PHP 2550: Project 3

Peirong Hao

Abstract

(300 words or less) that summarizes the major results and conclusions to a non-technical reader

Simulation Design Using ADEMP Framework

Aims

We aim to enhance the design of a cluster-randomized trial by accounting for budget constraints and varying parameters. This project explores two data-generating scenarios: Normally distributed outcomes and Poisson-distributed outcomes. The simulation focuses on identifying the optimal combination of the number of clusters (G) and the number of observations per cluster (R) that minimizes estimation variability within a fixed budget (B). Moreover, the study investigates the impact of varying the relative costs of sampling the first observation from a cluster (c_1) compared to additional measurements within the same cluster (c_2) under the condition that $c_1 > c_2$. The analysis further explores how adjustments to model parameters—such as the intercept (α), treatment effect (β), and cluster-level variance (γ^2)—influence the treatment effect estimates. Finally, it examines the effect of varying the budget (B) on the optimal design.

Data-Generating Mechanisms

We generate the data using hierarchical models. Observations j range from 1 to R, representing measurements within each cluster, while clusters i range from 1 to G (the total number of clusters). In the Normal scenario, treatment assignment (X_i) is determined by a Bernoulli distribution with a fixed probability of 0.5, ensuring that approximately half of the clusters are treated while the remainder serve as controls. Observations within the same cluster share a cluster-level mean, modeled as $\mu_i = \alpha + \beta X_i + \epsilon_i$, where ϵ_i represents the cluster-level deviation and follows a distribution $N(0, \gamma^2)$. Individual-level outcomes are generated as $Y_{ij} = \mu_i + e_{ij}$, where e_{ij} reflects within-cluster variation and is drawn from $N(0, \sigma^2)$. For the Poisson scenario, the cluster-level mean is modeled on the log scale as $\log(\mu_i) = \alpha + \beta X_i + \epsilon_i$, where $\epsilon_i \sim N(0, \gamma^2)$ captures the cluster-level variability. Individual-level outcomes are generated from a Poisson distribution, $Y_{ij} \sim \text{Poisson}(\mu_i)$. Unlike the Normal scenario, there is no σ parameter in this case because the Poisson distribution models variability through its mean (variance equal to mean).

Estimands

The primary estimand is the treatment effect (β), contributing to the outcome difference between the treatment and control groups. In the Normal scenario, β reflects the difference in cluster-level means, while in the Poisson scenario, β represents the log-scale difference in cluster-level means between the two groups. Importantly, β is an unbiased estimate. The precision of β is evaluated using the standard deviation of the estimates and the average standard error of the estimates.

Methods

The simulation generates hierarchical data using the `DataSim()` function, allowing for variation in parameters such as the number of clusters (G), the number of observations per cluster (R), and the cluster-level variance (γ^2). For each set of parameters, two datasets—one based on Normal models and the other on Poisson models—are generated per iteration, with multiple iterations conducted in total to ensure robust evaluation. To optimize the recruitment design, the number of observations per cluster (R) is determined using the `BudgetOpt()` function, which incorporates the fixed budget (B), the number of clusters (G), and the relative costs of sampling from new clusters (c_1) versus collecting additional measurements within existing clusters (c_2). Once the data are generated, models are fit using `lmer()` for the Normal scenario and `glmer()` for the Poisson scenario to estimate the treatment effect (β).

Parameter variation, implemented through the `ParamVary()` function, involves systematically adjusting one variable at a time—such as β, γ, c_1, and B—while keeping all other variables constant. Performance metrics, including the standard deviation of the estimated β, the average of standard errors, and the average number of clusters and observations per cluster, are summarized using the MeasureGen() function. All simulated datasets and their corresponding performance metrics are stored as CSV files in specified folders for further evaluation and replication. This approach provides detailed insights into the trade-offs and optimal parameter configurations for cluster-randomized trials.

Performance Measures

	C1	sd.est.beta	avg.se	avg.G	avg.R
[1,]	10	0.2741042	0.2902551	50	19
[2,]	15	0.3139891	0.2891685	50	12
[3,]	20	0.2702513	0.2897551	50	9
[4,]	25	0.2768017	0.2900370	50	7
[5,]	30	0.2982989	0.2919546	50	5

	C1	sd.est.beta	avg.se	avg.G	avg.R
[1,]	10	0.3272949	0.2799528	50	19
[2,]	15	0.2605945	0.2785228	50	12
[3,]	20	0.2655971	0.2772430	50	9
[4,]	25	0.2547592	0.2747246	50	7
[5,]	30	0.2924640	0.2804036	50	5

	relative.cost	sd.est.beta	avg.se	avg.G	avg.R
[1,]	2	1.411570	1.332744	10	19
[2,]	4	1.184162	1.324833	10	37
[3,]	5	1.486937	1.295937	10	46
[4,]	10	1.445985	1.394718	10	91
[5,]	20	1.456687	1.267056	10	181

	relative.cost	sd.est.beta	avg.se	avg.G	avg.R
[1,]	2	1.317709	1.129742	10	19
[2,]	4	1.337386	1.204039	10	37
[3,]	5	1.149710	1.123400	10	46
[4,]	10	1.230449	1.159242	10	91
[5,]	20	1.234737	1.129052	10	181

	G	sd.est.beta	avg.se	avg.G	avg.R
[1,]	10	1.397802	1.288664	10	19
[2,]	11	1.181056	1.295879	11	17
[3,]	12	1.082727	1.176143	12	15
[4,]	13	1.160067	1.163120	13	14
[5,]	14	1.155631	1.082111	14	13
[6,]	15	1.229231	1.077135	15	12

	G	sd.est.beta	avg.se	avg.G	avg.R
[1,]	10	1.283899	1.169125	10	19
[2,]	11	1.361136	1.054629	11	17

[3,]	12	1.171868	1.077104	12	15
[4,]	13	1.198771	1.076393	13	14
[5,]	14	1.058283	1.059651	14	13
[6,]	15	1.092523	1.007337	15	12

	alpha	sd.est.beta	avg.se	avg.G	avg.R
[1,]	1	0.2641237	0.2829286	50	9
[2,]	2	0.2706459	0.2870447	50	9
[3,]	3	0.2929889	0.2833941	50	9
[4,]	4	0.3039538	0.2876315	50	9
[5,]	5	0.2695995	0.2887808	50	9

	alpha	sd.est.beta	avg.se	avg.G	avg.R
[1,]	1	0.2815969	0.2809794	50	9
[2,]	2	0.2598077	0.2829152	50	9
[3,]	3	0.3088146	0.2746652	50	9
[4,]	4	0.3061329	0.2849467	50	9
[5,]	5	0.2995716	0.2795033	50	9

	beta	sd.est.beta	avg.se	avg.G	avg.R
[1,]	1	0.3317842	0.2907783	50	9
[2,]	2	0.3009058	0.2875792	50	9
[3,]	3	0.2740925	0.2894229	50	9
[4,]	4	0.3140218	0.2903440	50	9
[5,]	5	0.3156115	0.2854598	50	9

	beta	sd.est.beta	avg.se	avg.G	avg.R
[1,]	1	0.2872367	0.2774593	50	9
[2,]	2	0.3237265	0.2770358	50	9
[3,]	3	0.2920490	0.2808062	50	9
[4,]	4	0.2745185	0.2735426	50	9
[5,]	5	0.2673706	0.2792871	50	9

	gamma	sd.est.beta	avg.se	avg.G	avg.R
[1,]	1.0	0.2937777	0.2881550	50	9
[2,]	1.5	0.4413557	0.4251899	50	9
[3,]	2.0	0.5682497	0.5719050	50	9
[4,]	2.5	0.8382521	0.7111686	50	9
[5,]	3.0	0.7470062	0.8542530	50	9

```
     gamma sd.est.beta    avg.se avg.G avg.R
[1,]   1.0   0.2740868 0.2811024    50     9
[2,]   1.5   0.4187040 0.4206944    50     9
[3,]   2.0   0.5068276 0.5615657    50     9
[4,]   2.5   0.6742237 0.7018637    50     9
[5,]   3.0   0.8216183 0.8319821    50     9
```

```
        B sd.est.beta    avg.se avg.G avg.R
[1,] 3000   0.2869276 0.2949174    50     5
[2,] 3500   0.3092981 0.2890498    50     6
[3,] 4000   0.2703792 0.2846196    50     7
[4,] 4500   0.2922466 0.2898946    50     8
[5,] 5000   0.2744977 0.2876837    50     9
```

```
        B sd.est.beta    avg.se avg.G avg.R
[1,] 3000   0.2717654 0.2802983    50     5
[2,] 3500   0.2920992 0.2766314    50     6
[3,] 4000   0.2597260 0.2771127    50     7
[4,] 4500   0.2867003 0.2780047    50     8
[5,] 5000   0.2950794 0.2835006    50     9
```

Analysis and Results

Limitations of your methods and of the data

Code Appendix

```
set.seed(123456)
library(purrr)
library(lme4)
library(lmerTest)

library(brms)
library(blme)

library(Rlab)#!!! might not need
#data generation
#G clusters, each cluster has R members
#clusters are independent
#members in each cluster share same cluster mean (correlated)
DataSim <- function(G, R, alpha, beta, gamma, method, sigma=0.5,
 ↪  p.trt=0.5){
  data <-data.frame(matrix(ncol = 7, nrow = G*R))
  colnames(data) <- c('G', 'R', 'X', 'Y', 'alpha', 'beta', 'gamma')
  # data[,"G"] <- rep(1:G, each=R)
  # data[,"R"] <- rep(1:R, G)

  # Generate X: 0 for ctrl, 1 for trt
  x <- rbern(n=G, prob=p.trt)
  mu.0 <- alpha + beta * x

  epsilon <- rnorm(G, mean=0, sd=gamma)
  mu <- mu.0 + epsilon
  if(method == "poisson"){
      mu <- exp(mu)
    }
  #mu <- ifelse(method=="poisson", exp(mu.0 + epsilon), mu.0 + epsilon)

  for(i in 1:G){
    if(method == "poisson"){
      y <- rpois(R, lambda = mu[i])
    }
    else{
      eps <- rnorm(n=R, mean=0, sd=sigma)
      y <- mu[i] + eps
    }
```

```r
    for(j in 1:R){
      row.num <- i*R-(R-j)
      data[row.num, ] <- c(i, j, x[i], y[j],
                           alpha, beta, gamma)
    }
  }
  return(data)
}

#alpha refers to intercept
#method = "poisson" or "normal"
DataGen <- function(folder.data, filename, alpha, beta, gamma, B, C1,
 ↪  relative.cost, G, method){
  R <- BudgetOpt(B, C1, relative.cost, G)

  # Simulate data
  data <- DataSim(G, R, alpha, beta, gamma, method)

  # Create data folder
  if (!dir.exists(folder.data)) {  # Check if the folder already exists
    dir.create(folder.data)
  }

  # Create csv file
  write.csv(data, paste0(folder.data, filename, "_data.csv"),
   ↪  row.names=FALSE)
}

# ##########################################test data generation
# #method="normal"
# method="poisson"
# for(i in 1:2){
#   DataGen(folder.data =
 ↪  "~/Documents/GitHub/PHP2550-PDA-project3/Data/",
#           filename = paste0("sim",'_',i,'_',method), method = method)
# }
#
 ↪  #test1<-read.csv("~/Documents/GitHub/PHP2550-PDA-project3/Data/sim_1_data.csv")
#
# #method="normal"
```

```
# method="poisson"
# folder.data = "~/Documents/GitHub/PHP2550-PDA-project3/Data/"
# filename.data = paste0("sim",'_',i,'_',method)
# data.path = paste0(folder.data, filename.data, "_data.csv")
# #ModelFit(data.path, method=method)
# ###########################################test data generation

#fit hierarchical model
#method = "poisson" or "normal"
ModelFit <- function(data.path, method, true.beta=0.5){

  data <- read.csv(data.path)

  if(method == "poisson"){
      mdl <- glmer(Y ~ X + (1 | G), data=data, family="poisson")
    }
  else{
      mdl <- lmer(Y ~ X + (1 | G), data=data, control = lmerControl(optimizer = "Nelder_Mead"))
    }

  #do not need to worry about p-value < or > alpha=0.05
  summ <- summary(mdl)
  est.beta <- summ$coefficients["X", "Estimate"]
  se <- summ$coefficients["X", "Std. Error"]
  #??? include coverage
  measure <- c(est.beta, se, max(data$G), max(data$R))
  return(measure)
}

#???set constraint: G>=2, R>=2
#make sure C1 > C2
BudgetOpt <- function(B, C1, relative.cost, G){
  C2 <- C1/relative.cost
  #constraint: C1*G + C2*G*(R-1) <= B
  R <- floor(1 + (B-C1*G)/(C2*G))
  return(R)
}

#param: C1, relative.cost, G, beta, gamma, (change alpha only for poisson case)
```

```
#do not change p.trt, sigma
#could change B as well
ParamVary <- function(param.ls, param.name, method, max.iter = 100,
                      folder.data =
                       ↪  "~/Documents/GitHub/PHP2550-PDA-project3/Data/",
                      folder.perf =
                       ↪  "~/Documents/GitHub/PHP2550-PDA-project3/Perf/",
                      C1=20, relative.cost=2, G=10, alpha=5, beta=3,
                       ↪  gamma=2, B=2000){

  for(i in 1:length(param.ls)){
      param = param.ls[i]

      for(j in 1:max.iter){
        if(j==1){
          perf.measure <- as.data.frame(matrix(NA, nrow = max.iter, ncol
 ↪   = 4))
          colnames(perf.measure) <- c("est.beta", "se", "G", "R")
        }

        filename.data =
 ↪   paste0("sim",'_',i,'_',j,'_',param.name,'_',method)
        data.path = paste0(folder.data, filename.data, "_data.csv")

        if(param.name=="C1"){
          DataGen(folder.data, filename.data, C1=param, method=method,
                  relative.cost=relative.cost, G=G, alpha=alpha,
                   ↪  beta=beta, gamma=gamma, B=B)
          perf.measure[j,] <- ModelFit(data.path, method=method)
        }
        else if(param.name=="relative.cost"){
          DataGen(folder.data, filename.data, relative.cost=param,
           ↪  method=method,
                  C1=C1, G=G, alpha=alpha, beta=beta, gamma=gamma, B=B)
          perf.measure[j,] <- ModelFit(data.path, method=method)
        }
        else if(param.name=="G"){
          DataGen(folder.data, filename.data, G=param, method=method,
                  relative.cost=relative.cost, C1=C1, alpha=alpha,
                   ↪  beta=beta, gamma=gamma, B=B)
          perf.measure[j,] <- ModelFit(data.path, method=method)
```

```
      }
      else if(param.name=="alpha"){
        DataGen(folder.data, filename.data, alpha=param,
         ↪  method=method,
                relative.cost=relative.cost, G=G, C1=C1, beta=beta,
                 ↪  gamma=gamma, B=B)
        perf.measure[j,] <- ModelFit(data.path, method=method)
      }
      else if(param.name=="beta"){
        DataGen(folder.data, filename.data, beta=param, method=method,
                relative.cost=relative.cost, G=G, alpha=alpha, C1=C1,
                 ↪  gamma=gamma, B=B)
        perf.measure[j,] <- ModelFit(data.path, true.beta=param,
↪  method=method)
      }
      else if(param.name=="gamma"){
        DataGen(folder.data, filename.data, gamma=param,
         ↪  method=method,
                relative.cost=relative.cost, G=G, alpha=alpha,
                 ↪  beta=beta, C1=C1, B=B)
        perf.measure[j,] <- ModelFit(data.path, method=method)
      }
      else if(param.name=="B"){
        DataGen(folder.data, filename.data, B=param, method=method,
                relative.cost=relative.cost, G=G, alpha=alpha,
                 ↪  beta=beta, gamma=gamma, C1=C1)
        perf.measure[j,] <- ModelFit(data.path, method=method)
      }

      if(j==max.iter){
        # Create performance folder
        if (!dir.exists(folder.perf)) {  # Check if the folder already
         ↪  exists
          dir.create(folder.perf)
        }

        # Create csv file
        filename.perf = paste0("sim",'_',i,'_',param.name,'_',method)
        write.csv(perf.measure, paste0(folder.perf, filename.perf,
         ↪  "_perf.csv"), row.names=FALSE)
      }
```

```r
      }
    }
  }

MeasureGen <- function(param.ls, param.name, method,
                       folder.perf =
                        ↪  "~/Documents/GitHub/PHP2550-PDA-project3/Perf"){

  perf.measure.final <- matrix(NA, nrow = length(param.ls), ncol = 5)
  colnames(perf.measure.final) <- c(param.name, "sd.est.beta", "avg.se",
   ↪  "avg.G", "avg.R")
  perf.measure.final[,1] <- param.ls

  for(i in 1:length(param.ls)){
    file.i <- list.files(path = folder.perf,
                             pattern =
                              ↪  paste0("sim",'_',i,'_',param.name,'_',method),
                              ↪
                             full.names = T)
    file.df <- read.csv(file.i, stringsAsFactors = FALSE, header = TRUE)
    perf.measure.final[i,-1] <- c(sd(file.df[,1]),
 ↪  colMeans(file.df[,-1], na.rm = T))
  }
  #print(perf.measure.final)
  return(perf.measure.final)
  # return(knitr::kable(perf.measure.final,
  #            digits = 5, caption = paste0("Vary ", param, " & Method
   ↪  is ", method)))
}
#??? 3 common problems:
#boundary (singular) fit: see help('isSingular')

#Error: no more error handlers available (recursive errors?); invoking
 ↪  'abort' restart

#Warning: type 29 is unimplemented in 'type2char'Error in
 ↪  data.frame(rbind(c("algorithm", "character",
 ↪  paste("NLOPT_GN_DIRECT",  : INTEGER() can only be applied to a
 ↪  'integer', not a 'unknown type #29'
#Error in summ$coefficients["X", "Estimate"] : subscript out of bounds
```

```
#Error in DataSim(G, R, alpha, beta, gamma, method) : INTEGER() can only
↪  be applied to a 'integer', not a 'unknown type #29'

#Show in New Window, Error in summ$coefficients["X", "Estimate"] :
↪  subscript out of bounds

#Warning: Model failed to converge with max|grad| = 0.218378 (tol =
↪  0.002, component 1)fixed-effect model matrix is rank deficient so
↪  dropping 1 column / coefficient fixed-effect model matrix is rank
↪  deficient so dropping 1 column / coefficient Error in
↪  summ$coefficients["X", "Estimate"] : subscript out of bounds

#Error: no more error handlers available (recursive errors?); invoking
↪  'abort' restart Warning: type 29 is unimplemented in
↪  'type2char'Error in stopifnot(length(class2) == 1L) : INTEGER() can
↪  only be applied to a 'integer', not a 'unknown type #29'

method = "normal"
param.name = "C1"
#param.ls = c(6,10,12,16,20)
#param.ls = c(10,20,30,40,50)
param.ls = c(10,15,20,25,30)
ParamVary(param.ls, param.name, method,
          relative.cost=2, G=50, alpha=5, beta=2, gamma=1, B=5000)
MeasureGen(param.ls, param.name, method)

method ="poisson"
ParamVary(param.ls, param.name, method,
          relative.cost=2, G=50, alpha=5, beta=2, gamma=1, B=5000)
MeasureGen(param.ls, param.name, method)
#Error in summ$coefficients["X", "Estimate"] : subscript out of bounds
param.name = "relative.cost"
param.ls = c(2, 4, 5, 10, 20)

method = "normal"
ParamVary(param.ls, param.name, method)
MeasureGen(param.ls, param.name, method)

method ="poisson"
ParamVary(param.ls, param.name, method)
```

```
MeasureGen(param.ls, param.name, method)
#Error in summ$coefficients["X", "Estimate"] : subscript out of bounds
param.name = "G"
param.ls = c(10,11,12,13,14,15)

method = "normal"
ParamVary(param.ls, param.name, method)
MeasureGen(param.ls, param.name, method)

method ="poisson"
ParamVary(param.ls, param.name, method)
MeasureGen(param.ls, param.name, method)
#Error in summ$coefficients["X", "Estimate"] : subscript out of bounds
param.name = "alpha"
#param.ls = c(1,3,5,7,9)
param.ls = c(1,2,3,4,5)

method = "normal"#should not matter
ParamVary(param.ls, param.name, method,
          relative.cost=2, G=50, C1=20, beta=2, gamma=1, B=5000)
MeasureGen(param.ls, param.name, method)

method ="poisson"
ParamVary(param.ls, param.name, method,
          relative.cost=2, G=50, C1=20, beta=2, gamma=1, B=5000)
MeasureGen(param.ls, param.name, method)
#Error in summ$coefficients["X", "Estimate"] : subscript out of bounds

#Warning: convergence code 3 from bobyqa: bobyqa -- a trust region step
 ↪  failed to reduce q

#fixed-effect model matrix is rank deficient so dropping 1 column /
 ↪  coefficient
#Error in summ$coefficients["X", "Estimate"] : subscript out of bounds

param.name = "beta"
#param.ls = c(0.1, 0.5, 1, 3, 5)
param.ls = c(1,2,3,4,5)

method = "normal"
ParamVary(param.ls, param.name, method,
```

```
          relative.cost=2, G=50, C1=20, alpha=5, gamma=1, B=5000)
MeasureGen(param.ls, param.name, method)

method ="poisson"
ParamVary(param.ls, param.name, method,
          relative.cost=2, G=50, C1=20, alpha=5, gamma=1, B=5000)
MeasureGen(param.ls, param.name, method)
#Error in summ$coefficients["X", "Estimate"] : subscript out of bounds
param.name = "gamma"
#param.ls = c(0.5, 1, 2, 3, 4, 5)
param.ls = c(1, 1.5, 2, 2.5, 3)
method = "normal"
ParamVary(param.ls, param.name, method,
          relative.cost=2, G=50, C1=20, alpha=5, beta=3, B=5000)
MeasureGen(param.ls, param.name, method)

method ="poisson"
ParamVary(param.ls, param.name, method,
          relative.cost=2, G=50, C1=20, alpha=5, beta=3, B=5000)
MeasureGen(param.ls, param.name, method)
#fixed-effect model matrix is rank deficient so dropping 1 column /
 ↪  coefficient
#Error in summ$coefficients["X", "Estimate"] : subscript out of bounds
param.name = "B"
#param.ls = c(1000, 1500, 2000, 2500, 3000)
#param.ls = c(3000, 3500, 4000)
param.ls = c(3000, 3500, 4000, 4500, 5000)

method = "normal"
ParamVary(param.ls, param.name, method,
          relative.cost=2, G=50, C1=20, alpha=5, beta=3, gamma=1)
MeasureGen(param.ls, param.name, method)

method ="poisson"
ParamVary(param.ls, param.name, method,
          relative.cost=2, G=50, C1=20, alpha=5, beta=3, gamma=1)
MeasureGen(param.ls, param.name, method)
```