Worked linkage example

Auke Rijpma

December 5, 2022

Introduction

Below is a lightweight example of the record linkage procedure with `capelinker`. The cleaning step has been omitted and we work with data where a subset has already been manually linked. In most cases, both those labour-intensive steps would still have to be done.

Libraries

We begin by loading the capelinker library, as well as the stringdist, xgboost, stringi, and data.table libraries.

```
library("capelinker")
library("data.table")
library("stringdist")
library("stringi")
library("xgboost")
```

Capelinker is still under development, so it's useful to check the version. Here we use version 0.1.0.

```
packageVersion("capelinker")
```

```
## [1] '0.1.0'
```

The latest version could be installed using:

```
library("remotes")
remotes::install_github("rijpma/capelinker")
```

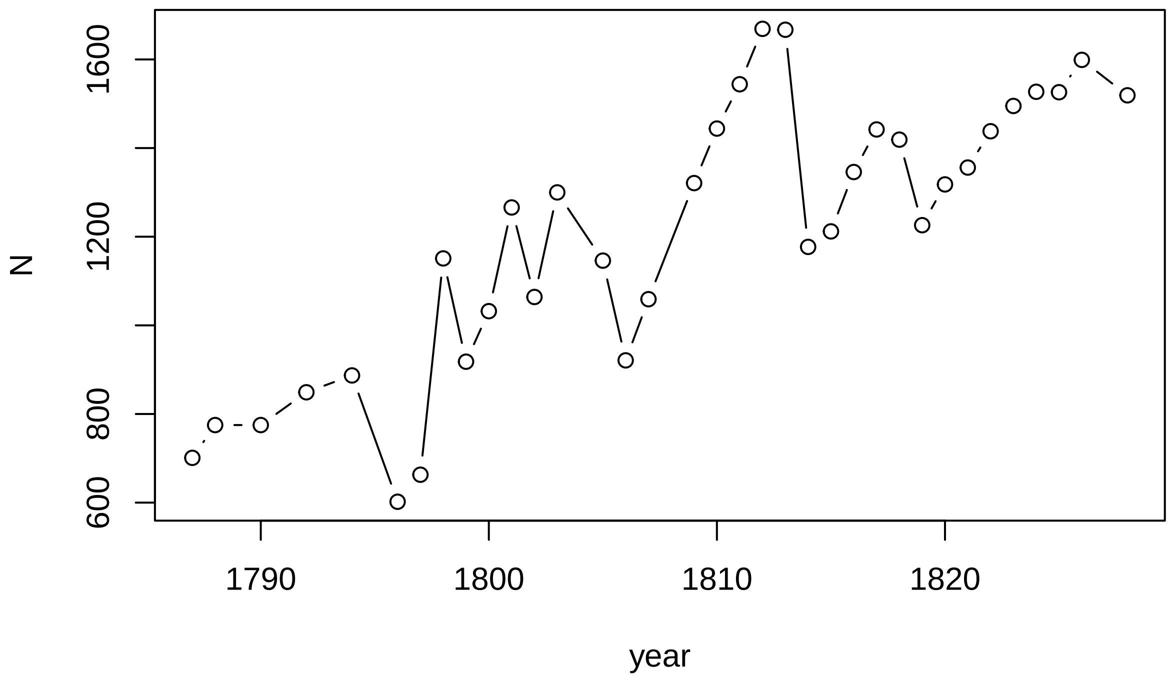
Loading data

Here we use the cleaned opgaafrolle data set for Graaff Reinet included with the `capelinker` package.

```
data(rein)
```

And a quick look at the data, here just the number of observations per year:

```
plot(rein[, .N, by = year], type = 'b')
```



It is important that each row in the dataset has unique identifiers than can be used to keep trach of observations along the linking process and so that the dataset can be reconstructed in the end. Here, this variable is `persid`. The name is a bit unfortunate as the opgaafrollen keep tack of farms/households, not persons.

Finally, note that part of the data has been manually linked by the `linkid` variable. This has been done for the letters A-L in the years 1828 and 1826.

```
rein[!is.na(linkid), unique(year)]
```

```
## [1] 1828 1826
```

```
rein[!is.na(linkid), sort(unique(stringi::stri_sub(mlast, 1, 1)))]
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "J" "K" "L"
```

However, when making your own training data, it is advised to sample from the entire dataset rather than on a subset like this which could lead to poor out-of-sample performance if other years or certain name groups are very different from the training data (for example, naming practices or demographic behaviour change). A separate vignette shows how to efficiently create datasets for manual labelling to deal with this.

Data check

We can use `preflight()` to check whether the data is ready for linkage by checking whether the data is consistently upper or lower case, patterns of missing data, and whether there aren't any unexpected (non-ascii) characters present. Because these data are pre-cleaned there, no real issues pop up here.

```
preflight(rein)
```

```
## Missing for  m_boost_stel_rein :
```

```
##  husb_wife_surname
## --------------
##                 expected_classes actual_classes
## mlast                  character      character
## mfirst                 character      character
## wlast                  character      character
## wfirst                 character      character
## winitials              character      character
## minitials              character      character
## mprof                  character           <NA>
## settlerchildren          numeric        numeric
## year                     numeric        integer
##
## Share missing variables (cannot be matched):
##          mlast      mfirst   minitials     wlast    wfirst winitials
## 1: 0.002549936 0.003234641 0.003234641 0.3553147 0.3569911 0.3569911
##    settlerchildren
## 1:    0.0002361052
##
##
## Share empty variables ("") (matching may fail):
##    mlast mfirst minitials wlast wfirst winitials settlerchildren
## 1:     0      0         0     0      0         0               0
##
##
## Share length one string variables ("*") (matching may fail):
##    mlast mfirst minitials        wlast       wfirst winitials settlerchildren
## 1:     0      0  0.349788 0.0001464933 0.0001101564 0.2168613       0.9931277
##
##
## Share multiword string variables (models expect joint middle names or joint prefixes):
##        mlast   mfirst minitials     wlast    wfirst winitials settlerchildren
## 1: 0.2479761 0.650212         0 0.2658121 0.7831387         0               0
##
##
## Share Sentence Case (matching requires consistency in case between datasets):
##    mlast mfirst minitials wlast wfirst winitials settlerchildren
## 1:     0      0         0     0      0         0               0
##
##
## Share UPPER CASE (matching requires consistency in case between datasets):
##    mlast mfirst minitials wlast wfirst winitials settlerchildren
## 1:     1      1         1     1      1         1               1
##
##
## Share lower case (matching requires consistency in case between datasets):
##    mlast mfirst minitials wlast wfirst winitials settlerchildren
## 1:     0      0         0     0      0         0               1
##
##
## Share leading and trailing spaces (count towards string distances):
##    mlast mfirst minitials wlast wfirst winitials settlerchildren
## 1:     0      0         0     0      0         0               0
##
```

```
##
## Share accented letters and non-alphabetic symbols per variable (matching might require consistency):
##          mlast     mfirst minitials       wlast     wfirst winitials
## 1: 0.004131841 0.02462577         0 0.002549936 0.02394107         0
##    settlerchildren
## 1:       0.9997639
##
##
## Character set
## $mlast
## [1] " .ABCDEFGHIJKLMNOPQRSTUVWXYZ"
##
## $mfirst
## [1] " .ABCDEFGHIJKLMNOPQRSTUVWXYZ"
##
## $minitials
## [1] "ABCDEFGHIJKLMNOPRSTUVWYZ"
##
## $wlast
## [1] " .ABCDEFGHIJKLMNOPQRSTUVWXYZ"
##
## $wfirst
## [1] " .ABCDEFGHIJKLMNOPRSTUVWXYZ"
##
## $winitials
## [1] "ABCDEFGHIJKLMNOPRSTUVWZ"
##
## $settlerchildren
## [1] "0123456789"
##
##
##
## Accented letters and non-alphabetic symbols
## $mlast
## [1] "."
##
## $mfirst
## [1] "."
##
## $minitials
## [1] ""
##
## $wlast
## [1] "."
##
## $wfirst
## [1] "."
##
## $winitials
## [1] ""
##
## $settlerchildren
## [1] "3045162789"
##
```

```
##
##
## Range (important for numeric variables):
##     mlast          mfirst minitials    wlast     wfirst winitials
## 1:   ABUE       A NDRIES         A     AARTS         A.         A
## 2: ZWARTZ ZIRK BERNARDUS        ZW ZWIEGERS ZYNA MARIA        ZS
##    settlerchildren
## 1:               0
## 2:              14
```

Candidates

A function `candidates()` is included to help in the making of a candidate set – a subset of all possible combinations that contains only plausible links. This is done on a pair of variables called the "blocking key", which preferably are reliable and cheap to compare between the datasets. For the Cape Colony we very rarely have good blocking keys like year of birth to make the candidates, so `candidates()` relies on a number of string distance functions to create the candidates, and these can take a lot of time to compute. The least inefficient way to do that is the default "bigram distance" option.

```
cnd = capelinker::candidates(
    dat_from = rein[year == 1828],
    dat_to = rein[year < 1828],
    blocktype = "bigram distance",
    blockvariable_from = "mlast",
    blockvariable_to = "mlast",
    maxdist = 0.5)
```

Here we create candidates for linkage between the years 1828 and all other years for the Graaff Reinet district. We also tell `candidates()` the variables you want to block on, and how much dissimilarity you're willing to accept (here 0.5 on a scale 0–1). Often you can be a bit stricter but it is advised to check the data to find out what works for you.

A quick look at what the candidates look like:

```
nrow(cnd)
```

```
## [1] 1448088
```

```
cnd[, .N, by = list(persid_from, year_to)][, mean(N)]
```

```
## [1] 30.62015
```

```
cnd[persid_from == 1 & year_to == 1826,
    .SD,
    .SDcols = patterns("(mlast|mfirst|wlast|wfirst)_(from|to)")]
```

```
##      mlast_from     mfirst_from wlast_from      wfirst_from     mlast_to
## 1: SCHINDEHUTTE JOHANNES JURGEN       WEIS JACOBA ELISABETH      SCHUTTE
## 2: SCHINDEHUTTE JOHANNES JURGEN       WEIS JACOBA ELISABETH SCHINDEKUTTE
##           mfirst_to wlast_to        wfirst_to
## 1: CHRISTIAAN ERNST DU PREEZ             ANNA
## 2: JOHANNES JURGENS     WEIS JACOBA ELISABETH
```

We have over 1.4 million candidates, with each household from 1828 having on average 31 candidates per year. For `persid` 1 in 1826 we have only two candidates, the second of which seems to be the correct match.

Any identical names in the two datasets get the suffices `_from` and `_to`. If you don't have the same column names, you'll have to keep track of the variables in some other way.

String distances

To train a model and use it for predictions we need to quantify the difference between the candidate pairs. For the names we'll rely on string distances implemented in the `stringdist` package to do that. Specifically, we use the Jaro-Winkler distances which downweight differences in the final characters as these are expected to vary more than the first characters.

```
cnd[, mlastdist := stringdist::stringdist(mlast_from, mlast_to, method = "jw")]
cnd[, mfirstdist := stringdist::stringdist(mfirst_from, mfirst_to, method = "jw")]
cnd[, wlastdist := stringdist::stringdist(wlast_from, wlast_to, method = "jw")]
cnd[, wfirstdist := stringdist::stringdist(wfirst_from, wfirst_to, method = "jw")]
cnd[, matches := .N, by = persid_from]

cnd[persid_from == 1 & year_to == 1826,
    .SD,
    .SDcols = patterns("(mlast|mfirst|wlast|wfirst)_(from|to)|dist$")]
```

```
##      mlast_from     mfirst_from wlast_from      wfirst_from     mlast_to
## 1: SCHINDEHUTTE JOHANNES JURGEN       WEIS JACOBA ELISABETH      SCHUTTE
## 2: SCHINDEHUTTE JOHANNES JURGEN       WEIS JACOBA ELISABETH SCHINDEKUTTE
##           mfirst_to wlast_to        wfirst_to  mlastdist mfirstdist wlastdist
## 1: CHRISTIAAN ERNST DU PREEZ             ANNA 0.21031746 0.42638889         1
## 2: JOHANNES JURGENS     WEIS JACOBA ELISABETH 0.05555556 0.02083333         0
##    wfirstdist
## 1:  0.4583333
## 2:  0.0000000
```

Training

For the remainder of this example, however, we work with the manually created links in the Graaff Reinet data. Remember that for the years 1828 and 1826 the surnames starting with A-L were linked using a variable called `linkid` which gave the same number to both households. We subset our complete candidates data set, `cnd`, to these observations and create an outcome variable from the `linkid` variable. Note that we assume that wherever no value for `linkid` was assigned, the observation could not be linked and we impute that `correct`, the variable indicating whether a candidate link pair is correct, should be 0.

```
cnd_lbl = cnd[year_to == 1826
    & stringi::stri_detect_regex(mlast_from, "^[A-L]")
    & stringi::stri_detect_regex(mlast_to, "^[A-L]")]
cnd_lbl[, correct := linkid_from == linkid_to]
cnd_lbl[is.na(correct), correct := 0]
```

This data is next split in a training and test data set, using the latter to asses the model's performance without the performance metrics being affected by overfitting. We use a 70-30 split for the data. Note that we split the data on the basis of the `persid_from` identifier, not on the rows, to ensure that full candidate blocks end up in the training and test data sets.

```
set.seed(123871)
share_train = 0.7
sample_ids = sample(x = unique(cnd_lbl$persid_from),
    size = ceiling(length(unique(cnd_lbl$persid_from)) * share_train))

cnd_lbl[, train := persid_from %in% sample_ids]
trn = cnd_lbl[train == 1]
tst = cnd_lbl[train == 0]
```

We next select a number of features to predict correct links. Here we keep the model very simple and only look at the string distances of the the spouses' first and surnames as well as the number of matches in a candidate block. The models we used to link within opgaafrollen (and which are available as pretrained models) have more features to squeeze out a few percentage points better performance. As a next step, we create `xgboost`'s specific matrix objects, `xgb.DMatrix`. R's formula interface could also be used for this, but it takes an extra step to convert this object to an `xgboost`-ready object. (`capelinker` has a convenience function for this: `xgbm_ff()`).

```
predictors = c("mlastdist", "mfirstdist", "wlastdist", "wfirstdist", "matches")
trn_xgb = xgboost::xgb.DMatrix(as.matrix(trn[, ..predictors]), label = trn$correct)
tst_xgb = xgboost::xgb.DMatrix(as.matrix(tst[, ..predictors]), label = tst$correct)
```

Next, we train a model on the basis of this data. Our best experiences are with a gradient boosting model, as implemented in the `xgboost` library. It is as accurate as any other model we tried, but also performant in terms of computational efficiency as the dataset grows larger. It also deals with missing values which are a frequent issue in historical data like this. As a downside it comes with lots of tuning parameters, most of them to prevent overfitting. Below we largely stick to the default parameters. It is important to set the model objective to "binary:logistic" when working with a binary outcome, or performance will be much reduced.

```
m = xgboost::xgb.train(
    data = trn_xgb,
    nrounds = 500,
    params = list(
        max_depth = 6,          # default 6
        min_child_weight = 1,   # default 1
        gamma = 1,              # default 0
        eta = 0.3,              # default 0.3
        subsample = 0.8,        # default 1
        colsample_bytree = 0.5, # default 1
        objective = "binary:logistic"
))
```

Evaluation

Next we evaluate how well the model performs. We do this on the basis of three metrics (calculated from the number of True Positives (TP), False Positives (FP), and False Negatives (FN):

- precision, $\frac{TP}{TP+FP}$, the share of true predictions that are actually correct,
- recall: $\frac{TP}{TP+FN}$, the share of actually correct links that are predicted as true
- F1: $\frac{2}{prec^{-1}+rec^{-1}}$, the harmonic mean of the precision and the recall.

More informally, the recall tells us how many correct links we manage to recover; and the precision tells us how many of the predicted links are actually correct. If the model performs poorly, we can tweak the model and re-evaluate. We use the test data for these evaluations.

We first have a quick look at the relative importance of each feature in prediction correct links using the `xgb.importance` function. The male first name contributes most, and the male surname the least. The latter's unimportance is because we use the surname to create the candidate blocks, so each candidate pair should have similar surnames and there should be little variation for this features for the model to exploit.

```
xgboost::xgb.importance(model = m)
```

```
##       Feature       Gain      Cover  Frequency
## 1: mfirstdist 0.38484792 0.30924512 0.38566553
## 2:  wlastdist 0.22959988 0.09660071 0.05688282
## 3: wfirstdist 0.21935459 0.29606707 0.16723549
## 4:    matches 0.13253295 0.23687691 0.32536974
## 5:  mlastdist 0.03366467 0.06121019 0.06484642
```

Evaluation is then done by creating a data.table containing the actual correct links from the test data and the associated predictions from the model. These are first cross-tabulated into a confusion matrix.

```
predictions = data.table(
    correct = tst$correct,
    predicted = predict(m, newdata = tst_xgb))

knitr::kable(table(predictions$correct, predictions$predicted > 0.5))
```

	FALSE	TRUE
FALSE	2395	13
TRUE	22	116

The metrics are calculated with the `Metrics` library. In all cases the true prediction threshold is set to 0.5. We can use use precision-recall curves to choose a better value, but so far the experience is that 0.5 works pretty well as nearly all true and false links tend to cluster at high (>0.8) and low (<0.2) predicted probabilities ayway.

```
Metrics::precision(predictions$correct, predictions$predicted > 0.5)
```

```
## [1] 0.8992248
```

```
Metrics::recall(predictions$correct, predictions$predicted > 0.5)
```

```
## [1] 0.8405797
```

```
Metrics::fbeta_score(predictions$correct, predictions$predicted > 0.5)
```

```
## [1] 0.8689139
```

Here the precision and recall are 90% and 84% respectively, and combine to an F-beta score of 87%, which is already fairly good for such a simple model. Adding more features would increase the scores, but probably make the model less portable.

Prediction

Finally, predicting links on the basis of this model is done with the generic `predict()` function and inserting them into the candidates data. The `newdata` argument is needed for these predictions, and `xgboost` again requires this data to be in the from of an `xgb.DMatrix`. We use the `feature_names` element from the model `m` to construct the matrix.

```
m$feature_names
```

```
## [1] "mlastdist"  "mfirstdist" "wlastdist"  "wfirstdist" "matches"
```

```
cnd[,
    pred := predict(
        object = m,
        newdata = xgboost::xgb.DMatrix(as.matrix(.SD)))
    , .SDcols = m$feature_names]
```

Link selection

As a final step, we need to use the predicted probabilities to select links. Here we create two rank variables, one finding the best candidate for a household in a candidate block (`persid_from`), and one finding the best links to a certain household across all candidate blocks (elsewhere we have just gone with the best candidate within each block). Only when a candidate pairs is thethe best in both rangks, and the predicted probability is higher than 0.5, do we keep the link. It is possible to be more or less strict, both in terms of the probability threshold (which should be evaluated on the basis of the metrics above), and whether the observation should be best on two ranks or only one.

```
cnd[, rnk_from := rank(-pred), by=list(year_to, persid_from)]
cnd[, rnk_to := rank(-pred), by=list(persid_to)]

out = cnd[rnk_to == 1 & rnk_from == 1 & pred > 0.5, list(persid_from, persid_to, pred)]
```

When exporting the linkset we of course include the two household identifiers to . We also include the predicted probability, `pred` so users can take the confidence the model had in the link onboard when working with the data.

Panel reconstruction

These links can now be used to reconstruct the panel, but this will not be covered here. When linking two cross-sections only, using this linkset should be straightforward.