for each permutation in the multinomial coefficient table, using independent Laplace distributions for each of S_1, S_0, S_{err} centered at S_{n_1}, S_{n_0}, and $S_{n_{err}}$, respectively

[14]The R package `Rcpp` is used to integrate C functions with R scripts. For more on `Rcpp`, see *Reference* ***`Rcpp`*** *package*.
[15]Note that C arrays are 0-based.
[16]Results are mean computation time of ten groups of 100,000 products.
[17]Reference github link to R package containing `DP.threshold.multinomial()` and `pMultinomPermS()` functions.
[18]Reference
[19]Replace this with Duke DP function package

- Line 114: Generate initial Dirichlet distributed p_1, p_0, and p_{err} using $S_1 = 1, S_0 = 1$, and $S_{err} = 1$ (although this is not a valid permutation with sum of M, it is an accepted initial value for MCMC simulation[20])
- Lines 119 through 129: Compute 5,000 random posterior p values
  - Line 122: Compute vector of S_1, S_0, S_{err} permutation sampling weights, `prob`, as product of multinomial probabilities (computed in C function `pMultinomPermS` using `permS` coefficients and current values of p_1, p_0, and p_{err}) and corresponding `dslap` permutation Laplace densities
  - Line 125: Sample a permutation, s_1, s_0, s_{err} from `permS` using vector of weights `prob`
  - Line 127: Generate random Dirichlet(s_1, s_0, s_{err}) distributed p_1, p_0, p_{err} triplet (note the use of `alpha` $= (1, 1, 1)^{\mathrm{T}}$ as a buffer to prevent results being trapped at 0.0 or 1.0 when one p value is exactly 1.0)
  - Line 129: Return the final 4,000 generated p triplets, rows 1,001 through 4,000 of resulting matrix `pDP`
- Lines 134 through 136: Compute modes of simulated p_1, p_0, and p_{err} distributions using the means of the five most frequent cells of 250 cell histograms (note that the modes of p_1, p_0, and p_{err} are returned as `r-hat`, `p0-hat`, and `e-hat`, respectively)
- Line 137: Include the generated p_1, p_0, and p_{err} distributions (columns 1, 2, 3 of the `pDP` matrix) in the result list

R Script:

```
1  # Duke University Synthetic Data Project
2  # Differential Private Threshold Verification Measure Multinomial Noise Algorithm
3
4  # This implementation composes a table of all permutations of partitions at threshold (S1),
5  # partitions not at threshold (S0), and partitions with non-computable model estimates (Se,
6  # e for error) such that S1+S0+Se=M, the number of partitions. The multinomial coefficient
7  # (product of combinations involving S1, S0, and Se) is computed for each permutation.
8
9  # For each number of partitions, M, there exists a finite list of permutations of S1, S0,
10 # and Se, each with a corresponding multinomial coefficient. The coefficients are
11 # computationally expensive, each involving multiple factorials and since each coeffifient
12 # is used once per iteration (the current implementation involes 5,000 iterations) it is
13 # efficient to compute coefficients once only. Note that this is an alternative to
14 # repeated, independent calls to dmultinomial() which, presumably, must recompute coef-
15 # ficients since it has no knowledge or memory of what was computed during past cycles.
16
17 # Multinomial probabilities are used as sampling weights in selecting random S1, S0, and Se
18 # triplets. The weights are computed as the product of the multinomial coefficients and
19 # corresponding exponentiated random (Dirichlet) probabilities p1, p0, and pe. One weight
20 # is computed per S1, S0, Se permutation as mc * p1**S1 * p0**S0 * pe**Se, where mc is the
21 # multinomial coefficient for S1, S0, and Se. Computing the exponentiated values is also
22 # expensive (for M=50 there are 1326 permutations with three exponentiated p values for
23 # each, and products are iterated 5,000 times - computation time accumulates). As an
24 # alternative to using apply() to compute weights, a C function (pMultinomPermS) computes
25 # mc * p1**S1 * p0**S0 * pe**Se from permutation table values. In trials, it is 50 times
26 # efficient than corresponding apply() implementations.
27
28 ##########################################################################################
29 #### Create differential privacy noise function using multinomial probability algorithm
30 ##########################################################################################
31
32 DP.threshold.multinomial <- function (S, epsilon, alpha) {
33
34   # Parameters:
35   # S ......... Three element vector containing partitions at threshold (S1) in position one,
36   #             partitions not at threshold (S0) in position two, and partitions with non-
37   #             computable model estimates (Se) in position three
38   #             Note that M, number of partitions is computed as M=S1+S0+Se
39   # epsilon ... Laplace privacy parameter (scalar)
40   # alpha ..... Dirichlet buffer (three position vector), (protects against random values
41   #             stalling and becoming trapped near 0 or 1
42
43   # Result is a two element list:
44   # Element one ... A three element vector containing the modes of the posterior p1, p0, and
45   #                 pe vectors (in positions 1, 2, and 3 respectively)
46   #                 Note that the posterior p1 and p2 vectors are proportions conditional
```

[20] Reference

```
#                 on non-error partitions
# Element two ... A three column matrix of posterior p values, column 1 corresponds to S1,
#                 col 2 to S3, and col 3 to Se

library(LaplacesDemon)     # for Dirichlet functions
library(commonFunctions)   # for frequency mode function

# Compute M, the total number of partitions
M <- sum(S)

# Construct matrix of all permutations of S1, S0, and Se that sum to M
permS <- do.call(rbind, apply(as.matrix(0:M), 1,
                              function(S1) t(apply(as.matrix(0:(M-S1)), 1,
                                                   function(S0) c(S1, S0, M-S1-S0)))))
colnames(permS) <- c("S1", "S0", "Se")

# Order permutations in S1=M-0, M-1, M-2, ... sequence
# Append column for computed multinomial coefficient value
permS <- cbind(permS[order(permS[,"S1"], permS[,"S0"], permS[,"Se"],
                           decreasing=c(T, F, F)),],
               "mc"=0)

# Compute multinomial coefficient values for S1, S0, Se
# Note that the table structure is
#
# i=row   k   S1=M-k   S0   Se   mc
#   1     0   M         0    0   M!/(M!0!0!) = 1
#   2     1   M-1       0    1   M!/[(M-1)!0!1!] = M
#   3     1   M-1       1    0   M!/[(M-1)!1!0!] = M
#   4     2   M-2       0    2   M!/[(M-2)!0!2!] = mc[2]*(M-1)/2 = mc[i-k]*(M-k+1)/k
#   5     2   M-2       1    1   M!/[(M-2)!1!1!] = mc[3]*(M-1)/1 = mc[i-k]*(M-k+1)/(k-1)
#   6     2   M-2       2    0   M!/[(M-2)!2!0!] = mc[i-k]
#   7     3   M-3       0    3   M!/[(M-3)!0!3!] = mc[4]*(M-2)/3 = mc[i-k]*(M-k+1)/k
#   8     3   M-3       1    2   M!/[(M-3)!1!2!] = mc[5]*(M-2)/2 = mc[i-k]*(M-k+1)/(k-1)
#   9     3   M-3       2    1   M!/[(M-3)!2!1!] = mc[6]*(M-2)/1 = mc[i-k]*(M-k+1)/(k-2)
#  10     3   M-3       3    0   M!/[(M-3)!3!0!] = mc[i-k]
#   .     .     .       .    .
#   .     .     .       .    .
#   .     .     .       .    .
#
# where all permutations of S1, S0, Se s.t. S1+S0+Se = M and
# mc = combinations(M, S1)*combinations(M-S1, S0)
#    = M!/[S1!*(M-S1)!] * (M-S1)!/[S0!*(M-S1-S0)!]
#    = M!/[S1!*S0!*(M-S1-S0)!] = M!/(S1!*S0!*Se!)

# Assign trivial multinomial coefficient (mc) values for rows 1, 2, and 3
permS[1:3,"mc"]=c(1, M, M)

# Compute subsequent rows from prior computed rows
for(k in 2:M) {
  # Compute first row for set k (note that each set k has k+1 rows)
  i0 <- (k-1)*k/2+k+1
  for(i in i0:(i0+k-1)) {
    permS[i,"mc"] <- permS[i-k,"mc"]*(M-k+1)/(k-i+i0)
  }
  permS[i0+k,"mc"] <- permS[i0,"mc"]
}

# Generate Laplace perturbed S1, S0, and Se
nS <- S + rlaplace(3,0,2/epsilon)

# Compute static Laplace deviations from S1, S0, and Se
dslap <- dlaplace(nS[1], permS[,1], 2/epsilon, log = T) +
         dlaplace(nS[2], permS[,2], 2/epsilon, log = T) +
         dlaplace(nS[3], permS[,3], 2/epsilon, log = T)

# Generate initial p1, p0, pe triplet
p <- rdirichlet(1, c(1, 1, 1))

# Compute multinomial probabilities using permutations table, Laplace offsets, and
# iterated Dirichlet probabilities, on posterior p value per iteration
# Exclude first 1,000 computed values (standard Gibbs sampling technique)
pDP <- t(apply(as.matrix(1:5000), 1,
           function(i) {
             # Compute weights for each permutation of S1, S0, and Se
             prob <- log(pMultinomPermS(permS, p)) + dslap
             prob <- exp(prob-max(prob))
             # Sample one S1, S0, Se triplet using computed weights
             S <- permS[sample(1:nrow(permS), 1, prob=prob),1:3]
             # Compute a posterior p
             p <<- rdirichlet(1, alpha+S)
```

```
            }))[1001:5000,]
  colnames(pDP) <- c("p1", "p0", "pe")

  # Compute mode of posterior p1, p0, and pe values
  # Note that p1 and p0 probabilities are given as proportions of non-Se partitions
  # Return pDP matrix and modes
  list("Mode"=c("r-hat"=frequencyMode(pDP[,1]/(pDP[,1]+pDP[,2]), 250, 5),
                "p0-hat"=frequencyMode(pDP[,2]/(pDP[,1]+pDP[,2]), 250, 5),
                "e-hat"=frequencyMode(pDP[,3], 250, 5)),
       "pDP"=pDP)

}
```

6.5 C Function to Compute Multinomial Probabilities

Following is the `pMultinomPermS()` C function that computes multinomial probabilities using the multinomial permutation and coefficient table provided in the `permS` parameter and values of p_1, p_0, and p_{err} supplied in the `p` parameter. The product $p_1^{S_1} p_0^{S_0} p_{err}^{S_{err}}$ is computed more efficiently as $e^{S_1 \log(p_1)+S_0 \log(p_0)+S_{err} \log(p_{err})}$.

Explain function parameters and return values.

```
// Compute multinomial probabilities using supplied S1, S0, Se permutations and
// corresponding p1, p0, pe probabilities

// Parameters:
// permS .... Matrix of S1, S0, Se, and multinomial coeff values, one per row
//             col 0 ... S1
//             col 1 ... S0
//             col 2 ... Se
//             col 3 ... mc = standard multinomial coefficient from product of
//                          combinations involving M, S1, S0, and Se, where M=S1+S0+Se
// p ........ Vector of S1, S0, and Se probabilities, p1 in pos 0, p0 in pos1, and
//             pe in pos 3 - note the assumption that p1+p0+pe=1

// Result is a numeric vector where element i contains the computed multinomial
// probability corresponding to row i of permS

// Note that the multinomial probability corresponding to S1, S0, Se, p1, p0, and pe is
// mc * p1**S1 * p2**S2 * pe, where mc = combinations(M, S1)*combinations(M-S1, S2) =
// M!/[S1!*(M-S1)!] * (M-S1)!/[S2!*(M-S1-S2)!] =
// M!/[S1!*S2!*(M-S1-S2)!] = M!/(S1!*S2!*Se!)

// [[Rcpp::export]]
NumericVector pMultinomPermS(NumericMatrix permS, NumericVector p) {
  int i, n=permS.nrow();
  double lnp[p.size()];
  NumericVector prob(n);
  // Convert p to ln(p) and use to compute p1**S1 * p0**S0 * pe**Se
  // exp(Si*ln(pi)) is generally more efficient than pi**Si
  for(i=0; i<p.size(); i++)
    lnp[i]=log(p(i));
  for(i=0; i<n; i++)
    prob(i)=permS(i,3)*exp(permS(i,0)*lnp[0]+permS(i,1)*lnp[1]+permS(i,2)*lnp[2]);
  return(prob);
}
```

7 Performance Evaluation

Finally, we compare the computational efficiency of the improved algorithm as implemented in section 6.4 to that of the original algorithm of section 5.2. Figure 3 plots, for total partition counts $M \in (10, 20, 30, 40, 50)$, fifty random sets of simulated S_1, S_0, S_{err} partition counts ($S_1 + S_0 + S_{err} = M$), and three values of ϵ (0.5, 1.0, and 1.5), the mean execution time in seconds of five iterations of computing p_1, p_2, p_{err} posterior distributions and modes using both algorithms. Increase of computation time with respect to M appears exponential for the original algorithm, while remaining linear for the improved algorithm (identified as `permS`).[21] In real terms, with a typical problem size of $M = 50$, the approximate fifty times performance

[21] Due to the scale of compute time and the separation of corresponding times between algorithms, `permS` exhibits slight, barely perceptible, increase of time with respect to M, making it comparatively linear.

advantage of the improved algorithm over using standard R functions reduces overall analysis time from many minutes (typically one posterior distribution is produced per parameter estimate being verified) to seconds. *More on this in the conclusion.* Choice of ϵ, in the range presented here, does not appear to affect the efficiency of either algorithm.

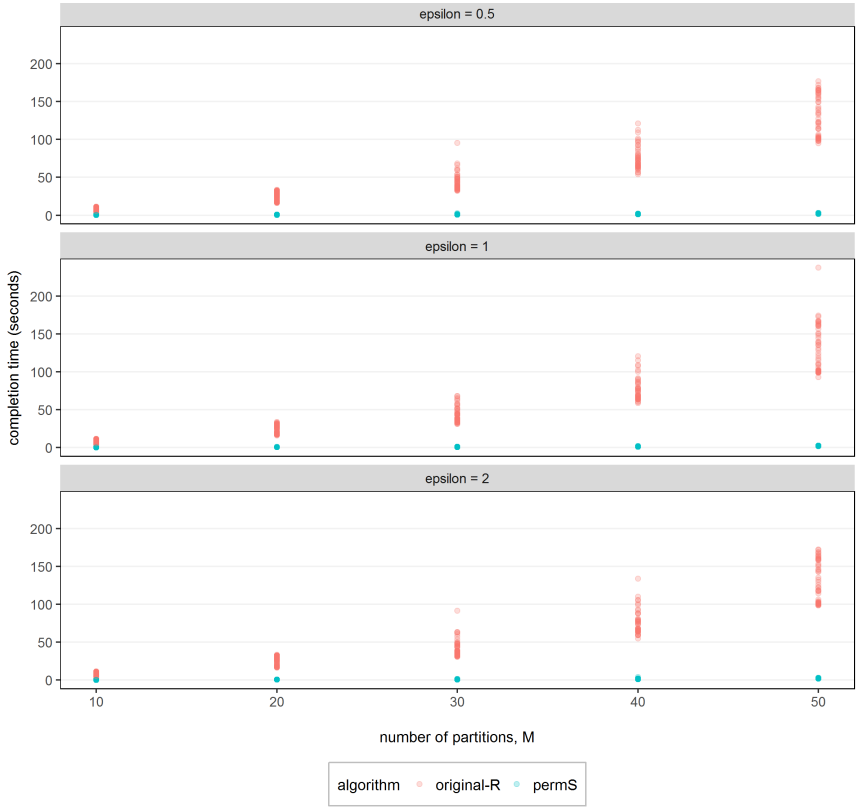


Figure 3: Execution times of original R and improved (`permS`) algorithms. $M = 10$ to 50. Fifty random S_1, S_0, S_{err} permutations.