Multilink Tutorial

Serge Aleshin-Guendel

9/7/2022

1 Introduction

The purpose of this tutorial to is to demonstrate how to use the R package `multilink` for multifile duplicate detection and record linkage. The tutorial will be broken into two tasks: data processing and running `multilink`. We use a synthetic example dataset throughout in order to demonstrate important steps in both tasks.

We will now load in the `multilink` package, and the `tidyverse` package which will be used for data processing.

```
library(multilink)
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.1.0     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

2 The Example Dataset

We have created a synthetic example dataset for the purpose of this tutorial. It was generated using using code from Peter Christen's group https://dmm.anu.edu.au/geco/index.php. We now read in the data set to give an overview for the reader.

```
load("mixed_data_messy.RData")
names(mixed_data_messy)
```

```
## [1] "mixed_data_file1" "mixed_data_file2" "mixed_data_file3" "IDs_file1"
## [5] "IDs_file2"        "IDs_file3"
```

`mixed_data_messy` is a list with six elements. The first three elements are three data files which we will be linking using `multilink`. The first data set is known a priori to have no duplicate records within it. The second and third data set are known a priori to have duplicates within them.

```
head(mixed_data_messy$mixed_data_file1)
```

```
##    sex    gname   fname     phone postcode occup
## 1    m    jarod hugg8ns 726264816      61q  <NA>
## 2    m    blak3  waller 864645259     6104  <NA>
## 3 <NA> harrison  varcoe 393726227      606     G
## 4    f      ruby   boyie 250951820      905  <NA>
## 5    f    ya5min haskett      <NA>      914  <NA>
## 6    f   lauren millary 84921z777      6o4     E
```

```
head(mixed_data_messy$mixed_data_file2)
```

```
##    sex     fname     phone postcode   age occup given_name
## 1    f  halligan      <NA>      715 35-44  <NA>       ella
## 2    f    oboyle 250951820      904  <NA>     F       ruby
## 3    m  fearnalx 264057941      607  <NA>     B   dimdothy
## 4 <NA> fearriall 264057941      617 25-34     B    timothy
## 5 <NA>      lock 467695008      615  <NA>     C      chl0e
## 6    m      hnna 781649813      913 55-59  <NA>       hugh
```

```
head(mixed_data_messy$mixed_data_file3)
```

```
##       sex    gname    fname     phone postcode   age occup
## 1       f stephani  everett 4813y2469      602  <NA>     G
## 2 unknown    jarod  huggins 7w6264816      614  <NA>     F
## 3       m    tyler  liersch      <NA>      61s  <NA>     A
## 4 unknown     ruby    boyoe 250951820      905  <NA>     F
## 5       m   tlmthy vearnall 264057941      607 25-34  <NA>
## 6       m   robbie    dakid 362144933      607 45-54  <NA>
```

The last three elements are IDs which represent which records are true matches. Any records across the three files which have the same IDs are true matches. For example, if two records both have the value 1 for their ID, they refer to the same individual. These IDS are for evaluating how the performance of `multilink`, and we do not use them until after running `multilink`.

3 Data Processing

Data processing is an extremely important part of using not just `multilink`, but any software for record linkage. We will first describe the format we will eventually need the data to be in for use in `multilink`, and then we will go about processing the data to get it into the required format.

3.1 Data format in multilink

Data will enter into `multilink` through the function `create_comparison_data`. The arguments of `create_comparison_data` which will be important for data processing are:

- `records`: A `data.frame` containing all the records to be linked, where each column is a field that will be compared. If there are more than one file, records should be obtained by stacking each file on top of each other (for example using `rbind` as we will show later). Missing values for each field need to be explicitly coded as `NA`. This means if a field has missing values coded as some other value (e.g. -1, "unknown", "missing", "John Doe"), we need to recode the field so that the missing values are all `NA`. Another important consideration is that if one file is missing a field altogether, every record in that file needs to have that field recorded as missing.
- `file_sizes`: A `numeric` vector that indicates the size of each file.
- `duplicates`: A `numeric` vector that indicates which files are assumed to have duplicates a priori. `duplicates[k]` should be 1 if file `k` has duplicates, and `duplicates[k]` should be 0 if file k has no duplicates. If any files do not have duplicates, we strongly recommend that the largest such file is organized to be the first file represented in `records`.
- `types`: A character vector, indicating the comparison to be used for each field (i.e. each column of records). The options are: "`bi`" for binary comparisons, "`nu`" for numeric comparisons (using the absolute difference), "`lv`" for string comparisons (using the normalized Levenshtein distance), "`lv_sep`" for string comparisons (using the normalized Levenshtein distance) where each string may contain

multiple spellings separated by the "|" character. What is important for data processing is that fields using options "bi", "lv", and "lv_sep" must be of class character, and fields using the "nu" option must be of class numeric. Further, string comparisons using the "lv" and "lv_sep" options are case-sensitive, so we need to make any fields that use string comparisons to be all lower or upper case.

3.2 Recoding missing values

The fields in each file already have missing values coded as NA, except for the field sex in mixed_data_file3. In this file, missing values of sex are coded as unknown. Let's recode these missing values to NA. Note that NA is a special value in R and not a character. See e.g. https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/NA for more information. If missing values are coded as the character "NA", they still need to be recoded to the non-character value NA.

```
mixed_data_messy$mixed_data_file3 <-
  mixed_data_messy$mixed_data_file3 %>%
  mutate(sex = ifelse(sex == "unknown", NA, sex))
```

3.3 Aligning fields

The fields in each file need to be "aligned" so that they can eventually be stacked together when being passed to create_comparison_data. What we mean by "aligned" here is that the columns of each file need to all have the same names when referring to the same fields, and the columns need to be all be in the same order.

We need to do a few tasks to align the three files. First, mixed_data_file1 is missing the age field. Thus we need to create a column age in mixed_data_file1 that has age as missing for each record.

```
mixed_data_messy$mixed_data_file1 <-
  mixed_data_messy$mixed_data_file1 %>%
  mutate(age = NA)
```

Next, in mixed_data_file2 the column recording given name is called given_name, rather than gname as in the two other files. Thus we need to rename the given_name column in mixed_data_file2 to gname.

```
mixed_data_messy$mixed_data_file2 <-
  mixed_data_messy$mixed_data_file2 %>%
  rename(gname = given_name)
```

Finally, the order of the columns are different in each file. Let's reorder the first two files so that they're in the same order as the third file.

```
mixed_data_messy$mixed_data_file1 <-
  mixed_data_messy$mixed_data_file1 %>%
  select(sex, gname, fname, phone, postcode, age, occup)

mixed_data_messy$mixed_data_file2 <-
  mixed_data_messy$mixed_data_file2 %>%
  select(sex, gname, fname, phone, postcode, age, occup)
```

3.4 Stacking files

Now that we have recoded all missing values to NA and aligned the files, let's stack the files.

```
records <- rbind(mixed_data_messy$mixed_data_file1,
                 mixed_data_messy$mixed_data_file2,
                 mixed_data_messy$mixed_data_file3)
```

We note here that when using multilink, we reccomend that you stack the largest file that is assumed to have no duplicates in it a priori as the first file (this lets multilink run a bit faster). Here we only have one file with no duplciates, the first file, and it's stacked first.

3.5 Making fields correct classes

We will be using string comparisons for `gname`, `fname`, `phone`, and `postcode`, and binary comparisons for the rest of the fields. Thus we need all of the fields to be of class `character`. This is already done in our example dataset, however it's important to check this before running `multilink`. If you end up not having the correct class for a given field, you will end up receiving an error message.

3.6 Making string comparison fields the same case

String comparisons are case sensitive, so we need to make sure any fields that use string comparisons are all upper or lower case. This is already done in our example dataset, as `gname`, `fname`, `phone`, and `postcode` are all lower case, however it's important to check this before running `multilink`. If you end up not having a field be mixed cases, you will *not* receive an error message.

4 Running multilink

Now that we have processed our example data set we will demonstrate how to use `multilink` to perform multifile duplicate detection and record linkage. This can be broken into five steps:

- Creating comparison data. `multilink` performs multifile duplicate detection and record linkage using a comparison-based approach. That is to say, for each pair of records that are potential matches, we generate comparisons for each field in that record pair that tell us how similar that field is. The function `create_comparison_data` takes in records and outputs comparisons based on some user arguments.
- Reducing comparison data. While this step is optional, in many applications it is advantageous computationally to reduce "filter" the comparisons. By "filter", we mean to use rules to declare a priori, based on values of comparisons, that certain record pairs are *not* potential matches. While this can potentially introduce false non-matches if the rules used are too strict, it greatly reduces the computation time necesarry to run `multilink`.
- Specifying prior information. `multilink` is a Bayesian method, and thus requires the user to specify prior distributions. As this is a more statistically technical component of `multilink`, we will introduce some default prior specifications for those with minimal experience with Bayesian methods.
- Run a Gibbs sampler. `multilink` uses a Gibbs sampler to explore the posterior distribution of linkages. This requires the user to specify a initial linkage and the number of iterations to run the Gibbs sampler. We will discuss some basic rules of thumb for setting the number of iterations. -Find and relabel the Bayes estimate. Using the samples for the Gibbs sampler, `multilink` can produce a point estimate for the linkage. As the comparison data was reduced in an earlier step, we need to relabel the point estimate so that it aligns with the initial input records.

4.1 Creating comparison data

The first step is to create comparison data. We have already in the data processing section processed the data files into the appropriate format for `create_comparison_data`.

Next, we need to specify the comparison type for each field using the `types` argument. We will be using string comparisons for `gname`, `fname`, `phone`, and `postcode`, and binary comparisons for the rest of the fields.

```r
types <- c("bi", "lv", "lv", "lv", "lv", "bi", "bi")
```

Rather than working with raw comparisons, we will discretize the comparisons. The `breaks` argument is a list with as many elements as there are fields being compared. The elements of `breaks` need to be in the same order as the columns of `records`. While binary comparisons are already discretized, the other comparison types are not, so we need to specify `breaks` for these fields.

In particular, elements of breaks corresponding to binary comparisons should be specified as `NA`. If a field is being compared with a numeric or string comparison, then the corresponding element of breaks should be a vector of cut points used to discretize the comparisons. To give more detail, suppose you pass in cut points `breaks[[f]]=c(cut_1, ...,cut_L)`. These cut points discretize the range of the comparisons into L+1 intervals: `I_0=(-∞, cut_1]`, `I_1=(cut_1, cut_2]`, ..., `I_L=(cut_L,∞]`. The raw comparisons, which lie in `[0,∞)` for numeric comparisons and `[0,1]` for string comparisons, are then replaced with indicators of which interval the comparisons lie in. The interval `I_0` corresponds to the lowest level of disagreement for a comparison, while the interval `I_L` corresponds to the highest level of disagreement for a comparison.

As a default for string comparisons, we recommend using the cut points `c(0, 0.25, 0.5)`. For these fields that use a "`lv`" or "`lv_sep`" string comparison type, we're using something called the normalized levenshtein distance to compare strings. For two strings `s_1` and `s_2` this is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one string into the other (see https://en.wikipedia.org/wiki/Levenshtein_distance), normalized by the length of the longer string. Because of this normalization, the raw comparisons using this comparison are between 0 and 1, with 0 meaning the strings are the same and 1 meaning they are completely different. When we specify the cut points `c(0, 0.25, 0.5)`, we're discretizing the string comparisons into 4 intervals: `I_0=(-∞,0]` (i.e. this interval just contains 0 since the raw comparisons can't be less than 0), `I_1=(0, 0.25]`, `I_2=(0.25,0.5]`, `I_3=(0.5,1]`. So `I_0` is the highest level of agreement (the strings are identical), and `I_3` is the lowest level of agreement. For two strings to have the lowest level of agreement, you'd need to make at least `A*0.5` edits, where `A` is the length of the longer string, to change one string into another.

```r
breaks <- list(sex = NA,
              gname = c(0, 0.25, 0.5),
              fname = c(0, 0.25, 0.5),
              phone = c(0, 0.25, 0.5),
              postcode = c(0, 0.25, 0.5),
              age = NA,
              occup = NA)
```

We need to pass in a vector with the sizes of each file, where the vector is in the same order as the files are stacked in `records`

```r
file_sizes <- c(nrow(mixed_data_messy$mixed_data_file1),
                nrow(mixed_data_messy$mixed_data_file2),
                nrow(mixed_data_messy$mixed_data_file3))
```

We need to pass in a vector indicating which files are assumed to have duplicates a priori. In our example, the first file has no duplicates, and the second and third file have duplicates.

```r
duplicates <- c(0, 1, 1)
```

Finally, we run `create_comparison_data`.

```r
comparison_list <- create_comparison_data(records = records,
                                          types = types,
                                          breaks = breaks,
                                          file_sizes = file_sizes,
                                          duplicates = duplicates)
```

```
## [1] "Creating comparison data for field sex"
## [1] "Creating comparison data for field gname"
## [1] "Creating comparison data for field fname"
## [1] "Creating comparison data for field phone"
## [1] "Creating comparison data for field postcode"
## [1] "Creating comparison data for field age"
## [1] "Creating comparison data for field occup"
```

4.2 Reducing comparison data

`comparison_list$comparisons` contains the discretized comparisons for each record pair. Note the columns labelled with " '. . . _DL_0" represent the highest level of agreement.

```
head(comparison_list$comparisons)
```

```
##      sex_DL_0 sex_DL_1 gname_DL_0 gname_DL_1 gname_DL_2 gname_DL_3 fname_DL_0
## [1,]    FALSE     TRUE      FALSE      FALSE      FALSE       TRUE      FALSE
## [2,]    FALSE     TRUE      FALSE      FALSE      FALSE       TRUE      FALSE
## [3,]     TRUE    FALSE      FALSE      FALSE      FALSE       TRUE      FALSE
## [4,]    FALSE    FALSE      FALSE      FALSE      FALSE       TRUE      FALSE
## [5,]    FALSE    FALSE      FALSE      FALSE      FALSE       TRUE      FALSE
## [6,]     TRUE    FALSE      FALSE      FALSE      FALSE       TRUE      FALSE
##      fname_DL_1 fname_DL_2 fname_DL_3 phone_DL_0 phone_DL_1 phone_DL_2
## [1,]      FALSE      FALSE       TRUE      FALSE      FALSE      FALSE
## [2,]      FALSE      FALSE       TRUE      FALSE      FALSE      FALSE
## [3,]      FALSE      FALSE       TRUE      FALSE      FALSE      FALSE
## [4,]      FALSE      FALSE       TRUE      FALSE      FALSE      FALSE
## [5,]      FALSE      FALSE       TRUE      FALSE      FALSE      FALSE
## [6,]      FALSE      FALSE       TRUE      FALSE      FALSE      FALSE
##      phone_DL_3 postcode_DL_0 postcode_DL_1 postcode_DL_2 postcode_DL_3
## [1,]      FALSE         FALSE         FALSE         FALSE          TRUE
## [2,]       TRUE         FALSE         FALSE         FALSE          TRUE
## [3,]       TRUE         FALSE         FALSE         FALSE          TRUE
## [4,]       TRUE         FALSE         FALSE          TRUE         FALSE
## [5,]       TRUE         FALSE         FALSE          TRUE         FALSE
## [6,]       TRUE         FALSE         FALSE         FALSE          TRUE
##      age_DL_0 age_DL_1 occup_DL_0 occup_DL_1
## [1,]    FALSE    FALSE      FALSE      FALSE
## [2,]    FALSE    FALSE      FALSE      FALSE
## [3,]    FALSE    FALSE      FALSE      FALSE
## [4,]    FALSE    FALSE      FALSE      FALSE
## [5,]    FALSE    FALSE      FALSE      FALSE
## [6,]    FALSE    FALSE      FALSE      FALSE
```

`comparison_list$record_pairs` contains the record pairs being compared in the corresponding rows of `comparison_list$comparisons`.

```
head(comparison_list$record_pairs)
```

```
##   i   j
## 1 1 182
## 2 1 183
## 3 1 184
## 4 1 185
## 5 1 186
## 6 1 187
```

Based on these comparisons, *and potentially other data not being used to link the files*, we create a logical vector, the same length as the number of rows of `comparison_list$comparisons` (i.e. the number of record pairs), that tells us whether we keep each record pair as a potential match. We will use the following rule: only pairs of records for which neither gname nor fname disagree at the highest level are potential matches.

```
pairs_to_keep <- (comparison_list$comparisons[, "gname_DL_3"] != TRUE) &
  (comparison_list$comparisons[, "fname_DL_3"] != TRUE)
```

Depending on the rule used to declare some record pairs are not potential matches, we could find ourselves in something like the following situation: record 1 is a potential match for record 2, but not record 3, and record 2 is potential match for record 3. This non-transitivity can potentially cause the Gibbs sampler to mix slowly. We include an argument `cc`, that when set to 1 takes the transitive closure of record pairs that are potential matches in `pairs_to_keep` (otherwise it should be set to 0).

There is a specific siutation where we *do not recommend setting* `cc` *to 1*: when we filter out comparisons that are *truly not possible matches*. For example, suppose records had timestamps, and records from file one could only match records from file two if they had an earlier timestamp. In a scenario like this, we would not want to set `cc` to 1, as it could potentially allow records from file one with timestamps later than records in file two to be potential matches. Outside of this type of situation, we recommend setting `cc` to 1.

```
cc <- 1
```

Finally, we run `reduced_comparison_list`.

```
reduced_comparison_list <-
  reduce_comparison_data(comparison_list = comparison_list,
                         pairs_to_keep = pairs_to_keep,
                         cc = cc)
```

4.3 Specify prior information

We use the function `specify_prior` to specify prior distributions. We will rely on defaults for most of the specifications. However, the following could be tuned by the user.

First, we specify, for each file, the maximum number of duplicates that could appear in a given file, as a `numeric` vector. Or in other words the maximum size of a cluster of matched records in a given file. For files with no duplicates, this should be set to 1. For files with duplicates, this should be based on the application. As files two and three in our example both have duplicates, we will set this upper bound to 10 for these files. This means that within each file, there cannot be a cluster of matched records larger than 10. Note that a cluster of matched records from all three files could still be larger than 10. In most applications, it is reasonable to set the upper bound for a file to be 10-20.

```
dup_upper_bound <- c(1, 10, 10)
```

Next, for each file with duplicates, we specify a prior distribution on the number of duplicates in each cluster from each file. Or in other words a prior on the sizes of clusters of matched records from each file. Currently, the user can specify a Poisson distribution with parameter λ, truncated to `[1, dup_upper_bound[k]]` for each file. As a default, we recommend specifying $\lambda = 1$ for each file, although this can be changed if there is expected to be a large or small amount of duplication in a given file.

```
dup_count_prior_family <- rep("Poisson", reduced_comparison_list$K)
dup_count_prior_pars <- rep(list(c(1)), reduced_comparison_list$K)
```

Finally, we run `specify_prior`.

```
prior_list <- specify_prior(comparison_list = reduced_comparison_list,
                            mus = NA,
                            nus = NA,
                            flat = 0,
                            alphas = NA,
                            dup_upper_bound = dup_upper_bound,
                            dup_count_prior_family = dup_count_prior_family,
                            dup_count_prior_pars = dup_count_prior_pars,
                            n_prior_family = NA,
                            n_prior_pars = NA)
```

4.4 Run Gibbs sampler

We now set the seed for Gibbs sampler and specify the number of iterations to run it for. We recommend initially setting the number of iterations to 1000-2000.

```
n_iter <- 2000
seed <- 44
```

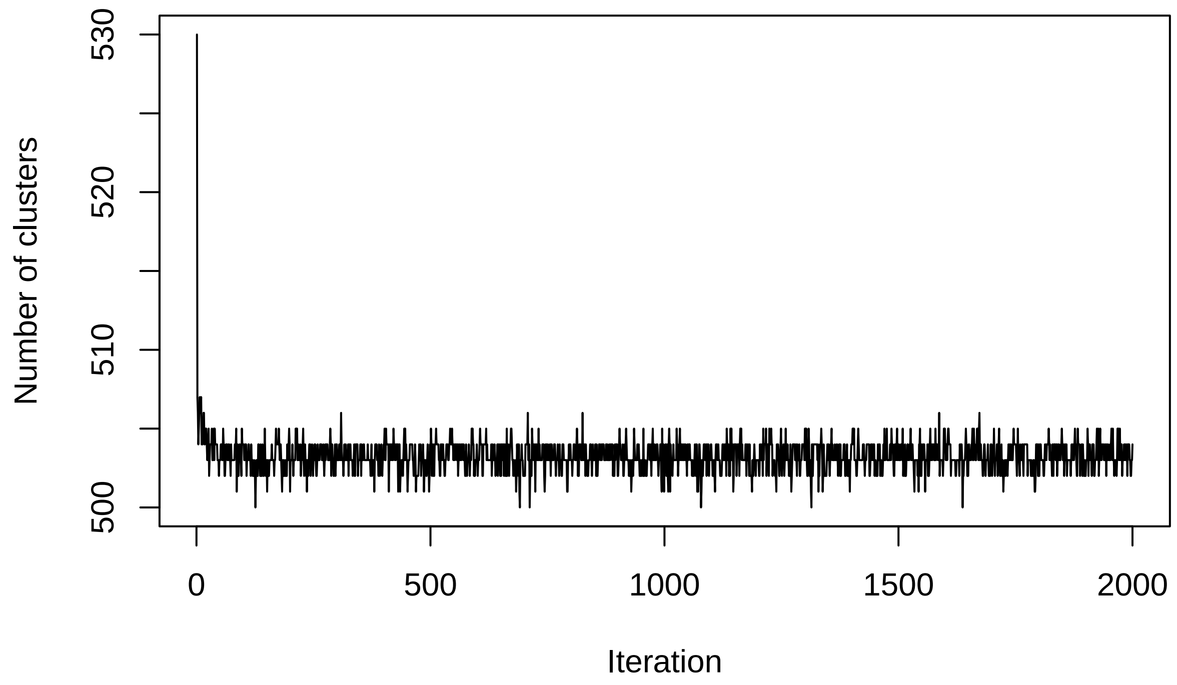
We run `gibbs_sampler`.

```
results <- gibbs_sampler(comparison_list = reduced_comparison_list,
                         prior_list = prior_list,
                         n_iter = n_iter,
                         seed = seed)
```

```
## [1] "Processing inputs"
## [1] "Beginning sampling"
## Beginning iteration 1/2000
## Beginning iteration 100/2000
## Beginning iteration 200/2000
## Beginning iteration 300/2000
## Beginning iteration 400/2000
## Beginning iteration 500/2000
## Beginning iteration 600/2000
## Beginning iteration 700/2000
## Beginning iteration 800/2000
## Beginning iteration 900/2000
## Beginning iteration 1000/2000
## Beginning iteration 1100/2000
## Beginning iteration 1200/2000
## Beginning iteration 1300/2000
## Beginning iteration 1400/2000
## Beginning iteration 1500/2000
## Beginning iteration 1600/2000
## Beginning iteration 1700/2000
## Beginning iteration 1800/2000
## Beginning iteration 1900/2000
## Beginning iteration 2000/2000
## [1] "Finished sampling in 2.18246507644653 seconds"
```

We then examine the trace plots for summaries of the posterior linkage. Here we examine the posterior chain for the number of clusters. We would like the chain to demonstrate mixing: after some iteration the chain levels off and appears to randomly scatter around some central value (this is an informal definition). If it appears the chain hasn't mixed yet, we should rerun the Gibbs sampler for a longer amount of iterations until it has.

```
plot(1:n_iter, colSums(results$contingency_tables), type = "l",
     ylab = "Number of clusters", xlab = "Iteration")
```



4.5 Find and relabel the Bayes estimate

We can now use the samples from the Gibbs sampler to compute a point estimate for the linkage. We need to specify the number of iterations of the Gibbs sampler to discard for burn in. This should be some iteration after the chain had demonstrated mixing. Based on the previous plot, a burn_in of 1000 would be appropriate.

```
burn_in <- 1000
```

There are a number of parameters that control the loss function used to compute the point estimate. We will set most of these to default values. However, there is one important parameter the user should set. That is the parameter that determines whether a full or partial point estimate is produced. A full estimate is an estimate where a decision is made for each record as to whether or not it matches any other records. A partial estimate is an estimate where for some records `multilink` abstains from making a decision due to uncertainty in the posterior, so that these records can then have their matching status decided in a manual review. To produce a full estimate, set `L_A` to `Inf`, otherwise to produce a partial estimate set `L_A` to a positive non-infinite value (as a default we recommend setting this value to 0.1). There is also a computational parameter `max_cc_size` that should be set based on type of estimate being produced. If a full estimate is being produced, we recommend setting `max_cc_size` to 50, and if a partial estimate is being produced we recommend setting `max_cc_size` to 10-12. A larger `max_cc_size` makes the approximation used to compute the point more accurate, but also make it more computationally expensive.

We now run `find_bayes_estimate` to produce both partial and full estimates.

```
L_A <- 0.1
max_cc_size <- 12
```

```
partial_estimate <- find_bayes_estimate(partitions = results$partitions,
                                        burn_in = burn_in,
                                        L_A = L_A,
                                        max_cc_size = max_cc_size)
```

```
## [1] "Finding Bayes estimate with a threshold of 0 and a maximum connected component of size 6"
```

```
L_A <- Inf
max_cc_size <- 50
```

```
full_estimate <- find_bayes_estimate(partitions = results$partitions,
                                     burn_in = burn_in,
                                     L_A = L_A,
                                     max_cc_size = max_cc_size)
```

```
## [1] "Finding Bayes estimate with a threshold of 0 and a maximum connected component of size 6"
```

Finally, we relabel the point estimates to get them to line up with the records that were originally input to `create_comparison_data`.

```
full_estimate_relabel <-
  relabel_bayes_estimate(reduced_comparison_list = reduced_comparison_list,
                         bayes_estimate = full_estimate)
partial_estimate_relabel <-
  relabel_bayes_estimate(reduced_comparison_list = reduced_comparison_list,
                         bayes_estimate = partial_estimate)
```

We can then attach the point estimates to the original records to examine which records `multilink` declared as matches.

```
records <- cbind(records,
                 full_estimate_id = full_estimate_relabel$link_id,
                 partial_estimate_id = partial_estimate_relabel$link_id)

head(records)
```

```
##    sex    gname   fname     phone postcode  age occup full_estimate_id
## 1    m    jarod hugg8ns 726264816      61q <NA>  <NA>                1
## 2    m    blak3  waller 864645259     6104 <NA>  <NA>              200
## 3 <NA> harrison  varcoe 393726227      606 <NA>     G              201
## 4    f     ruby   boyie 250951820      905 <NA>  <NA>                2
## 5    f   ya5min haskett      <NA>      914 <NA>  <NA>              202
## 6    f   lauren millary 84921z777      6o4 <NA>     E              203
##   partial_estimate_id
## 1                   1
## 2                 198
## 3                 199
## 4                   2
## 5                 200
## 6                 201
```