Module 4: Probabilistic Blocking, Part II

Rebecca C. Steorts

Agenda

- Data Cleaning Pipeline
- Blocking
- Locality Sensitive Hashing (LSH)
- Hash functions
- Hashed shingles
- Signatures
- Characteristic Matrix
- Minhash (Jaccard Similarity Approximation)
- Back to LSH

Load R packages

```
library(RecordLinkage)
library(blink)
library(knitr)
library(textreuse) # text reuse/document similarity
library(tokenizers) # shingles
library(devtools)
library(cora)
library(ggplot2)
# install_github("resteorts/cora")
data(cora) # load the cora data set
```

Data Cleaning Pipeline

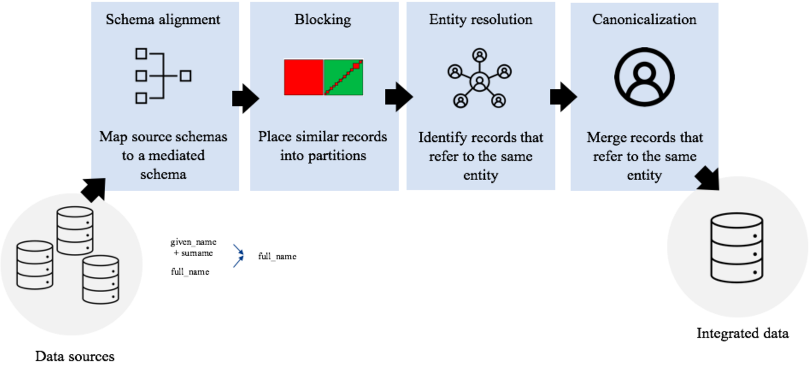

Schema alignment
Map source schemas to a mediated schema
Blocking
Place similar records into partitions
Entity resolution
Identify records that refer to the same entity
Canonicalization
Merge records that refer to the same entity
Data sources
given_name + surname
full_name
full_name
Integrated data

Figure 1: Data cleaning pipeline.

Blocking

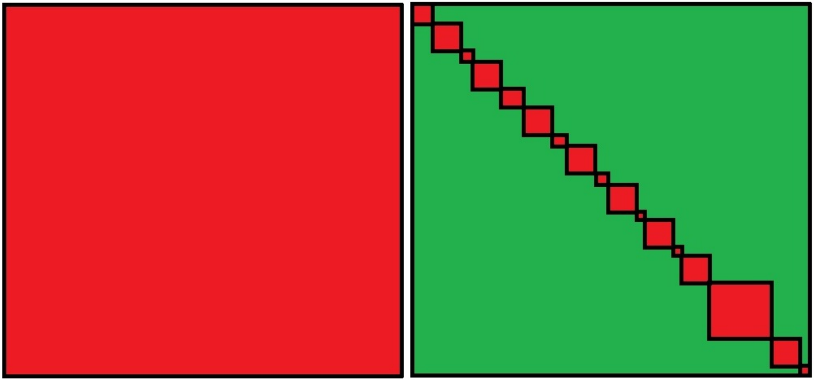


Figure 2: Left: All to all record comparison. Right: Example of resulting blocking partitions.

LSH

Locality sensitive hashing (LSH) is a fast method of blocking for record linkage that orginates from the computer science literature.

Citation data set

Recall that last time we worked with the cora citation data set.

```
# get only the columns we want
# number of records
n <- nrow(cora)
# create id column
dat <- data.frame(id = seq_len(n))
# get columns we want
dat <- cbind(dat, cora[, c("title", "authors", "journal")])
shingles <- apply(dat, 1, function(x) {
  # tokenize strings
  tokenize_character_shingles(paste(x[-1], collapse=" "), n = 3)
})
# empty holder for similarities
jaccard <- expand.grid(record1 = seq_len(n),
                       record2 = seq_len(n))

# don't need to compare the same things twice
jaccard <- jaccard[jaccard$record1 < jaccard$record2,]
```

Hash function overview

- Traditionally, a *hash function* maps objects to integers such that similar objects are far apart
- Instead, we want special hash functions that do the **opposite** of this, i.e. similar objects are placed closed together!

Hash function

Hash functions $h()$ are defined such that

> *If records A and B have high similarity, then the probability that* $h(A) = h(B)$ *is* ***high*** *and if records A and B have low similarity, then the probability that* $h(A) \neq h(B)$ *is* ***low****.*

Hashing shingles

Instead of storing the strings as shingles, we can instead store the *hashed values*

These are integers (instead of strings), and they take up less space/memory.

Hashing shingles (continued)

```r
# instead store hash values (less memory)
hashed_shingles <- apply(dat, 1, function(x) {
  # get the string
  string <- paste(x[-1], collapse=" ")
  shingles <-
    # 3-shingles
    tokenize_character_shingles(string, n = 3)[[1]]
  # return hashed shingles
  hash_string(shingles)
})
```

Hashing shingles (continued)

```r
# Jaccard similarity on hashed shingles
hashed_jaccard <-
  expand.grid(record1 = seq_len(n), record2 = seq_len(n))

# don't need to compare the same things twice
hashed_jaccard <-
  hashed_jaccard[hashed_jaccard$record1 < hashed_jaccard$record2
# see how long this takes
time <- Sys.time()
hashed_jaccard$similarity <-
  apply(hashed_jaccard, 1, function(pair) {
  jaccard_similarity(hashed_shingles[[pair[1]]],
                     hashed_shingles[[pair[2]]])
}) # get jaccard for each hashed pair
time <- difftime(Sys.time(), time, units = "secs")
```

Hashing shingles (continued)

- To hash the shingles, it took 6.53296×10^5 bytes.
- To store the shingles, it took 8.411816×10^6 bytes.
- The entire pairwise comparison still took the same amount of time (≈ 1.82 minutes) for both approaches.

Similarity preserving summaries of sets

Sets of shingles are large (larger than the original data set)

If we have millions of records in our data set, it may not be possible to store all the shingle-sets in memory

We can replace large sets by smaller representations, called *signatures*

We can use signatures to **approximate** Jaccard similarity

Characteristic matrix

In order to get a signature of our data set, we build a *characteristic matrix*

Columns correspond to records and the rows correspond to all hashed shingles

Helper Function for Building Characteristic Matrix

```r
# return if an item is in a list
item_in_list <- function(item, list) {
  as.integer(item %in% list)
}
```

Characteristic matrix

```r
# get the characteristic matrix
# items are all the unique hash values
# columns will be each record
# we want to keep track of where each hash is included
char_mat <- data.frame(item = unique(unlist(hashed_shingles)))
# for each hashed shingle, see if it is in each row
contained <- lapply(hashed_shingles, function(col) {
  vapply(char_mat$item, FUN = item_in_list,
         FUN.VALUE = integer(1), list = col)
})
# list to matrix
char_mat <- do.call(cbind, contained)
# row names
rownames(char_mat) <- unique(unlist(hashed_shingles))
# column names
colnames(char_mat) <- paste("Record", seq_len(nrow(dat)))
```

Characteristic matrix (continued)

```r
# inspect results
kable(char_mat[10:15, 1:5])
```

	Record 1	Record 2	Record 3	Record 4	Record 5
-78464425	1	1	1	1	1
-78234440	1	0	0	0	0
-78221717	1	0	0	0	0
-78235289	1	1	1	1	1
-78555255	1	1	1	1	1
-78132973	1	1	1	1	1

The result is a 3551×1879 matrix

Question: Why would we not store the data as a characteristic matrix?

Minhashing

We want to create the signature matrix through minhashing

1. Permute the rows of the characteristic matrix m times
2. Iterate over each column of the permuted matrix
3. Populate the signature matrix, row-wise, with the row index from the first 1 value found in the column

The signature matrix is a hashing of values from the permuted characteristic matrix and has one row for the number of permutations calculated (m), and a column for each record

Minhashing (cont'd)

```r
# set seed for reproducibility
set.seed(02082018)
# function to get signature for 1 permutation
get_sig <- function(char_mat) {
  # get permutation order
  permute_order <- sample(seq_len(nrow(char_mat)))
  # get min location of "1" for each column (apply(2, ...))
  t(apply(char_mat[permute_order, ], 2,
          function(col) min(which(col == 1))))
}
# repeat many times
m <- 360
sig_mat <- matrix(NA, nrow=m,
                  ncol=ncol(char_mat)) #empty matrix
for(i in 1:m) {
  sig_mat[i, ] <- get_sig(char_mat) #fill matrix
}
colnames(sig_mat) <- colnames(char_mat) #column names
```

Minhashing (cont'd)

```r
# inspect results
kable(sig_mat[1:10, 1:5])
```

Record 1	Record 2	Record 3	Record 4	Record 5
3	3	3	3	3
38	38	38	38	38
46	46	46	46	46
36	36	36	36	36
31	31	31	31	31
124	124	124	124	124
21	21	21	21	21
9	9	9	9	9
85	85	85	85	85
44	44	44	44	44

Signature matrix and Jaccard similarity

The relationship between the random permutations of the characteristic matrix and the Jaccard Similarity is

$$Pr\{\min[h(A)] = \min[h(B)]\} = \frac{|A \cap B|}{|A \cup B|}$$

We use this relationship to **approximate** the similarity between any two records

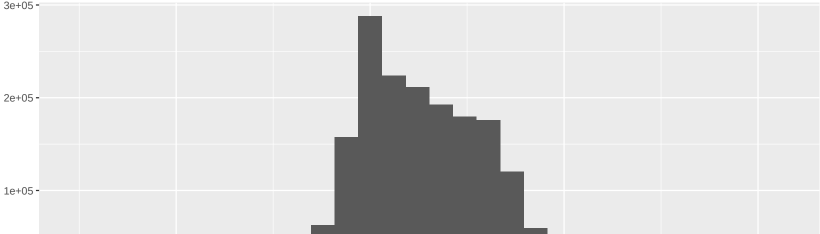
We look down each column of the signature matrix, and compare it to any other column

The number of agreements over the total number of combinations is an approximation to Jaccard measure

Jaccard similarity approximation

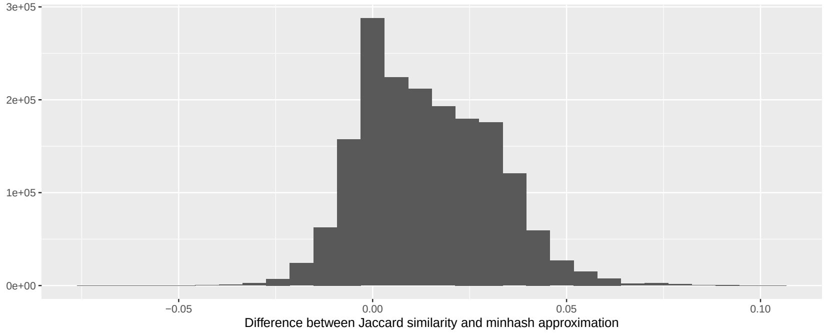
```
# add jaccard similarity approximated from the minhash to compar
# number of agreements over the total number of combinations
hashed_jaccard$similarity_minhash <-
  apply(hashed_jaccard, 1, function(row) {
  sum(sig_mat[, row[["record1"]]]
      == sig_mat[, row[["record2"]]])/nrow(sig_mat)
})

# how far off is this approximation? plot differences
qplot(hashed_jaccard$similarity_minhash - hashed_jaccard$similar
  xlab("Difference between Jaccard similarity and minhash approx
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwi
```



Jaccard similarity approximation

```
## `stat_bin()` using `bins = 30`. Pick better value with `
```



Used minhashing to get an approximation to the Jaccard similarity, which helps by allowing us to store less data (hashing) and avoid storing sparse data (signature matrix)

Wait did I miss something?

We still haven not addressed the issue of **pairwise comparisons**

Locality Sensitive Hashing (LSH)

We want to hash items several times such that similar items are more likely to be hashed into the same bucket.

1. Divide signature matrix into b bands with r rows each so $m = b * r$ where m is the number of times that we drew a permutation of the characteristic matrix in the process of minhashing
2. Each band is hashed to a bucket by comparing the minhash for those permutations
   - If they match within the band, then they will be hashed to the same bucket
3. If two documents are hashed to the same bucket they will be considered candidate pairs

We only check *candidate pairs* for similarity

Banding and buckets

```r
# view the signature matrix
print(xtable::xtable(sig_mat[1:10, 1:5]), hline.after = c(
```

	Record 1	Record 2	Record 3	Record 4	Record 5
1	3	3	3	3	3
2	38	38	38	38	38
3	46	46	46	46	46
4	36	36	36	36	36
5	31	31	31	31	31
6	124	124	124	124	124
7	21	21	21	21	21
8	9	9	9	9	9
9	85	85	85	85	85
10	44	44	44	44	44

Tuning

How to choose k

How large k should be depends on how long our data strings are
The important thing is k should be picked large enough such that the probability of any given shingle is *low*

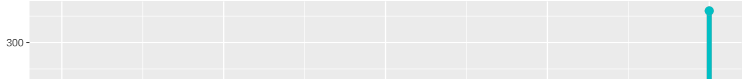
How to choose b

b must divide m evenly such that there are the same number of rows r in each band
What else?

Choosing b

$P(\text{two documents w/ Jaccard similarity } s \text{ marked as potential match}) = 1-(1-s^{m/b})^b$

```
# library to get divisors of m
library(numbers)

# look at probability of binned together for various bin s
bin_probs <- expand.grid(s = c(.25, .75), h = m, b = diviso
bin_probs$prob <- apply(bin_probs, 1, function(x) lsh_proba


# plot as curves
ggplot(bin_probs) +
  geom_line(aes(x = prob, y = b, colour = factor(s), group
  geom_point(aes(x = prob, y = b, colour = factor(s)), size
  xlab("Probability") +
  scale_color_discrete("s")
```



“Easy” LSH in R

There an easy way to do LSH using the built in functions in the `textreuse` package via the functions `minhash_generator` and `lsh` (so we don’t have to perform it by hand):

```r
# choose appropriate num of bands
b <- 90

# create the minhash function
minhash <- minhash_generator(n = m, seed = 02082018)
```

“Easy” LSH in R (Continued)

```r
# build the corpus using textreuse
docs <- apply(dat, 1, function(x) paste(x[-1], collapse = "
names(docs) <- dat$id # add id as names in vector
corpus <- TextReuseCorpus(text = docs, # dataset
                          tokenizer = tokenize_character_sh
                          progress = FALSE, # quietly
                          keep_tokens = TRUE, # store shing
                          minhash_func = minhash) # use mi
```

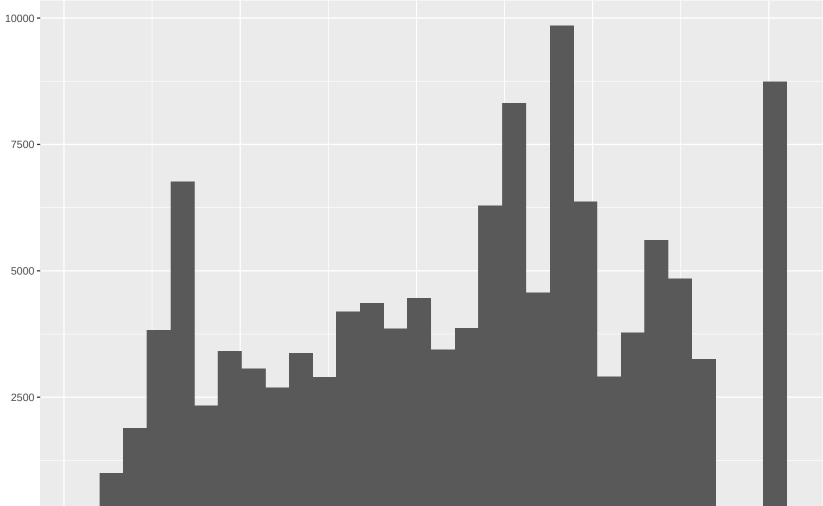
“Easy” LSH in R (Continued)

```r
# perform lsh to get buckets
buckets <- lsh(corpus, bands = b, progress = FALSE)

# grab candidate pairs
candidates <- lsh_candidates(buckets)

# get Jaccard similarities only for candidates
lsh_jaccard <- lsh_compare(candidates, corpus, jaccard_simi
```

“Easy” LSH in R (cont’d)

```r
# plot jaccard similarities that are candidates
qplot(lsh_jaccard$score)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `
```



Putting it all together

The last thing we need is to go from candidate pairs to blocks

```r
library(igraph) #graph package
# think of each record as a node
# there is an edge between nodes if they are candidates
```

Putting it all together

```
g <- make_empty_graph(n, directed = FALSE) # empty graph
g <- add_edges(g, is.vector((candidates[, 1:2]))) # candid
g <- set_vertex_attr(g, "id", value = dat$id) # add id

# get custers, these are the blocks
clust <- components(g, "weak") # get clusters
blocks <- data.frame(id = V(g)$id, # record id
                     block = clust$membership) # block num
head(blocks)
```

```
##   id block
## 1  1     1
## 2  2     2
## 3  3     3
## 4  4     4
## 5  5     5
## 6  6     6
```

Your turn

Using the `fname_c1` and `lname_c1` columns in the `RecordLinkage::RL500` dataset,

1. Use LSH to get candidate pairs for the dataset

- What *k* to use for shingling?
- What *b* to use for bucket size?

2. Append the blocks to the original dataset as a new column, `block`

Even faster?

(**fast**): In minhashing we have to perform *m* permutations to create multiple hashes

(**faster**): We would like to reduce the number of hashes we need to create – “Densified” One Permutation Hashing (DOPH)

- One permutation of the signature matrix is used
- The feature space is then binned into *m* evenly spaced bins
- The *m* minimums (for each bin separately) are the *m* different hash values