

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(cora_gold) # contains the cora unique identifiers
```

LSH

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

LSH

- ▶ LSH tries to preserve similarity after dimension reduction.
 - ▶ What kind of similarity? \leftrightarrow What kind of dimension reduction?

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

We are looking for a hash function $h()$ such that

- ▶ if $\text{sim}(A, B)$ is high, then with high prob. $h(A) = h(B)$.
- ▶ if $\text{sim}(A, B)$ is low, then with high prob. $h(A) \neq h(B)$.

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 store the shingles, it takes 8.411816×10^6 bytes.

To hash the shingles, it takes 6.53296×10^5 bytes.

Thus, we will **hash shingles** because it takes up less memory.

The entire pairwise comparison still took the same amount of time (≈ 1.16 minutes) for both approaches, so keep in mind we have not improved this aspect of our approach, but we will improve upon this later!

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[1:10, 1:5])
```

	Record 1	Record 2	Record 3	Record 4	Record 5
466016402	1	1	1	1	1
-105604907	1	1	1	1	1
1270983012	1	1	1	1	1
-1170519627	1	1	1	1	1
2082481628	1	1	1	1	1
60115271	1	1	1	1	1
-883068501	1	1	1	1	1
-1062212181	1	1	1	1	1
-1631420392	1	1	1	1	1
1146728047	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 3562×1879 matrix

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

Big Idea

Let's apply a permutation to the characteristic matrix:

Input Matrix	π	Permuted Matrix																																													
<table><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	1	<table><tr><td>4</td></tr><tr><td>2</td></tr><tr><td>1</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	4	2	1	3	5	<table><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	1	1	0	0	1	0	0	1	1	1	1	0	0	1	0	0	1
0	0	1	1																																												
1	0	0	1																																												
1	1	0	0																																												
0	0	0	1																																												
1	0	0	1																																												
4																																															
2																																															
1																																															
3																																															
5																																															
0	0	0	1																																												
1	0	0	1																																												
0	0	1	1																																												
1	1	0	0																																												
1	0	0	1																																												

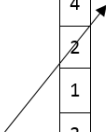


Figure 1: Consider applying The π vector to the input (characteristic matrix) which provides the permuted matrix.

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

Input Matrix	π	Permuted Matrix																																													
<table><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	1	<table><tr><td>4</td></tr><tr><td>2</td></tr><tr><td>1</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	4	2	1	3	5	<table><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	1	1	0	0	1	0	0	1	1	1	1	0	0	1	0	0	1
0	0	1	1																																												
1	0	0	1																																												
1	1	0	0																																												
0	0	0	1																																												
1	0	0	1																																												
4																																															
2																																															
1																																															
3																																															
5																																															
0	0	0	1																																												
1	0	0	1																																												
0	0	1	1																																												
1	1	0	0																																												
1	0	0	1																																												

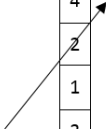


Figure 2: Consider applying The π vector to the input (characteristic matrix) which provides the permuted matrix.

```
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

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

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, 5:10])
```

Record 5	Record 6	Record 7	Record 8	Record 9	Record 10
3	3	3	3	122	21
38	38	38	38	21	13
32	32	32	32	235	107
3	3	3	3	1	111
101	101	101	101	55	4
70	70	70	31	7	18
153	153	153	153	87	53
36	36	36	36	36	19
55	55	55	39	3	68
44	44	44	21	34	34

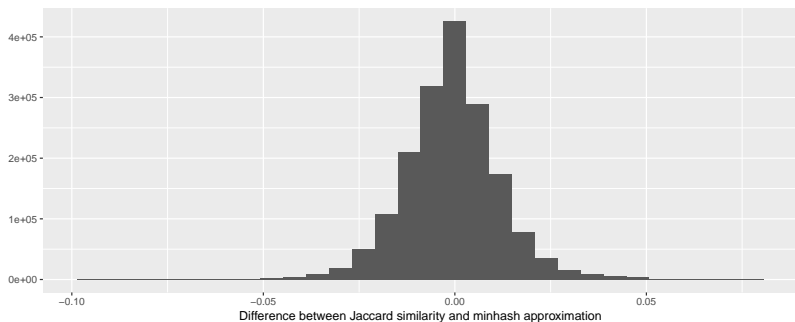
Jaccard similarity approximation

```
# add jaccard similarity approximated from the minhash to compare  
# 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$similarity_minhash - hashed_jaccard$similarity_minhash,  
       xlab("Difference between Jaccard similarity and minhash approximation"))
```

```
## Warning: `qplot()` was deprecated in ggplot2 3.4.0.  
## This warning is displayed once every 8 hours.  
## Call `lifecycle::last_lifecycle_warnings()` to see where this  
## generated.  
  
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`
```

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't addressed the issue of **pairwise comparisons** but we have addressed 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

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

Choosing b

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

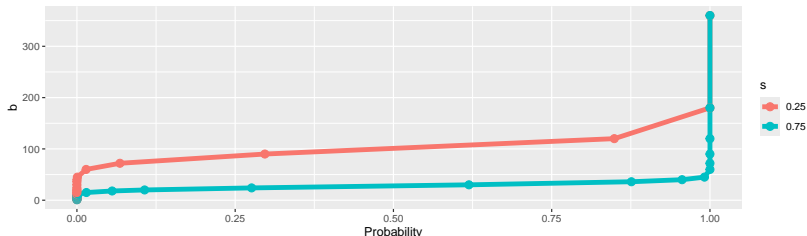


Figure 3: Probability that a pair of documents with a Jaccard similarity s will be marked as potential matches for various bin sizes b for $s = .25, .75$ for the number of permutations we did, $m = 360$.

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):

```
# choose appropriate num of bands  
b <- 90  
  
# create the minhash function  
minhash <- minhash_generator(n = m, seed = 02082018)
```


“Easy” LSH in R (Continued)

```
# 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)

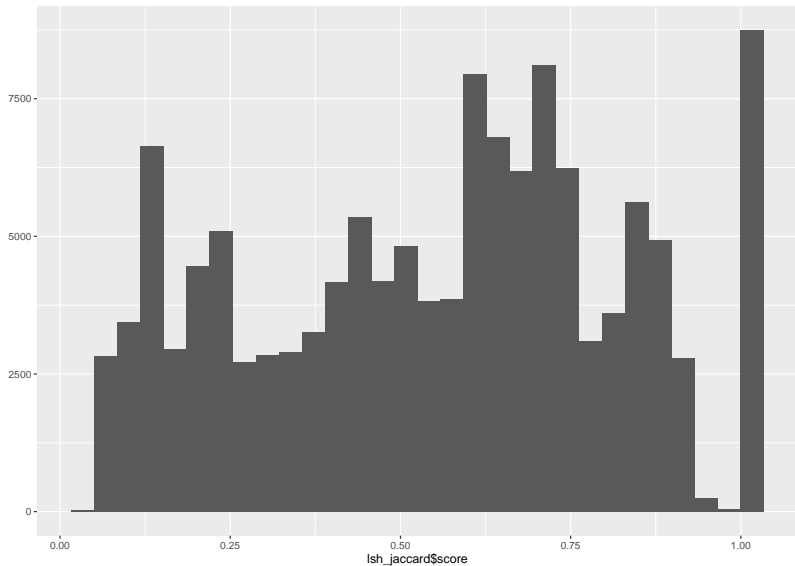
```
# perform lsh to get buckets
buckets <- lsh(corpus, bands = b, progress = FALSE)

# grab candidate pairs
candidates <- lsh_candidates(buckets)

# get Jaccard similarities only for candidates
lsh_jaccard <- lsh_compare(candidates, corpus,
                           jaccard_similarity, progress = TRUE)
```

“Easy” LSH in R (cont'd)

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



Your turn

1. Identify all pairs of blocks.
2. Find the reduction ratio.
3. Write code to find the precision and recall of the blocks.
4. Make sure this is reproducible.

Hint: Why can't you use existing code.

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