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

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

April 22, 2018

Abstract

Privacy protecting synthetic data make available to the public a version of private data that retain key covariate relationships while limiting risk of disclosure of sensitive information regarding individuals or subjects that participate in the private data. Role of verification servers. Role of ϵ-Differential Privacy (ϵ-DP)[1] in verification server results. Use of simulation to research distributions of verification server results and sensitivity to choices of data subsets, ϵ, and partitions. ϵ-DP probability function analysis, isolation of inefficient components, and methods of improving performance.

1 Introduction

2 ϵ-Differential Privacy

Introduce concepts of differential privacy with references and examples

3 ϵ-DP Multinomial Algorithm

- Explain:
  - Partitioning of units for privacy masking (number of partitions = M)
  - Regression threshold parameter verification measure
  - Identification of beyond-threshold partitions (S_1), within-threshold (S_0), and non-computable partitions (S_{err}) partitions
- Present multinomial, Dirichlet, Laplace posterior probability distribution function, $f_p(S_1, S_0, S_{err}, M, \epsilon)$
- Discuss properties of $f_p(S_1, S_0, S_{err}, M, \epsilon)$
- Include and evaluate surface plots of grid evaluated Markov sample space

*Tom Balmat is Statistician, Social Science Research Institute, Duke University, Durham, NC 27708 (thomas.balmat@duke.edu); Andrés F. Barrientos is Postdoctoral Associate, Department of Statistical Science, Duke University, Durham, NC 27708 (email: afb26@stat.duke.edu); Jerome Reiter is Professor, Department of Statistical Science, Duke University, Durham, NC 27708 (jerry@stat.duke.edu). This research was supported by NSF grants ACI-14-43014 and SES-11-31897, as well as the Alfred P. Sloan Foundation grant G-2-15-20166003.

[1] ϵ-DP will be used to denote ϵ-Differential Privacy or an ϵ-Differentially Private algorithm.

4 Native R Implementation of Posterior Multinomial Algorithm

Discuss R implementation of iterated evaluation of posterior distribution

R script for DP.threshold function:

```
library(LaplacesDemon)

# Diferential private 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 -- p: multinomial parameter
  p = rdirichlet(1,c(1,1,1))

  # 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
    p <<- rdirichlet(1,alpha+s)
    c(s,p)
  }

  # MCMC evaluation of posterior distribution
  pDP <- t(sapply(1:5000, f.iter)[,1001:5000])

  # Return p1, p0, and pe
  # Note that p1 and pe are computed as conditional on not(pe)
  c("p1"=frequencyMode(pDP[,4]/(pDP[,4]+pDP[,5]), 250, 5),
    "p0"=frequencyMode(pDP[,5]/(pDP[,4]+pDP[,5]), 250, 5),
    "pe"=frequencyMode(pDP[,6], 250, 5))

}
```

5 Efficiency Diagnosis

Primary inefficiency lies in 5,000 successive evaluations of `dmultinom()`, since multinomial combination coefficients for all permutations of S_1, S_0, and S_{err} must be recomputed for each iteration. For a common value of M=50, there are 1,326 permutations, giving 5,000 $\times$ 1,326 = 6,630,000 combination coefficients, each involving factorials on the order of M!

6 Alternative Implementation

6.1 Review of Multinomial Probability Mass Function

6.2 Opportunity: Single Instance Construction of Multinomial Coefficient Table

- Computational aspects of combination products
- Identify pattern of combinations for efficient construction of table
- Measure and present performance of R statements to evaluate $f_p(S_1, S_0, S_{err}, M, \epsilon)$ - improvement over `dmultinom()`, but remains somewhat inefficient

6.3 Improved Implementation

Strategy:

- Construct multinomial coefficient table in R
- Pass table to C function for evaluation of $f_p(S_1, S_0, S_{err}, M, \epsilon)$
- Use addition of logarithms for exponentiation

R script for DP.threshold.multinomial function to construct multinomial coefficient table (lines 57 through 101) and iteratively call `pMultinomPermS()` C function to compute multinomial probabilities (line 120):

```
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, e
6  # 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 is
12 # 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, independen calls to dmultinomial() which, presumably, must recompute coefficients
15 # 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 (and, in trials, 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:
44  # A two element list:
45  # Element one ... A two element vector:
46  #                 Position one ... The mode of posterior perturbed S1 partitions conditional
47  #                                  on non-error partitions
48  #                 Position two ... A three column matrix of perturbed posterior p values
49  #                                  Column 1 contains S1 values, col 2 contains S0 values,
50  #                                  and col 3 contains Se values
51
52  library(LaplacesDemon)
53
54  # Compute M, the total number of partitions
55  M <- sum(S)
56
57  # Construct matrix of all permutations of S1, S0, and Se that sum to M
58  permS <- do.call(rbind,
59  apply(as.matrix(0:M), 1,
60  function(S1) t(apply(as.matrix(0:(M-S1)), 1,
61  function(S0) c(S1, S0, M-S1-S0)))))
62  colnames(permS) <- c("S1", "S0", "Se")
63
64  # Order permutations in S1=M-0, M-1, ... sequence
65  # Append column for computed multinomial coefficient value
66  permS <- cbind(permS[order(permS[,"S1"], permS[,"S0"], permS[,"Se"], decreasing=c(T, F, F)),], "mc"=0)
67
68  # Compute multinomial coefficient values for S1, S0, Se
69  # Note that the table structure is
70  #
71  # i=row  k  S1=M-k  S0  Se  mc
72  #   1    0  M       0   0  M!/(M!0!0!) = 1
73  #   2    1  M-1     0   1  M!/[(M-1)!0!1!] = M/(0!1!)
74  #   3    1  M-1     1   0  M!/[(M-1)!1!0!] = M/(1!0!)
75  #   4    2  M-2     0   2  M!/[(M-2)!0!2!] = M(M-1)/(0!2!) = mc[2]*(M-1)/2 = mc[i-k]*(M-k+1)/k
76  #   5    2  M-2     1   1                    M(M-1)/(1!1!) = mc[3]*(M-1)/1 = mc[i-k]*(M-k+1)/(k-1)
77  #   6    2  M-2     2   0                    M(M-1)/(2!0!) = mc[4]         = mc[i-k]
78  #   7    3  M-3     0   3  M!/[(M-3)!0!3!] = M(M-1)(M-2)/(0!3!) = mc[4]*(M-2)/3 = mc[i-k]*(M-k+1)/k
79  #   8    3  M-3     1   2                    M(M-1)(M-2)/(1!2!) = mc[5]*(M-2)/2 = mc[i-k]*(M-k+1)/(k-1)
80  #   9    3  M-3     2   1                    M(M-1)(M-2)/(2!1!) = mc[6]*(M-2)/1 = mc[i-k]*(M-k+1)/(k-2)
81  #  10    3  M-3     3   0                    M(M-1)(M-2)/(3!0!) = mc[7]         = mc[i-k]
82  #   .       .       .   .
83  #   .       .       .   .
84  #   .       .       .   .
85  #
86  # where all permutations of S1, S0, Se s.t. S1+S0+Se = M and
87  # mc = combinations(M, S1)*combinations(M-S1, S0) = M!/[S1!*(M-S1)!] * (M-S1)!/[S0!*(M-S1-S0)!] =
88  # M!/[S1!*S0!*(M-S1-S0)!] = M!/(S1!*S0!*Se!)
89
90  # Assign trivial multinomial coefficient (mc) values for rows 1, 2, and 3
91  permS[1:3,"mc"]=c(1, M, M)
92
93  # Compute subsequent rows from prior computed rows
94  for(k in 2:M) {
95      # Compute first row for set k (note that each set k has k+1 rows)
96      i0 <- (k-1)*k/2+k+1
97      for(i in i0:(i0+k-1)) {
98          permS[i,"mc"] <- permS[i-k,"mc"]*(M-k+1)/(k-i+i0)
99      }
100     permS[i0+k,"mc"] <- permS[i0,"mc"]
101 }
102
103 # Generate noisy S1, S0, and Se
104 nS <- S + rlaplace(3,0,2/epsilon)
105
106 # Compute static Laplace deviations from S1, S0, and Se
107 dslap <- dlaplace(nS[1], permS[,1], 2/epsilon, log = T)+
108 dlaplace(nS[2], permS[,2], 2/epsilon, log = T)+
109 dlaplace(nS[3], permS[,3], 2/epsilon, log = T)
110
111 # Generate initial p1, p0, pe triplet
112 p <- rdirichlet(1, c(1, 1, 1))
113
114 # Compute multinomial probabilities using permutations table, Laplace offsets, and
115 # iterated Dirichlet probabilities, on posterior p value per iteration
116 # Exclude first 1,000 computed values (standard Gibbs sampling technique)
117 pDP <- t(apply(as.matrix(1:5000), 1,
118 function(i) {
119     # 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)

}
```

`pMultinomPermS()` C function to compute multinomial probabilities:

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

// Parameters:
// permS .... Matrix of S1, S0, Se, and multinomial coefficient 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
// Addition of log values is more efficeient than corresponding exponentiation operations
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);
}
```

6.4 Performance Evaluation

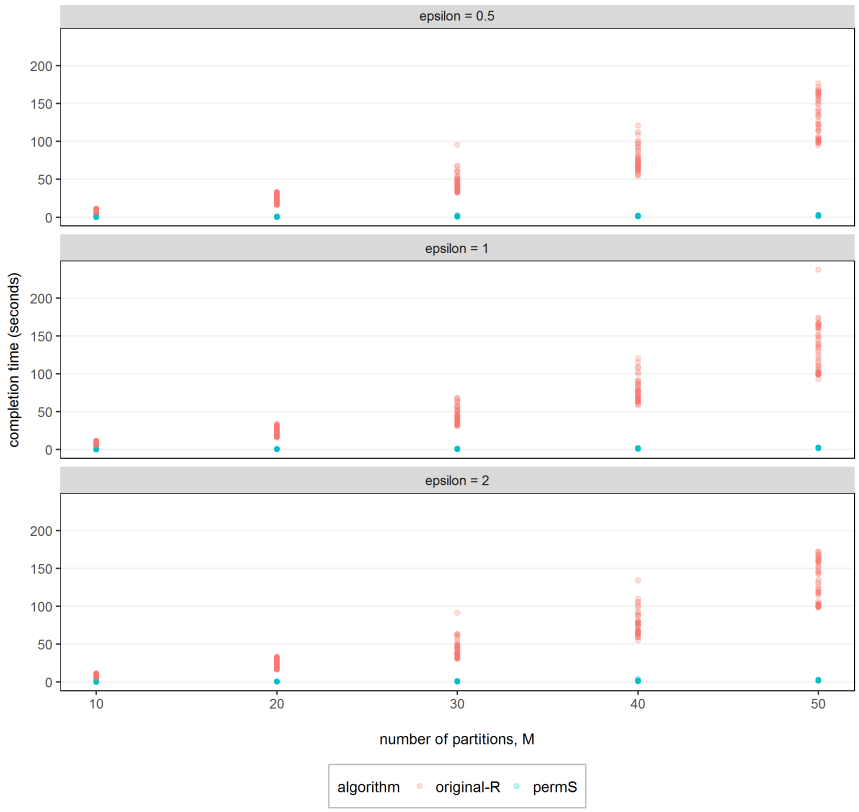

Figure 1: Execution times of original R and permS (single instance table with C function) algorithms

7 Conclusion

References