Module 5: 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

```r
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
```

LSH

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

Hash function overview

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

Technical reading on this: Chen et al. (2018) and Shrivastava and Steorts (2018)

Hash function

A *hash function* $h()$ is defined such that

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

Hashing shingles

1. Instead of storing the strings as shingles, we store the **hashed values**.
2. These are integers (instead of strings).

We do this because the integers take up less memory, so we are performing a type of **dimension reduction**.

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 (≈ 1.55 minutes) for both approaches, so keep in mind we have not improved this aspect of our approach.

Characteristic matrix (continued)

We can visualize the records (columns) and the hashed shingles in a large, binary **characteristic matrix**

We can think of the fact that we have transformed the input (original data) into the **characteristic matrix**

Characteristic matrix (continued)

```
# 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

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 the *signatures* to **approximate** Jaccard similarity (using a cool fact)

The result is a 3551×1879 matrix

Question: Why is storing the data in this way not a good idea?

Big Idea

Let's apply a permutation to the charateristic matrix:

Input Matrix

0	0	1	1
1	0	0	1
1	1	0	0
0	0	0	1
1	0	0	1

π

4
2
1
3
5

Permuted Matrix

0	0	0	1
1	0	0	1
0	0	1	1
1	1	0	0
1	0	0	1

Figure 1: Consider applying The π vector to the input (charateristic matrix) which provides the permuted matrix.

Why are we performing this permutation?

Now apply the Minhash

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.

Signature Matrix

The resulting signature matrix of the permuted matrix is

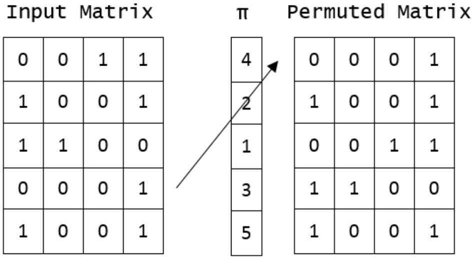


Figure 2: Consider applying The π vector to the input (charateristic matrix) which provides the permuted matrix.

```
signature.matrix <- c(2,4,3,1)
```

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

Signature Matrix

Using the relationship between the Jaccard similarity and the signature matrix, do any records agree? Explain.

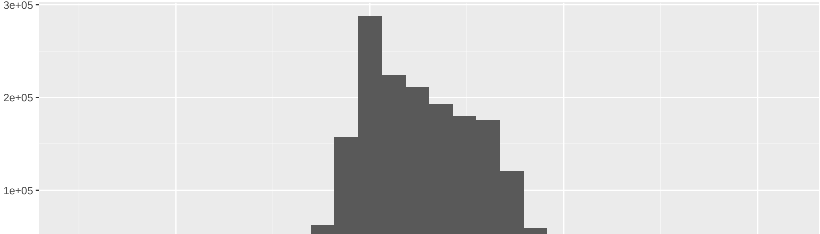
```
# inspect results
kable(sig_mat[1:10, 1:5])
```

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

Jaccard similarity approximation

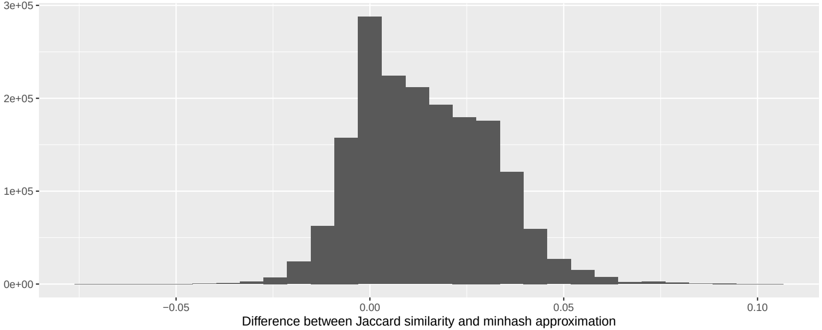
```
# add jaccard similarity approximated from the minhash to compar
# number of agreements over the total number of combinations
hashed_jaccard$similarity_minhash <-
  apply(hashed_jaccard, 1, function(row) {
  sum(sig_mat[, row[["record1"]]]
      == sig_mat[, row[["record2"]]])/nrow(sig_mat)
})

# 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** but we have address the issue of storing things **more efficiently**!

Locality Sensitive Hashing (LSH) to the Rescue

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

1. Divide the **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

```
# view the signature matrix
print(xtable::xtable(sig_mat[1:10, 1:5]),
      hline.after = c(-1,0,5,10), comment=F)
```

	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

Choosing b

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

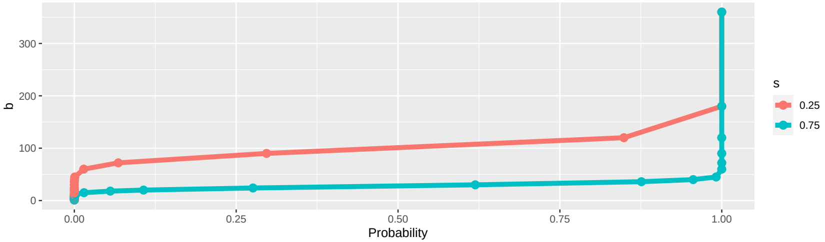


Figure 3: Probability that a pair of documents with a Jaccard similarity s will be marked as potential matches for various bin sizes b for $s = .25, .75$ for the number of permutations we did, $m = 360$.

For $b = 90$, a pair of records with Jaccard similarity .25 will have a 30% chance of being matched as candidates and a pair of records with Jaccard similarity .75 will have a 100% chance of being matched as candidates

“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_shingle
                          progress = FALSE, # quietly
                          keep_tokens = TRUE, # store shingles
                          minhash_func = minhash) # use minhash
```

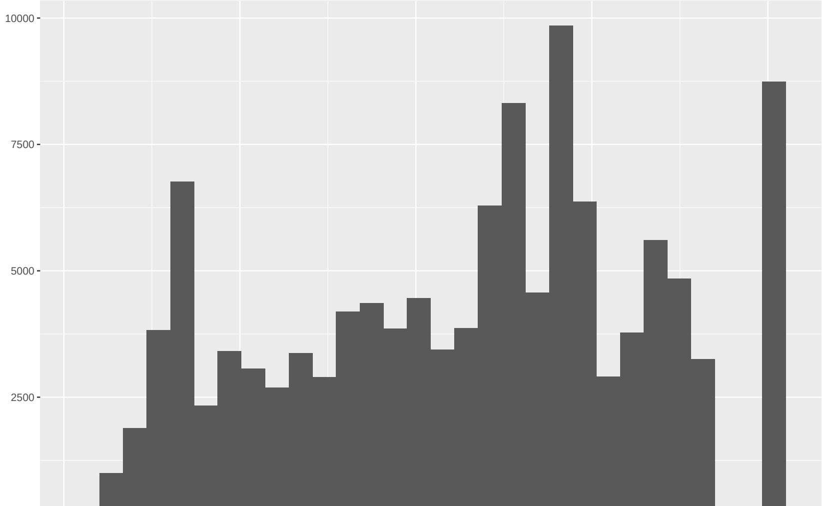
“Easy” LSH in R (Continued)

```
# 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
```

```
##
## Attaching package: 'igraph'

## The following objects are masked from 'package:stats':
##
##     decompose, spectrum

## The following object is masked from 'package:base':
##
##     union
```

```r
# 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]))) # candida
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 numb
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