DataCreation

Dr. Dalzell

November 17, 2017

Preparing the data

This document contains the steps used to convert from the NCERDC EOG data files to the data set used for analysis. For access to the data, apply at https://childandfamilypolicy.duke.edu/research/nc-education-data-center/.

No data is contained in this folder.

Initial Steps

Each student in grades 3 through 8 is tested in reading and math each year. The North Carolina Education Research Data Center has a separate file for each of these grades, and each file has one record for each student who was a member of that school at the time of the test. Data include records for students who were absent or exempt from the test for various reasons. If a student takes a retest, this information is included in a separate student record within the file.

The data from the NCERDC (sas files) include the following information: the individual student's test records; the use of any testing modifications; the student's classroom activities, such as using calculators in math class or having written assignments in reading class; and the student's demographic data, including birth date, learning disabled status, and exceptionality status.

This data contains 2011 and 2012 EOG exam data from the NCERDC. The first step is to merge these two data sets to create create roughly 84000 possible records. This step was perfoemd in SAS Any records which did not have an id match were removed from consideration for this simulation.

```
#WholeThing = read.csv("N:/Working Space/Matched.csv")
```

The variables in the data set are as follows. We have recordings from both 2011 and 2012.

- birthdt - Student's Birth Date in 2012 To help protect the identities of the students, the NCERDC set the day of each birth date to the 15th without changing the month or the year. For example, anyone born in April 1998 would have a birth date of April 15, 1998. This date allows researchers to calculate and compare ages of students while further ensuring confidentiality of student records.
- birthmonth - The birth month as recorded
- birthyear - The birth year as recorded
- lname - student last name
- fname - student first name
- lea - Local Education Agency ** 010-995 = LEAs (Charter schools alphanumeric) 997 = Department of Human Resources, 998 = Office of Juvenile Justice
- schlcode - the school code
- sex - Student's Sex; M = Male, F = Female ethnic - Student's Ethnicity; A = Asian, B = Black, H = Hispanic, I = American Indian, M = Multi-Racial, P = Pacific-Islander, W = White
- readscal - This is the student's reading achievement test score. Note that this scale is not necessarily comparable across exams or years.
- mathscal - This is the student's math achievement test score. Note that this scale is not necessarily comparable across exams or years.

Exploring the data

Blocking Variables: Field Agreement

We know that we are going to be performing our linkage based on a few categorical variables which are common to both data sets. In the language of the Gutman paper, these common variables are denoted (Z). For our work, we denote the set of the common variables used for blocking as (B), and we refer to these variables as the blocking variables. Because these common variables will be used for linking, it is important to understand how reliable, i.e., error free, these variables are in the linked data.

```
# The total number of record pairs
#N = nrow(WholeThing)

# Determine which records agree on school, race, gender, birthdate.

# Birthday:
#BDmatch = which(apply(WholeThing, 1, function(x) sum(x[1] == x[12])) == 1)
# School:
#School_match = which(apply(WholeThing, 1, function(x) sum(x[7] == x[18])) == 1)
# Race:
#Race_match = which(apply(WholeThing, 1, function(x) sum(x[9] == x[20])) == 1)
# Sex:
#Sex_match = which(apply(WholeThing, 1, function(x) sum(x[8] == x[19])) == 1)

# Determine the number of records in each group
#SeedPool = intersect(intersect(BDmatch, School_match), intersect(Race_match, Sex_match))
#DisagreePool = setdiff(1:N, SeedPool)
```

Based on this initial exploration, less than 10% of the records in this data set will be placed in the correct block if we block directly based on these variables.

We already have the month and year for birthday split, but for matching it would also be nice to have day to allow for the creation of smaller blocks.

```
#BDay2011 = WholeThing[, "birthdt2011"]
#BDay2012 = WholeThing[, "birthdt"]

#AllDays11 = as.numeric(format(as.Date(BDay2011, "%m/%d/%Y"), format = "%d"))
#AllDays12 = as.numeric(format(as.Date(BDay2012, "%m/%d/%Y"), format = "%d"))

# Bind on the Day Columns
#WholeThing = cbind(WholeThing, birthday2011 = AllDays11, birthday2012 = AllDays12)

# How often do the birthdays match exactly? On almost all. We do not need
# to model an error on birthday.

#Day_Diff = which((WholeThing$birthday2011 == WholeThing$birthday2012) == 0)
```

We note that because less than 1% of the records do not match on birthday, we will not model birthday as being in error. This modelling decision means that we assert that day of birth is indepedent of test score.

The next step is determine the size of the deterministic clusters. In other words, if we block on the variables we currently have, assuming no error, what type of clusters are we dealing with?

In order to answer this question, we begin by moving the records into blocks.

```
#vars <- c("birthday", "birthmonth", "birthyear", "ethnicity", "sex", "school")

#B11 = WholeThing[, c("birthday2011", "birthmonth2011", "birthyear2011", "ethnic2011",    "sex2011", "s
#B12 = WholeThing[, c("birthday2012", "birthmonth", "birthyear", "ethnic", "sex",     "schlcode")]

# Check the levels for ethnicity
#table(B11[, "ethnic2011"])
#table(B12[, "ethnic"])


# Because the column names have to be the same in order for us to bind
# columms, we conver the names
#B12.holder = B12
#colnames(B12.holder) = colnames(B11)
#B = rbind(B11, B12.holder)
#colnames(B) = bvars

# Store the levels associated with each variable
#d = c()

# Levels for Birthday: 1
#d[1] = length(unique(B[, 1]))

# Levels for Birth Month : 2
#d[2] = length(unique(B[, 2]))

# Levels for BirthYear: 7
#d[3] = length(unique(B[, 3]))

# Levels for Ethnicity : 7
#d[4] = length(levels(B[, 4]))

# Levels for Sex: 2
#d[5] = length(levels(B[, 5]))

# Levels for School: 192
#d[6] = length(unique(B[, 6]))
```

Since we know that all of the records should be paired with another record, we can check to see how many of the records will be placed in the wrong block.

```
#inError = which(B11[, 1:6] != B12[, 1:6])
# We have 7809 records which will not end up in the correct block based on
# day
#length(which(B11[, 1:6] != B12[, 1:6]))
```

Now that we have the blocking variables, we can assign each record to a block based on the values of the blocking variables.

```
category.block <- function(x, p, d) {
    x = as.numeric(x)
    category = 0
    for (j in 1:(p - 1)) category = category + prod(d[(j + 1):p]) * (x[j] -
        1)
    category = category + x[p]
```

```
    return(category)
}
## This code is from Tancredi and Liseo (2011)(https://arxiv.org/pdf/1011.2649.pdf)

# Note that the below takes some time to run!

#blocks = NULL
#for (i in 1:(2 * N)) {
#    blocks[i] = category.block(B[i, ], length(d), d)
#}

#blocks1 = blocks[1:N]
#blocks2 = blocks[(N + 1):(2 * N)]
```

The code above takes each record and assigns it an index. For these 6 blocking variables, there are roughly 7,000,000 possible blocks that the records could be assigned to. The index tells us which of these possible blocks is assigned to each record based on the observed values of the blocking variables $\hat{B}$.

As a simplification, we will not allow for the creation of new blocks during this method. The only blocks that are allowed are blocks that are observed in the original data. This is necessary because of these vast number of possible blocks; this space is too large to effectively explore.

Because of this simplification, we convert the indices from the step above into "blocks". The smallest unique index that it is observed in $\hat{B}$ becomes the first block, and so on.

```
#Block_Ops = c(blocks1, blocks2)
#Index_to_Block = Legal_Index = sort(unique(Block_Ops))
#Block_Ops = as.numeric(as.factor(Block_Ops))
#indices1 <- blocks1
#indices2 <- blocks2
#blocks1 <- Block_Ops[1:N]
#blocks2 <- Block_Ops[(N + 1):(2 * N)]
```

Now we have roughly (K = 74,000) unique blocks that are filled. We explore this space rather than the roughly 7,000,000 possible blocks in the index space.

We recall that we are working with records from two different Files-2011 and 2012. For clarity, we assume that the records from 2011 are in the correct block, and only records from 2012 may be in error. Note to Self: We should probably run the simulations both ways to see what happens.

We now get an overall picture of block size by sorting the records into their blocks by File. If we have 3 records from File 2011 in a block and 4 from File 2012, we have a possible 4 unique individuals in the block. This gives us an idea of how well the baseline method will perform. We recall that the Gutman algorithm, the foundation of this work, is reliant upon the assumption that the sizes of the blocks involved are tiny. This allows the algorithm to more effectively explore the possible linking structures within each block.

```
#K = length(unique(blocks))

# Storage: How many records in each block?
#n.RIB = matrix(NA, nrow = 2, ncol = K)
# Storage: Which records in each block?
#RIB = vector("list", length = 2)
#RIB[[1]] = vector("list", length = K)
#RIB[[2]] = vector("list", length = K)

#for (k in 1:K) {
#    RIB[[1]][[k]] <- which(blocks1 == k)
```

```
#    RIB[[2]][[k]] <- which(blocks2 == k)
#}

#n.RIB[1, ] <- sapply(RIB[[1]], length)

#n.RIB[2, ] <- sapply(RIB[[2]], length)

#block.size <- apply(n.RIB, 2, max)

#table(block.size)
```

We notice that right now, the largest block contains a possible 9 unique individuals. This is a size that GAZM can still handle fairly effectively., as there are 9 choose 2, or 36, possible permutations (linking structures) within the block.

Seeds and Errors

For this simulation, we will begin by assuming that the records from 2011 are error free, but the new records from 2012 have yet to be verified. At this point, we need two more things before we can proceed with attempting to determine which records in the block correspond to which individual.

We begin with the need to identify Type 1 Seeds. Type 1 Seeds are records which, a priori, we believe are already matched to one another. For Gutman, it makes sense to choose these two ways. We note that in the Gutman framework, records have to stay in their original block. This means that these records must be matched to one another. By default, these can be treated as a priori Type 1 Seeds by the nature of the algorithm. This relies on no knowledge, but rather an inherent feature of the algorithm.

A second way that we obtain Type 1 Seeds is from expert information. If we somehow know that a set of records is exactly matched, then we declare tehse pairs Type 1 Seeds as well. Technically, based on the data that we have (prematched by SSN), we have a certain number which agree exactly on their blocking variables. If we can assume that we somehow know this, then we can proceed with calling these roughly 76000 records from 2012 Type 1 Seeds.

Using the first approach, we have roughly 60000 Singletons blocks, or blocks with exactly one record pair. However, some of these blocks have only a record from File 2011. They will never be matched with a record from 2012 using the Gutman algorithm.

This means that we have roughly 55000 Type 1 Seeds using the first approach to obtaining Type 1 Seeds.

The second approach to obtaining seeds yields roughly 76000 type 1 seeds. However, because of the reliance on the prematching information which is necessary for this step, for we rely on the first approach and keep the roughly 55000 seeds.

For GAZM we have roughly (K = 74000) blocks, and roughly 60% of the record pairs are Type 1 Seeds.

```
# Count the number of individuals currently suggested by the GAZM
#sum(block.size)

# Create a Vector for seed indicators. Remember that File 2012 is
#the file which we refer to when we discuss seeds
#Seeds = rep(0, N)

# How many pairs agree on their block?sum(blocks1 == blocks2)

# Which blocks contain exactly 1 pair?
#Singletons = which(block.size == 1)
```

```
# Which records from 2012 are assigned these blocks?
#type1Seeds = which(blocks1 == blocks2 & blocks2 %in% Singletons)

#Seeds[type1Seeds] <- 1

# Count these as type 1 seeds
#n.type1 = length(type1Seeds)
```

Model: matched data

In addition to the Type 1 Seeds, we also need some sort of model for the data in order to proceed with the GAZM.

There are again two ways that the model could come about. We could have models in mind or given by the client or we could play with the data and try to find the relationship.

The easiest way to do this would be to model a bivariate normal distribution and try to model the correlation between the two test scores. However, my extension relies on some dependence between a subset of the blocking variables and the response variable.

If we are going to be doing the modelling via exploring, then we need to limit our data pool to the type 1 seeds. These would be the only way we have to determine the relationship between the File 2011 and File 2012 records.

For relationships in the years individually, we could use what we have. From the EDA below, we note that while the math scores do not seem to vary significantly with gender, month, or birth day, there is a distinct difference in birth year and ethnicity. We may put gender in the model anyway for modelling purposes.

Another question of interest could be whether or not the relationship shifts from year to year, so we may want to model both ways. For now, we use the proper model

$$\begin{aligned} MathScore_{2012} = \beta_0 + \beta_1 MathScore_{2011} + \beta_2 year\_over\_1996 + \beta_3 ethblack \\ + \beta_4 ethhispanic + \beta_5 ethI + \beta_6 ethMulti + \beta_7 ethwhite \\ + \beta_8 sex\_male \end{aligned}$$

$$\begin{aligned} MathScore_{2011} = \eta_0 + \eta_1 yearover1996 + \eta_2 ethblack + \eta_3 ethhispanic + \\ \eta_4 ethI + \eta_5 ethMulti + \eta_6 ethwhite + \eta_7 sexmale \end{aligned}$$

```
# Create a data base of just the seeded data.
#Type1Only = WholeThing[type1Seeds, ]

# What is the relationship between the reading scores for 2012 and 2011?
#cor(Type1Only[, "mathscal"], Type1Only[, "mathscal2011"])

# We do have a problem with the linear model assumptions for reading
# score, which is why right now we are using math score.

# Exploratory Data Analysis

# Right now, it looks like it makes sense to include an indicator for
# whether or not the year is greater than 1996, and the ethnicity.

#EthNew = WholeThing$ethnic
```

```
#for (i in 1:N) {
#    if (EthNew[i] == "P") {
#        EthNew[i] <- "M"
#    }
#}
#Model1 = lm(mathscal ~ mathscal2011 + (birthyear > 1996) + EthNew, data = WholeThing)

#summary(Model1)

#ModelSeeds = lm(mathscal ~ mathscal2011 + (birthyear > 1996) + EthNew[type1Seeds], data = Type1Only)

#summary(ModelSeeds)
```

We notice that the released NCERDC data has different levels of ethnicity, which we will use for modelling purposes. Our blocks allow for the additional level of P (presumably Pacific Islander).

Type 2 Seeds

Type 2 seeds are records in which we "know" that the record is in the correct block, but don't know the match. This may be due to some verification in pre-processing, or, as wil be this case in this simulation, records which do not have another block that they are capable of moving to.

Another variant of Type 2 Seeds is to try and check the records which agree on the fields which I do not permit errors on. By default, they can't move blocks:

```
#perfect.fields = c(1, 3, 4)
#error.fields = c(2, 5)

#type2check = c()
#for (i in 68885:N) {
#    holder = as.numeric(sum(B[i, perfect.fields] == B[i + N, perfect.fields]) ==
#        3)
#    type2check[i] = holder
#}

#sum(type2check)/N
```

If we allow for errors only on the school variable, then we can only hope to recover these matches. We can potentially recover roughly 95% of the error prone records pairs, leaving the rest of the matches not recoverable. This constitutes less than .5% of the possible matches.

The Dirichlet Process

In this Markdown, we note the steps and stages necessary to run the Dirichlet Process Mixture model on the data for this project. While the draws from the DP occur during the sampler, we initialize with the seeds records.

We begin by loading in the working space, and all of the functions that we need.

```
# There are things we don't need
#rm(File2011, File2012, FileHolder, Type1Only, WholeThing, n.RIB, BlockRow, Block_Ops, DisagreePool, Et

### %%%%%%%%%%%%%%%%%% DP Functions #### %%%%%%%%%%%%%%%%%%
```

```
# DP Functions

initialize_DP <- function(Bhat1, Bhat2, prior.alpha, prior.Hstar, p) {
    alpha = prior.alpha
    Hstar = prior.Hstar

    pi.step = stick_breaking(Hstar, alpha)

    stacked = rbind(Bhat1, Bhat2)
    # if( is.list(stacked)=='TRUE'){ stacked = rbind( stacked[[1]],
    # stacked[[2]]) }
    N = nrow(stacked)

    phi <- c()  # Initialize an empty list
    for (h in 1:Hstar) {
        holder <- c()  #For each latent class, we will have a list of vectors. Each vector represents t
        # probabilities of seeing each field entry for each variable, given the
        # latent class.
        for (j in 1:p) {
            holder[[j]] <- data.frame(table(stacked[, j])/N)$Freq
        }
        phi[[h]] = holder
    }

    newlist <- list(alpha = alpha, pi.step = pi.step, phi = phi)

    return(newlist)
}

stick_breaking <- function(Hstar, alpha) {
    ## Draw our k betas from a beta(1,alpha) distribution
    first = Hstar - 1
    betas = c(rbeta(first, 1, alpha), 1)
    # How long is the stick?
    whats_left = c(1, cumprod(1 - betas))[1:Hstar]
    # We now need to actually get our vector of weights The weights are
    # calculated by multiplying the beta by the remaining length of the stick
    weights = whats_left * betas
    return(weights)
}

run_dpmpm_step <- function(Bhat, dpmpm, Hstar, HstarM1, J, hvec, realmin, N,
    a.alpha, b.alpha, a.dirichlet, d) {

    phi = dpmpm$phi
    pi.step = dpmpm$pi.step
    alpha = dpmpm$alpha

    # Update Z
    z <- c()
    for (i in 1:N) {
        holder <- holder.step(Bhat[i, ], phi, Hstar, J)
        num = pi.step * holder
```

```
        probs.z = num/sum(num)
        zi = sample(1:Hstar, 1, replace = T, prob = probs.z)
        z = c(z, zi)
    }

    # Update V
    nvec <- create_nvec(z, Hstar)
    V = c()
    for (h in 1:HstarM1) {
        V = c(V, update.Vh(h, nvec, alpha, Hstar))
    }
    V = c(V, 1)

    # Update pi How long is the stick?
    whats_left = c(1, cumprod(1 - V))[1:Hstar]
    # We now need to actually get our vector of weights The weights are
    # calculated by multiplying the beta by the remaining length of the stick
    pi.step = whats_left * V
    pi.step = ifelse(pi.step > 0, pi.step, realmin)  #Correction

    # phi
    phi <- c()
    for (h in 1:Hstar) {
        holder <- c()  #For each latent class
        for (j in 1:J) {
            avec = create_avec(z, Bhat, d, j, h, a.dirichlet)
            holder[[j]] <- c(rdirichlet(1, avec))
            # holder[[j]] <-ifelse( holder[[j]] > 0 , holder[[j]], inits.pm$realmin)
        }
        phi[[h]] = holder
    }

    # Update alpha
    alpha = rgamma(1, shape = (a.alpha + Hstar - 1), rate = (b.alpha - log(pi.step[Hstar])))
    alpha = ifelse(alpha > 0, alpha, realmin)

    newlist <- list(pi.step = pi.step, alpha = alpha, phi = phi, z = z)

    return(newlist)

}

holder.step <- function(B.i, phi, Hstar, J) {
    holder <- c()
    for (h in 1:Hstar) {
        holder = c(holder, prod.part(J, h, B.i, phi))
    }
    return(holder)
}

create_nvec <- function(z, Hstar) {
    nvec = matrix(0, ncol = Hstar, nrow = 1)  #Bins for the latent classes
    table.step = data.frame(table(z))
```

```
    places = as.numeric(levels(table.step$z))
    nvec[, places] = table.step$Freq
    return(nvec)
}

create_avec <- function(z, Bhat, d, j, h, a.dirichlet) {
    # For the simulated data, look at the observations in the first latent
    # class, and the 2nd blocking variable What is the spread of the data? How
    # many observations in each of the fields?
    avec = matrix(0, nrow = 1, ncol = d[j])  #Bins for the fields
    if (sum(z == h) > 0) {
        table.step = table(Bhat[which(z == h), j])
        table.step = data.frame(table.step)
        level.fill = table.step$Var1
        avec[, as.numeric(levels(level.fill))] = table.step$Freq
        # From the prior, we add a.dirichlet[[j]][ d[j] ]
        avec = avec + a.dirichlet
    } else {
        avec = avec + a.dirichlet
    }
    return(avec)
}

update.Vh <- function(h, nvec, alpha, Hstar) {
    a.upd = 1 + nvec[h]
    b.up = alpha + sum(nvec[(h + 1):Hstar])
    Vh.up = rbeta(1, a.upd, b.up)
    return(Vh.up)
}

rigamma <- function(n, alpha, beta) {
    if (alpha > 0 & beta > 0)
        1/rgamma(n = n, alpha, beta) else stop("rigamma: invalid parameters\n")
}

prod.part <- function(J, h, B.i, phi.term) {
    holder1 = 1
    for (j in 1:J) {
        holder.part = phi.term[[h]][[j]][B.i[j]]
        holder1 = holder1 %*% holder.part
    }
    return(holder1)
}

rdirichlet <- function(n, alpha) {
    l <- length(alpha)
    x <- matrix(rgamma(l * n, alpha), ncol = l, byrow = TRUE)
    sm <- x %*% rep(1, l)
    return(x/as.vector(sm))
}
```

Once the functions are loaded, we can begin to gather the constants that we need. We recall also that we are not using the day of birth in the Dirichlet process, so we can remove this column from consideration.

```
#B11hat = B11[type1Seeds, -1]
#B12hat = B12[type1Seeds, -1]
#dhat = d[-1]
```

We need to load (J), the number of categorical variables which will be used to assign records to latent groups. Note that in the coding, (J) is denoted (p). We also need to determine the number of clusters which we will allow. For now, we choose this (Hˆ{*} = 20). We also note that in order for this to run, the two (B) matrices must have the same column names.

```
#J = ncol(B11hat)

#colnames(B11hat) = colnames(B12hat)

#dpmpm = initialize_DP(B12hat, B11hat, prior.alpha = 1, prior.Hstar = 20, p = J)
```

The above code initializes the DP. We now have a few processing steps to do to get the data ready to run.

```
# Now, we need to convert everything to numbers

# Bhat = rbind(B12hat, B11hat)
# Bhat[, 1] = as.numeric(Bhat[, 1])
# Bhat[, 2] = as.numeric(Bhat[, 2])
# Bhat[, 3] = as.numeric(Bhat[, 3])
# Bhat[, 4] = as.numeric(Bhat[, 4])
# Bhat[, 5] = as.numeric(Bhat[, 5])
# Bhat = as.matrix(Bhat)

# # We have to relevel the years so that they are 1-5, etc. 1993 =1, 1994
# # =2, ...
# Bhat[, "birthyear"] = Bhat[, "birthyear"] - 1992
#
# Hstar = 20
# a = 1
# N = nrow(B11hat)
#
# itsDP = 2000

# dir.create(file.path(mainDir=getwd(), subDir='DP')) start.WD = getwd()
# setwd(file.path(mainDir=getwd(), subDir='DP'))

# for( s in 1:itsDP){ dpmpm = run_dpmpm_step( Bhat , dpmpm , Hstar ,
# HstarM1 = Hstar-1, J-1, hvec = 1:classes,realmin, N=2*N, a.alpha = .25,
# b.alpha = .25, 1, dhat )

# if( s > 1 & s%%2==0 ){ save( dpmpm, file=
# paste('dpmpm_',a,'out.RData',sep='')) a =a + 1 }

# if( s %% 100 == 0){ print(s)} }

# save.image('DPRun.RData')
```