Finding Similar Items and Locality Sensitive Hashing

STA 325, Supplemental Material

Andee Kaplan and Rebecca C. Steorts

Background

We first go through a bit of R programming that you have not seen. This will not be a complete review, so please be sure to go through this on your own after the lecture.

Libraries

```
library(knitr)

opts_chunk$set(message=FALSE, warning=FALSE)
```

```
# load libraries
library(rvest) # web scraping
library(stringr) # string manipulation
library(dplyr) # data manipulation
library(tidyr) # tidy data
library(purrr) # functional programming
library(scales) # formatting for rmd output
library(ggplot2) # plots
library(numbers) # divisors function
library(textreuse) # detecting text reuse and
# document similarity
```

Piping

There is a helpful R trick (used in many other languages) known as piping that we will use. Piping is available through R in the `magrittr` package and using the `{%>%}` command.

How does one use the pipe command? The pipe operator is used to insert an argument into a function. The pipe operator takes the left-hand side (LHS) of the pipe and uses it as the first argument of the function on the right-hand side (RHS) of the pipe.

Examples of Piping

We give two simple examples

```
# Example One
1:50 %>% mean
```

```
## [1] 25.5
```

```
mean(1:50)
```

```
## [1] 25.5
```

Examples of Piping

```
# Example Two
years <- factor(2008:2012)
# nesting
as.numeric(as.character(years))
```

```
## [1] 2008 2009 2010 2011 2012
```

```
# piping
years %>% as.character %>% as.numeric
```

```
## [1] 2008 2009 2010 2011 2012
```

The dplyr package

A very useful function that I recommend that you explore and learn about is called dplyr and there is a very good blogpost about its functions: https://www.r-bloggers.com/useful-dplyr-functions-wexamples/

The `dplyr` package

Why should you learn and use `dplyr`?

- It will save you from writing your own complicated functions.
- It's also very useful as it will allow you to perform data cleaning, manipulation, visualisation, and analysis.

The `dplyr` package

There are many useful cheat sheets that have been made and you can find them here reading R and Rmarkdown in general that will help you with this module and optimizing your R code in general: `https://www.rstudio.com/resources/cheatsheets/`

We won't walk through an example as the blogpost above does a very nice job of going through functions within `dplyr`, such as `filter`, `mutate`, etc. Please go through these and your own become familiar with them.

Reading

The reading for this module can be found in Mining Massive Datasets, Chapter 3.

`http://infolab.stanford.edu/~ullman/mmds/book.pdf`

There are no applied examples in the book, however, we will be covering these in class.

Agenda

- Write an agenda of the lecture

Information Retrieval

One of the fundamental problems with having a lot of data is finding what you're looking for.

This is called **information retrieval**!

In this module, we will look at the problem of how to examine data for similar items for very large data sets.

Finding Similar Items

Suppose that we have a collection of webpages and we wish to find near-duplicate pages. (We might be looking for plagarism).

We will introduce how the problem of finding textually similar documents can be turned into a set using a method called shingling.

Then, we will introduce a concept called minhashing, which compresses large sets in such a way that we can deduce the similarity of the underlying sets from their compressed versions.

Finally, we will discuss an important problem that arises when we search for similar items of any kind, there can be too many pairs of items to test each pair for their degree of similarity. This concern motivates locality sensitive hashing (LSH), which focuses our search on pairs that are most likely to be similar.

Overall questions

- How can we use take similar data points and put them in clusters (buckets or bins)?
- How can we evalauate how well our method is doing? (Look at the precision and recall).

Blocking (partitioning)

Blocking partitions of data points so that we do not have to make all-to-all data point comparisons.

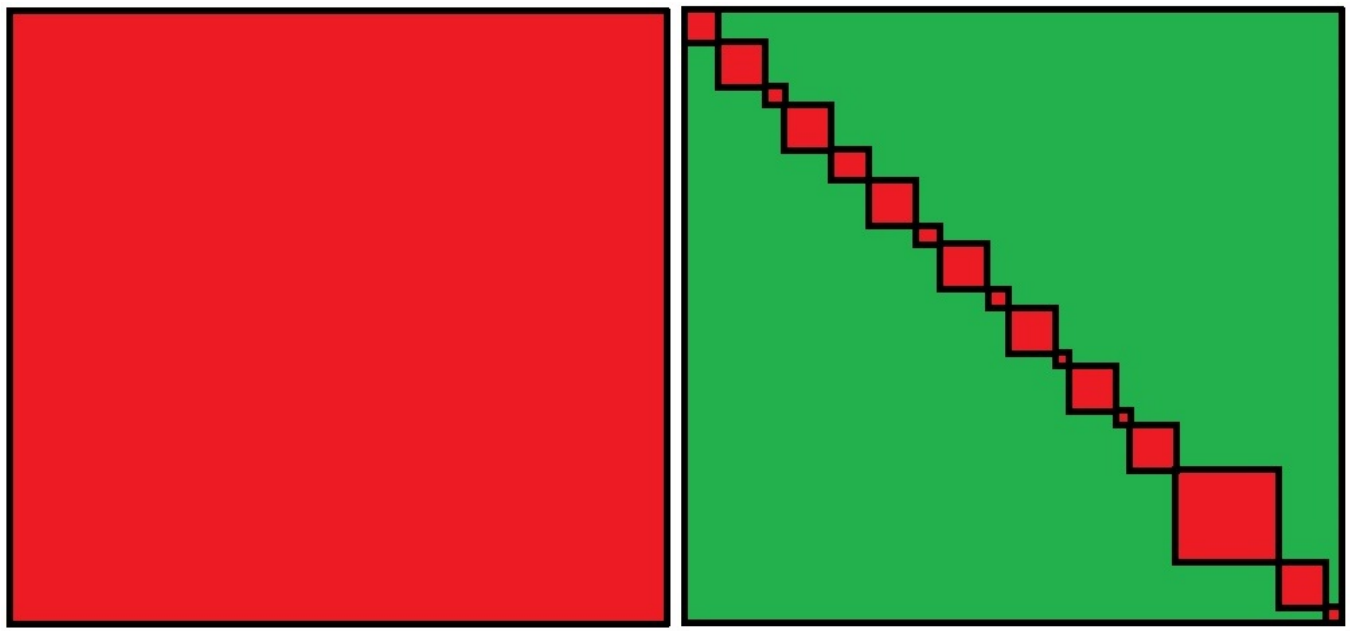


Figure 1: All-to-all data point comparisons (left) versus partitioning data points into blocks/bins (right).

Naive blocking approaches

We could "block" data points deterministically based upon their attribute values and place them into bins.

Example: Suppose we place all data points that have the same DOB year into the same bin.

Would this work very well in practice? Why or why not?

Goal

Our overall goal in this lecture will be able to quickly compare the similarity of the following:

- documents
- songs
- data points
- other examples

In this module, we'll look at an example where we want to figure out given a number of Beatles songs that we have, which songs are the most similar?

Goal

1. We'll learn one way that we can go about doing this — locality senstive hashing.
2. This method extends to comparing documents, records, etc.
3. We'll learn the basics beyond the method and why it works.
4. Then we'll apply it to the Beatles' song application.
5. Finally, we'll look at a computational speed up to see how to improve this method in practice.
6. Finally, you'll look at a similar problem for homework so you can get practice on your own.

Hash function

Find 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. $h(A) \neq h(B)$.

Locality sensitive hashing (LSH)

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

Background

Before going through LSH, we first introduce

1. Jaccard similarity
2. shingling

which are both essential to performing LSH.

Notions of Similarity

We have already discussed many notions of similarity, however, the main one that we work with in this module is the Jaccard similarity.

The Jaccard similarity of set S and T is

$$\frac{|S \cap T|}{|S \cup T|}$$

.

We will refer to the Jaccard similarity of S and T by $\mathsf{SIM}(S, T)$.

Jaccard Similarity

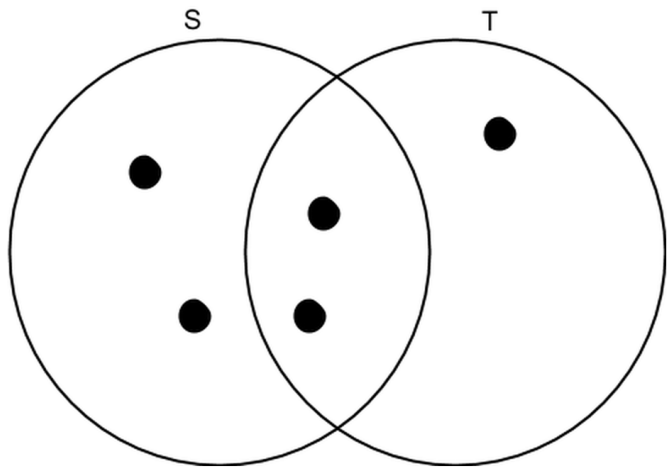


Figure 2: Two sets S and T with Jaccard similarity 2/5. The two sets share two elements in common, and there are five elements in total.

Overview of LSH

1. We first shingle a document (records, songs, etc), which breaks the document up into small subsets.
2. Next, we form a characteristic matrix using the shingled document. A characteristic matrix is an indicator matrix, where the columns of the matrix correspond to records (data points) in the document (so there is a column for each record) and the rows correspond to the shingles (so there is a row for each shingle).
3. The characteristic matrix is a large binary matrix. Next, we apply a permutation π to it to create a signature matrix which allows for some dimension reduction.
4. Based on the results of the signature matrix, we will observe the pairwise similarity of two records (data points). For the number of hash functions, there is an interesting relationship that we use between the columns for any given set of the signature matrix and a Jaccard similarity measure that we will eventually discover.

LSH (continued)

- Now that we have the high-level idea of LSH, let's look at it in more detail.

Remark: This is not the fastest approach for applying LSH, however, this will help us gain a deeper understanding of LSH before learning about faster variants.

Shingling of Documents

The most effective way to represent document as sets is to construct from the document the set of short strings that appear within it.

Then the documents that share pieces as short strings will have many common elements.

Here, we introduce the most simple and common approach — shingling.

k-Shingles

A document can be represented as a string of characters.

We define a k-shingle (k-gram) for a document to be any sub-string of length k found within the document.

Then we can associate with each document the set of k-shingles that appear one or more times within the document.

k-Shingles

Example: Suppose our document is the string `abcdabd` and $k = 2$.

The set of 2-shingles is $\{ab, bc, cd, da, bd\}$.

Note the sub-string `ab` appears twice within the document but only once as a shingle.

Note it also makes sense to replace any sequence of one or more white space characters by a single blank. We can then distinguish shingles that cover two or more words from those that do not.

Jaccard similarity of Beatles songs

Now that we have shingled records (sets), we can now look at the Jaccard similarity for each song by looking at the shingles for each song as a set (S_i) and computing the Jaccard similarity

$$\frac{|S_i \cap S_j|}{|S_i \cup S_j|} \text{ for } i \neq j.$$

We will use `textreuse::jaccard_similarity()` to perform these computations.

Data

We will scrape lyrics from `http://www.metrolyrics.com`. (Please note that I do not expect you to be able to scrape the data, but the code is on the next two slides in case you are interested in it.)

```
# get beatles lyrics
links <- read_html("http://www.metrolyrics.com/beatles-lyrics.html") %>% # lyrics site
  html_nodes("td a") # get all links to all songs

# get all the links to song lyrics
tibble(name = links %>% html_text(trim = TRUE) %>% str_replace(" Lyrics", ""), # get song names
       url = links %>% html_attr("href")) -> songs # get links

# function to extract lyric text from individual sites
get_lyrics <- function(url){
  test <- try(url %>% read_html(), silent=T)
  if ("try-error" %in% class(test)) {
    # if you can't go to the link, return NA
    return(NA)
  } else
    url %>% read_html() %>%
      html_nodes(".verse") %>% # this is the text
      html_text() -> words

    words %>%
    paste(collapse = ' ') %>% # paste all paragraphs together as one string
      str_replace_all("[\r\n]" , ". ") %>% # remove newline chars
      return()
}
```

Data (continued)

```
# get all the lyrics
# remove duplicates and broken links
songs %>%
  mutate(lyrics = (map_chr(url, get_lyrics))) %>%
  filter(nchar(lyrics) > 0) %>% #remove broken links
  group_by(name) %>%
  mutate(num_copy = n()) %>%
  # remove exact duplicates (by name)
  filter(num_copy == 1) %>%
  select(-num_copy) -> songs
```

We end up with the lyrics to 62 Beatles songs and we can start to think about how to first represent the songs as collections of short strings (*shingling*). As an example, let's shingle the best Beatles' song of all time, "Eleanor Rigby".

Back to the Beatles

- How would we create a set of k-shingles for the song "Eleanor Rigby"?
- We want to construct from the lyrics the set of short strings that appear within it.
- Then the songs that share pieces as short strings will have many common elements.
- We will get the k-shingles (sub-string of length k words found within the document) for "Eleanor Rigby" using the function `textreuse::tokenize_ngrams()`.

Shingling Eleanor Rigby

```
# get k = 5 shingles for "Eleanor Rigby"
# this song is the best
# filter finds when cases are true
best_song <- songs %>% filter(name == "Eleanor Rigby")
# shingle the lyrics
shingles <- tokenize_ngrams(best_song$lyrics, n = 5)

# inspect results
head(shingles) %>%
  kable() # pretty table
```

ah look at all the
look at all the lonely
at all the lonely people
all the lonely people ah
the lonely people ah look
lonely people ah look at

Shingling Eleanor Rigby

```
# get k = 3 shingles for "Eleanor Rigby"
shingles <- tokenize_ngrams(best_song$lyrics, n = 3)

# inspect results
head(shingles) %>%
  kable()
```

ah look at
look at all
at all the
all the lonely
the lonely people
lonely people ah

Shingling Eleanor Rigby

- We looked at $k = 3$ and $k = 5$ but what should k be?
- How large k should be depends on how long typical songs are.
- The important thing to remember is k should be picked large enough such that the probability of any given shingle in the document is low.
- For simplicity, I will assume $k = 3$.

(For more on how to optimally pick k, see `https://arxiv.org/abs/1510.07714`)

Shingling all Beatles songs

We can now apply the shingling to all 62 Beatles songs in our corpus.

```
# add shingles to each song
# map applies a function to each elt, returns vector
# mutate adds new variables and preserves existing
songs %>%
  mutate(shingles = (map(lyrics, tokenize_ngrams, n = 3)))
```

Shingling all Beatles songs

Again, you should think about the optimal choice of k here.

How would you do this? This is an exercise to think about on your own. (Hints: Can you actually evaluate k here. Think about the recall. Can you compute the recall? Why or why not?)

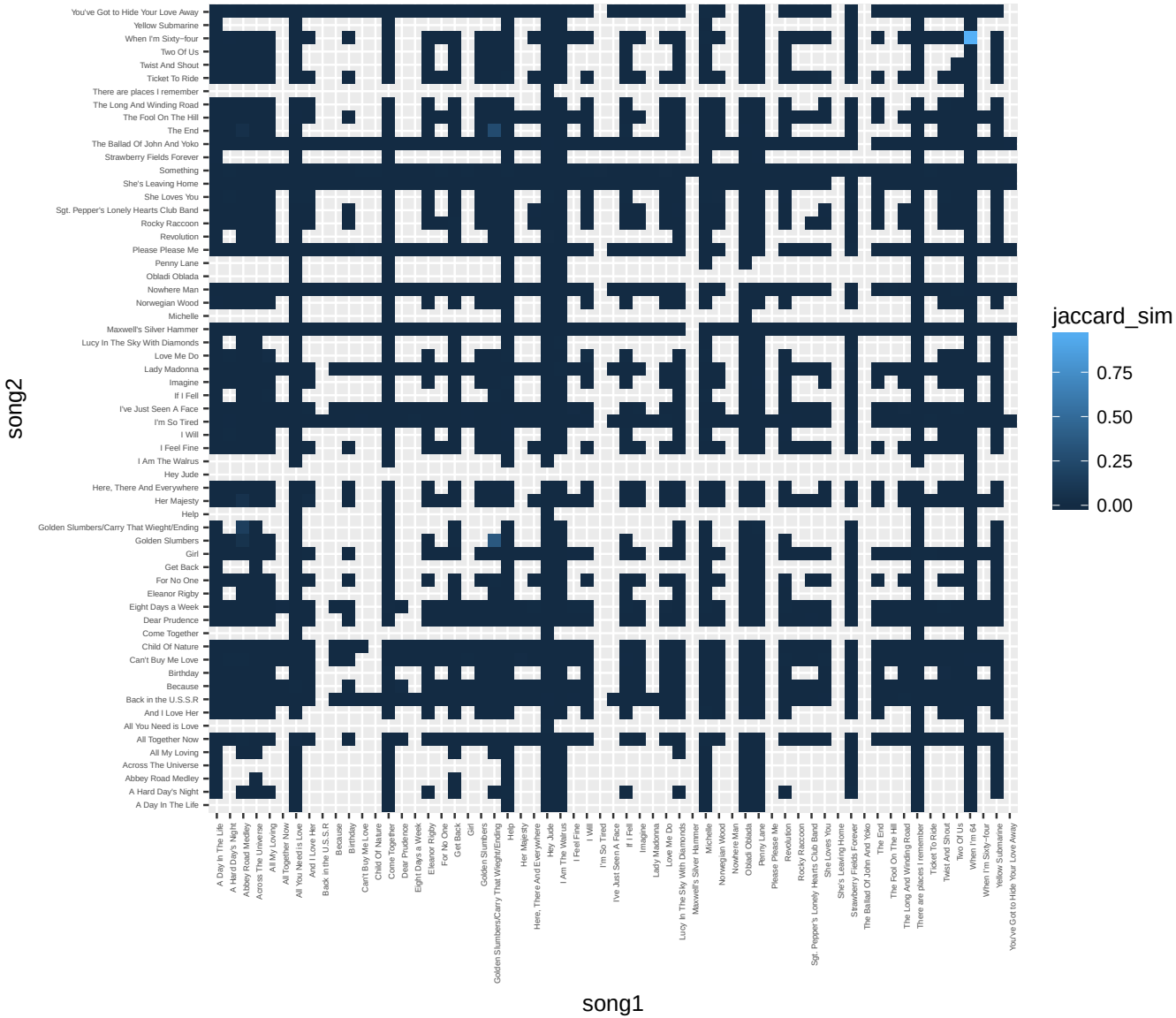
Jaccard similarity of Beatles songs

```
# create all pairs to compare then get the jacard similarity of each
# start by first getting all possible combinations
song_sim <- expand.grid(song1 = seq_len(nrow(songs)), song2 = seq_len(nrow(songs))) %>%
  filter(song1 < song2) %>% # don't need to compare the same things twice
  group_by(song1, song2) %>% # converts to a grouped table
  mutate(jaccard_sim = jaccard_similarity(songs$shingles[song1][[1]],
                                          songs$shingles[song2][[1]])) %>%
  ungroup() %>%  # Undoes the grouping
  mutate(song1 = songs$name[song1],
         song2 = songs$name[song2]) # store the names, not "id"

# inspect results
summary(song_sim)
```

```
##     song1              song2            jaccard_sim
##  Length:1891        Length:1891        Min.   :0.000000
##  Class :character   Class :character   1st Qu.:0.000000
##  Mode  :character   Mode  :character   Median :0.000000
##                                        Mean   :0.001582
##                                        3rd Qu.:0.000000
##                                        Max.   :0.982353
```

Jaccard similarity of Beatles songs



It looks like there are a couple song pairings with very high Jaccard similarity. Let's filter to find out which ones.

Jaccard similarity of Beatles songs

```
# filter high similarity pairs
song_sim %>%
  filter(jaccard_sim > .5) %>% # only look at those with similarity > .5
  kable() # pretty table
```

song1	song2	jaccard_sim
When I'm 64	When I'm Sixty-four	0.9823529

Two of the three matches seem to be from duplicate songs in the data, although it's interesting that their Jaccard similarity coefficients are not equal to 1. (This is not suprising. Why?). Perhaps these are different versions of the song.

Jaccard similarity of Beatles songs

```
# inspect lyrics
print(songs$lyrics[songs$name == "In My Life"])
```

```
## character(0)
```

```
print(songs$lyrics[songs$name == "There are places I remember"])
```

```
## [1] "There are places I remember all my life. Though some have changed. Some forever, not for better. S
```

By looking at the seemingly different pair of songs lyrics, we can see that these are actually the same song as well, but with slightly different transcription of the lyrics. Thus we have identified three songs that have duplicates in our database by using the Jaccard similarity. What about non-duplicates? Of these, which is the most similar pair of songs?

Jaccard dissimilarity of Beatles songs

```
# filter to find the songs that aren't very similar
# These are songs that we don't want to look at
song_sim %>%
  filter(jaccard_sim <= .5) %>%
  arrange(desc(jaccard_sim)) %>%  # sort by similarity
  head() %>%
  kable()
```

song1	song2	jaccard_sim
Golden Slumbers/Carry That Wieght/Ending	Golden Slumbers	0.3557692
Golden Slumbers/Carry That Wieght/Ending	The End	0.2616822
Abbey Road Medley	Golden Slumbers/Carry That Wieght/Ending	0.1583333
Abbey Road Medley	Golden Slumbers	0.0626058
Abbey Road Medley	Her Majesty	0.0612583
Abbey Road Medley	The End	0.0418760

These appear to not be the same songs, and hence, we don't want to look at these. Typically our goal is to look at items that are most similar and discard items that aren't similar.

Jaccard similarity of Beatles songs

Note, although we computed these pairwise comparisons manually, there is a set of functions that could have helped — `textreuse::TextReuseCorpus()`, `textreuse::pairwise_compare()` and `textreuse::pairwise_candidates()`.

Think about how you would do this and investigate this on your own (Exercise).

Solution to Exercise (Go over on your own)

```
# build the corpus using textreuse
docs <- songs$lyrics
names(docs) <- songs$name # named vector for document ids
corpus <- TextReuseCorpus(text = docs,
                          tokenizer = tokenize_ngrams, n = 3,
                          progress = FALSE,
                          keep_tokens = TRUE)
```

Solution to Exercise (Go over on your own)

```
# create the comparisons
comparisons <- pairwise_compare(corpus, jaccard_similarity, progress = FALSE)
comparisons[1:3, 1:3]
```

```
##                                When I'm 64 Hey Jude
## When I'm 64                             NA        0
## Hey Jude                                NA       NA
## There are places I remember             NA       NA
##                                When I'm 64 Hey Jude
## When I'm 64                                       0
## Hey Jude                                          0
## There are places I remember                      NA
##                                There are places I remember
## When I'm 64                                               0
## Hey Jude                                                  0
## There are places I remember                              NA
```

Solution to Exercise (Go over on your own)

```
# look at only those comparisons with Jaccard similarity > .1
candidates <- pairwise_candidates(comparisons)
candidates[candidates$score > 0.1, ] %>% kable()
```

a	b	score
Abbey Road Medley	Golden Slumbers/Carry That Wieght/Ending	0.1583333
Golden Slumbers	Golden Slumbers/Carry That Wieght/Ending	0.3557692
Golden Slumbers/Carry That Wieght/Ending	The End	0.2616822
When I'm 64	When I'm Sixty-four	0.9823529

Jaccard similarity of Beatles songs

For a corpus of size n, the number of comparisons we must compute is $\frac{n^2-n}{2}$, so for our set of songs, we needed to compute $1,891$ comparisons. This was doable for such a small set of documents, but if we were to perform this on the full number of Beatles songs (409 songs recorded), we would need to do $83,436$ comparisons. As you can see, for a corpus of any real size this becomes infeasible quickly. A better approach for corpora of any realistic size is to use *hashing*.

The `textreuse::TextReuseCorpus()` function hashes our shingled lyrics automatically using function that hashes a string to an integer. We can look at these hashes for "Eleanor Rigby".

Before we do this, let's talk about what hashing is and why it's useful!

Hashing

- Traditionally in computer science, a hash function will map objects to integers such that similar objects are far apart.
- We're going to look at special hash functions that do the opposite of this.
- Specifically, we're going to look at hash functions where similar objects are placed closed together!

Formally these hash functions $h()$ are defined such that

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

Hashing shingles

Instead of using substrings as shingles, we can use a hash function, which maps strings of length k to some number of buckets and treats the resulting bucket number as the shingle.

The set representing a document is then the set of integers of one or more k-shingles that appear in the document.

Example: We could construct the set of 9-shingles for a document. We could then map the set of 9-shingles to a bucket number in the range 0 to $2^{32} - 1$.

Thus, each shingle is represented by four bytes intead of 9. The data has been compressed and we can now manipulate (hashed) shingles by single-word machine operations.

Similarity Preserving Summaries of Sets

Sets of shingles are large. Even if we hash them to four bytes each, the space needed to store a set is still roughly four times the space taken up by the document.

If we have millions of documents, it may not be possible to store all the shingle-sets in memory. (There is another more serious problem, which we will address later in the module.)

Here, we replace large sets by smaller representations, called signatures. The important property we need for signatures is that we can compare the signatures of two sets and estimate the Jaccard similarity of the underlying sets from the signatures alone.

It is not possible for the signatures to give the exact similarity of the sets they represent, but the estimates they provide are close.

Characteristic Matrix

Before describing how to construct small signatures from large sets, we visualize a collection of sets as a characteristic matrix.

The columns correspond to sets and the rows correspond to the universal set from which the elements are drawn.

There is a 1 in row r, column c if the element for row r is a member of the set for column c. Otherwise, the value for (r, c) is 0.

Below is an example of a characteristic matrix, with four shingles and five records.

```
library("pander")
element <- c("a","b","c","d","e")
S1 <- c(0,1,1,0,1)
S2 <- c(0,0,1,0,0)
S3 <- c(1,0,0,0,0)
S4 <- c(1,1,0,1,1)
my.data <- cbind(element,S1,S2,S3,S4)
```

Characteristic Matrix

```
pandoc.table(my.data,style="rmarkdown")
```

```
##
##
## | element | S1 | S2 | S3 | S4 |
## |:-------:|:--:|:--:|:--:|:--:|
## |    a    | 0  | 0  | 1  | 1  |
## |    b    | 1  | 0  | 0  | 1  |
## |    c    | 1  | 1  | 0  | 0  |
## |    d    | 0  | 0  | 0  | 1  |
## |    e    | 1  | 0  | 0  | 1  |
```

Why would we not store the data as a characteristic matrix (think sparsity)?

Minhashing

The signatures we desire to construct for sets are composed of the results of a large number of calculations, say several hundred, each of which is a "minhash" of the characteristic matrix.

Here, we will learn how to compute the minhash in principle. Later, we will learn how a good approximation is computed in practice.

To minhash a set represented by a column of the characteristic matrix, pick a permutation of the rows.

The minhash value of any column is the index of the first row in the permuted order in which the column has a 1.

Minhashing

Now, we permute the rows of the characteristic matrix to form a permuted matrix.

The permuted matrix is simply a reordering of the original characteristic matrix, with the rows swapped in some arrangement.

Figure 2 shows the characteristic matrix converted to a permuted matrix by a given permutation. We repeat the permutation step for several iterations to obtain multiple permuted matrices.

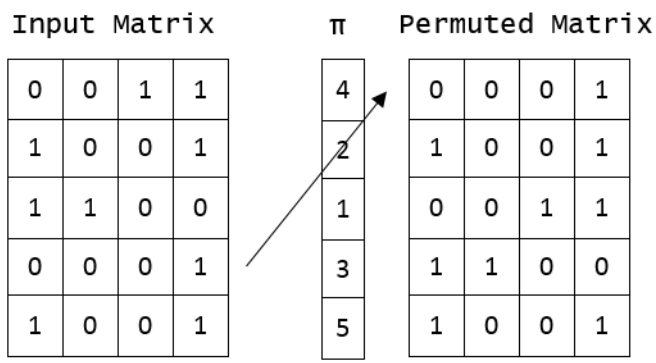


Figure 3: Permuted matrix from the characteristic one. The π vector is the specified permutation.

The Signature Matrix

Now, we compute the signature matrix.

The signature matrix is a hashing of values from the permuted one.

The signature has a row for the number of permutations calculated, and a column corresponding with the columns of the permuted matrix.

We iterate over each column of the permuted matrix, and populate the signature matrix, row-wise, with the row index from the first 1 value found in the column.

Signature Matrix

```
library("pander")
signature <- c(2,4,3,1)
pandoc.table(signature,style="rmarkdown")
```

```
##
## |   |   |   |   |
## |:-:|:-:|:-:|:-:|
## | 2 | 4 | 3 | 1 |
```

Minhashing and Jaccard Similarity

Based on the results of the signature matrix, we observe the pairwise similarity of two records. For the number of hash functions, there is an interesting relationship that we use between the columns for any given set of the signature matrix and a Jaccard Similarity measure.

Minhashing and Jaccard Similarity

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

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

The equation means that the probability that the minimum values of the given hash function, in this case h, is the same for sets A and B is equivalent to the Jaccard Similarity, especially as the number of record comparisons increases.

We use this relationship to calculate the similarity between any two records.

We look down each column, and compare it to any other column: the number of agreements over the total number of combinations is equal to Jaccard measure.

Back to the Beatles

Recall that `textreuse::TextReuseCorpus()` function hashes our shingled lyrics automatically using function that hashes a string to an integer. We can look at these hashes for "Eleanor Rigby".

```
# look at the hashed shingles for "Eleanor Rigby"
tokens(corpus)$`Eleanor Rigby` %>% head()
```

```
## [1] "ah look at"         "look at all"        "at all the"
## [4] "all the lonely"     "the lonely people" "lonely peop
```

```
hashes(corpus)$`Eleanor Rigby` %>% head()
```

```
## [1] -1188528426  1212580251  2071766612  -964323450  -61
```

Back to the Beatles

We can alternatively specify the hash function by hand as well using the function `textreuse::hash_string()`.

```
# manually hash shingles
songs %>%
  mutate(hash = (map(shingles, hash_string))) -> songs

# note by using the "map" function, we have a list column
# for details, see purrr documentation

songs$hash[songs$name == "Eleanor Rigby"][[1]] %>% head()
```

```
## [1] -1188528426  1212580251  2071766612  -964323450  -61
```

Back to the Beatles

Now instead of storing the strings (shingles), we can just store the hashed values. Since these are integers, they will take less space which will be useful if we have large documents. Instead of performing the pairwise Jaccard similarities on the strings, we can perform them on the hashes.

```
# compute jaccard similarity on hashes instead of shingled lyrics
# add this column to our song data.frame
song_sim %>%
  group_by(song1, song2) %>%
  mutate(jaccard_sim_hash =
           jaccard_similarity(songs$hash[songs$name == song1][[1]],
                                          songs$hash[songs$name == song2][[1]])) -> song_sim

# how many songs do the jaccard similarity computed from hashing
# versus the actual shingles NOT match?
sum(song_sim$jaccard_sim != song_sim$jaccard_sim_hash)
```

```
## [1] 0
```

Back to the Beatles

- We can start by visualizing our collection of sets as a characteristic matrix where the columns correspond to songs and the rows correspond to all of the possible shingled elements in all songs.
- There is a 1 in row r, column c if the shingled phrase for row r is in the song corresponding to column c.
- Otherwise, the value for (r, c) is 0.

Characterisitic Matrix for the Beatles

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

# get the characteristic matrix
# items are all the unique hash values
# lists will be each song
# we want to keep track of where each hash is included
data.frame(item = unique(unlist(songs$hash))) %>%
  group_by(item) %>%
  mutate(list = list(songs$name)) %>% # include the song name for each item
  unnest(list) %>% # expand the list column out
  # record if item in list
  mutate(included = (map2_int(item, songs$hash[songs$name == list], item_in_list))) %>%
  spread(list, included) -> char_matrix # tall data -> wide data (see tidyr documentation)

# inspect results
char_matrix[1:4, 1:4] %>%
  kable()
```

item	A Day In The Life	A Hard Day's Night	Abbey Road Medley
-2147299362	0	0	0
-2147099044	0	0	0
-2146305897	0	0	0
-2143657845	0	0	0

Characterisitic Matrix for the Beatles

As you can see (and imagine), this matrix tends to be sparse because the set of unique shingles across all documents will be fairly large. Instead we want create the signature matrix through minhashing which will give us an approximation to measure the similarity between song lyrics.

Signature Matrix for the Beatles

To compute the signature matrix, we first permute the characteristic matrix and then iterate over each column of the permuted matrix and 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 one and has a row for the number of permutations calculated, and a column corresponding with the columns of the permuted matrix. We can show this for the first permutation.

Signature Matrix for the Beatles

```
# set seed for reproducibility
set.seed(09142017)

# get permutation order
permute_order <- sample(seq_len(nrow(char_matrix)))

# get min location of "1" for each column (apply(2, ...))
sig_matrix <- char_matrix[permute_order, -1] %>%
  apply(2, function(col) min(which(col == 1))) %>%
  as.matrix() %>% t()

# inspect results
sig_matrix[1, 1:4] %>% kable()
```

A Day In The Life	4
A Hard Day's Night	59
Abbey Road Medley	1
Across The Universe	31

Signature Matrix for the Beatles

We can now repeat this process many times to get the full signature matrix.

```
# function to get signature for 1 permutation
get_sig <- function(char_matrix) {
  # get permutation order
  permute_order <- sample(seq_len(nrow(char_matrix)))

  # get min location of "1" for each column (apply(2, ...))
  char_matrix[permute_order, -1] %>%
    apply(2, function(col) min(which(col == 1))) %>%
    as.matrix() %>% t()
}

# repeat many times
m <- 360
for(i in 1:(m - 1)) {
  sig_matrix <- rbind(sig_matrix, get_sig(char_matrix))
}

# inspect results
sig_matrix[1:4, 1:4] %>% kable()
```

A Day In The Life	A Hard Day's Night	Abbey Road Medley	Across The Universe
4	59	1	31
22	25	17	33
18	40	2	17
88	2	10	63

Signature Matrix, Jaccard Similarity, and the Beatles

Since we know 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 can use this relationship to calculate 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, Jaccard Similarity, and the Beatles

```
# add jaccard similarity approximated from the minhash to compare
# number of agreements over the total number of combinations
song_sim <- song_sim %>%
  group_by(song1, song2) %>%
  mutate(jaccard_sim_minhash =
           sum(sig_matrix[, song1] == sig_matrix[, song2])/nrow(sig_matrix))

# how far off is this approximation?
summary(abs(song_sim$jaccard_sim_minhash - song_sim$jaccard_sim))
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0000000 0.0000000 0.0000000 0.0004128 0.0000000 0.0418138
```

Evaluating Performance

Suppose that we wish to evaluate the performance of how well our method works in practice.

We would require some training data (hand-matched data) on the Beatles songs such that we could compute the recall and precision.

Evaluating Performance

Our first step would be either accessing or creating pairs of songs and noting whether these songs are a match or a non-match.

We would want to store this into a data set so that we could use it later if needed (and would not need to repeat our results).

How would you go about doing this?

Evaluating Performance

Once you have a handmatched data set, we can calculate the recall and the precision.

How do we calculate the recall and precision?

We need to learn some new terminology first and then we're all set!

Classifications

1. Pairs of data can be linked in both the handmatched training data (which we refer to as 'truth') and under the estimated linked data. We refer to this situation as true positives (TP).
2. Pairs of data can be linked under the truth but not linked under the estimate, which are called false negatives (FN).
3. Pairs of data can be not linked under the truth but linked under the estimate, which are called false positives (FP).
4. Pairs of data can be not linked under the truth and also not linked under the estimate, which we refer to as true negatives (TN).

Then the true number of links is TP + FN, while the estimated number of links is TP + FP. The usual definitions of false negative rate and false positive rate are

$$\text{FNR} = \text{FN}/(\text{TP}+\text{FN})$$

$$\text{FPR} = \text{FP}/(\text{TP}+\text{FP})$$

Classifications

$$\text{FNR} = \text{FN}/(\text{TP}+\text{FN})$$

$$\text{FPR} = \text{FP}/(\text{TP}+\text{FP})$$

Then

$$\text{recall} = 1 - \text{FNR}.$$

$$\text{precision} = 1 - \text{FPR}.$$

How would you look at the sensitivity of a method for the recall andd the precision?

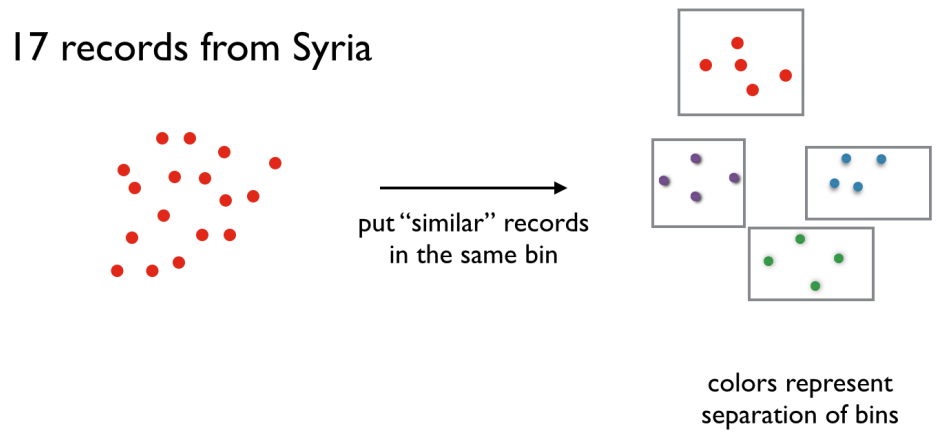
Signature Matrix, Jaccard Similarity, and the Beatles

While we have 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), we still haven't addressed the issue of pairwise comparisons.

This is where *locality-sensitive hashing (LSH)* can help.

General idea of LSH

Our general approach to LSH is to hash items several times such that similar items are more likely to be hashed into the same bucket (bin).



Candidate Pairs

Any pair that is hashed to the same bucket for any hashings is called a candidate pair.

We only check candidate pairs for similarity.

The hope is that most of the dissimilar pairs will never hash to the same bucket (and never be checked).

False Negative and False Positives

Dissimilar pairs that are hashed to the same bucket are called false positives.

We hope that most of the truly similar pairs will hash to the same bucket under at least one of the hash functions.

Those that don't are called false negatives.

How to choose the hashings

Suppose we have minhash signatures for the items.

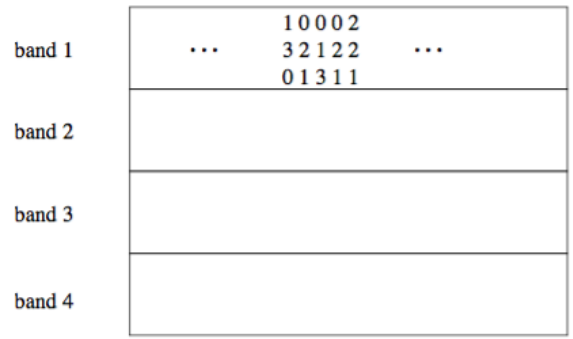
Then an effective way to choose the hashings is to divide up the signature matrix into b bands with r rows.

For each band b, there is a hash function that takes vectors of r integers (the portion of one column within that band) and hashes them to some large number of buckets.

We can use the same hash function for all the bands, but we use a separate bucket array for each band, so columns with the same vector in different bands will hash to the same bucket.

Example

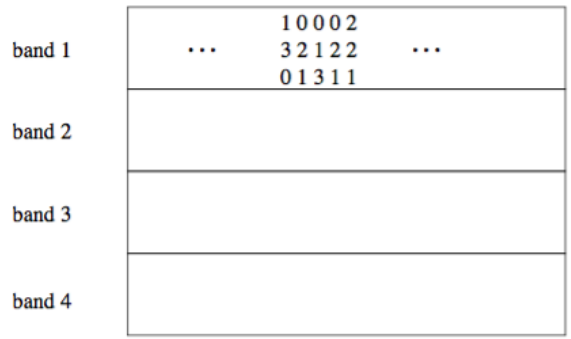
Consider the signature matrix



The second and fourth columns each have a column vector $[0, 2, 1]$ in the first band, so they will be mapped to the same bucket in the hashing for the first band.

Regardless of the other columns in the other bands, this pair of columns will be a candidate pair.

Example (continued)



It's possible that other columns such as the first two will hash to the same bucket according to the hashing in the first band.

But their column vectors are different, $[1, 3, 0]$ and $[0, 2, 1]$ and there are many buckets for each hashing, so an accidential collision here is thought to be small.

Analysis of the banding technique

You can learn some about the analysis of the banding technique on your own in Mining Massive Datasets, Ch 3, 3.4.2.

Combining the Techniques

We can now give an approach to find the set of candidate pairs for similar documents and then find similar documents among these.

1. Pick a value k and construct from each document the set of k-shingles. (Optionally, hash the k-shingles to shorter bucket numbers).
2. Sort the document-shingle pairs to order them by shingle.
3. Pick a length n for the minhash signatures. Feed the sorted list to the algorithm to compute the minhash signatures for all the documents.
4. Choose a threshold t that defines how similar doucments have to be in order to be regarded a similar pair. Pick a number b bands and r rows such that $br = n$ and then the threshold is approximately $(1/b)^{1/r}$.
5. Construct candidate pairs by applying LSH.
6. Examine each candidate pair's signatures and determine whether the fraction of components they agree is at least t.
7. Optionally, if the signatures are sufficiently similar, go to the documents themselves and check that they are truly similar.

LSH for the Beatles

- Recall, performing pairwise comparisons in a corpus is time-consuming because the number of comparisons grows at $O(n^2)$.
- Most of those comparisons, furthermore, are unnecessary because they do not result in matches due to sparsity.
- We will use the combination of minhash and locality-sensitive hashing (LSH) to solve these problems. They make it possible to compute possible matches only once for each document, so that the cost of computation grows linearly.

LSH for the Beatles

We want to hash items several times such that similar items are more likely to be hashed into the same bucket. Any pair that is hashed to the same bucket for any hashing is called a candidate pair and we only check candidate pairs for similarity.

1. Divide up the signature matrix into b bands with r rows such that $m = b * r$ where m is the number of times that we drew a permutation of the characteristic matrix in the process of minhashing (number of rows in the signature matrix).
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. Each pair of documents has as many chances to be considered a candidate as there are bands, and the fewer rows there are in each band, the more likely it is that each document will match another.

Choosing b?

- The question becomes, how to choose b, the number of bands?
- We now it must divide m evenly such that there are the same number of rows r in each band, but beyond that we need more guidance.
- We can estimate the probability that a pair of documents with a Jaccard similarity s will be marked as potential matches using the `textresuse::lsh_probability()` function.
- This is a function of m, h, and s,

  $P($pair of documents with a Jaccard similarity s will be marked as po

  We can then look as this function for different numbers of bands b and difference Jaccard similarities s.

LSH for the Beatles

```
# look at probability of binned together for various bin sizes and similarity values
tibble(s = c(.25, .75), h = m) %>% # look at two different similarity values
  mutate(b = (map(h, divisors))) %>% # find possible bin sizes for m
  unnest() %>% # expand dataframe
  group_by(h, b, s) %>%
  mutate(prob = lsh_probability(h, b, s)) %>%
  ungroup() -> bin_probs # get probabilities

# plot as curves
bin_probs %>%
  mutate(s = factor(s)) %>%
  ggplot() +
  geom_line(aes(x = prob, y = b, colour = s, group = s)) +
  geom_point(aes(x = prob, y = b, colour = factor(s)))
```

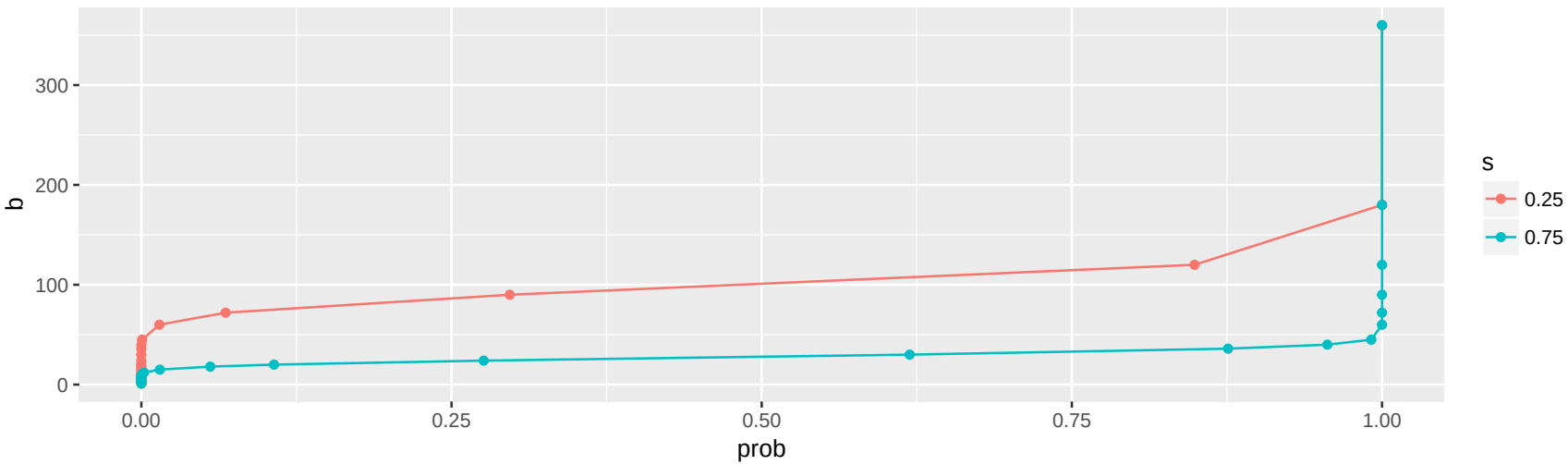


Figure 4: Probability that a pair of documents with a Jaccard similarity s will be marked as potential matches for various bin sizes b for $s = .25, .75$ for the number of permutations we did, $m = 360$.

LSH for the Beatles

We can then look at some candidate numbers of bins b and make our choice.

```
# look as some candidate b
bin_probs %>%
  spread(s, prob) %>%
  select(-h) %>%
  filter(b > 50 & b < 200) %>%
  kable()
```

b	0.25	0.75
60	0.0145434	0.9999922
72	0.0679295	1.0000000
90	0.2968963	1.0000000
120	0.8488984	1.0000000
180	0.9999910	1.0000000

For $b = 90$, a pair of records with Jaccard similarity of .25 will have a 29.7% chance of being matched as candidates and a pair of records with Jaccard similarity of .75 will have a 100.0% chance of being matched as candidates. This seems adequate for our application, so I will continue with $b = 90$.

Why would we not consider $b = 120$? What happens if you look at $b = 72$ and $b = 60$ (do this on your own).

LSH for the Beatles

Now that we now how to bin our signature matrix, we can go ahead and get the candidates.

```
# bin the signature matrix
b <- 90
sig_matrix %>%
  as_tibble() %>%
  mutate(bin = rep(1:b, each = m/b)) %>% # add bins
  gather(song, hash, -bin) %>% # tall data instead of wide
  group_by(bin, song) %>% # within each bin, get the min-hash values for each song
  summarise(hash = paste0(hash, collapse = "-")) %>%
  ungroup() -> binned_sig

# inspect results
binned_sig %>%
  head() %>%
  kable()
```

bin	song	hash
1	A Day In The Life	4-22-18-88
1	A Hard Day's Night	59-25-40-2
1	Abbey Road Medley	1-17-2-10
1	Across The Universe	31-33-17-63
1	All My Loving	24-1-13-4
1	All Together Now	71-70-83-175

LSH for the Beatles

```
binned_sig %>%
  group_by(bin, hash) %>%
  filter(n() > 1) %>% # find only those hashes with more ti
  summarise(song = paste0(song, collapse = ";;")) %>%
  separate(song, into = c("song1", "song2"), sep = ";;") %>
  group_by(song1, song2) %>%
  summarise() %>% # get the unique song pairs
  ungroup() -> candidates

# inspect candidates
candidates %>%
  kable()
```

song1	song2
Golden Slumbers	Golden Slumbers/Carry T
Golden Slumbers/Carry That Wieght/Ending	The End
When I'm 64	When I'm Sixty-four

LSH for the Beatles

Notice that LSH has identified the same pairs of documents as potential matches that we found with pairwise comparisons, but did so without having to calculate all of the pairwise comparisons. We can now compute the Jaccard similarity scores for only these candidate pairs of songs instead of all possible pairs.

```
# calculate the Jaccard similarity only for the candidate pairs of songs
candidates %>%
  group_by(song1, song2) %>%
  mutate(jaccard_sim = jaccard_similarity(songs$hash[songs$name == song1][[1]],
                                          songs$hash[songs$name == song2][[1]])) %>%
  arrange(desc(jaccard_sim)) %>%
  kable()
```

song1	song2	jaccard_sim
When I'm 64	When I'm Sixty-four	0.9823529
Golden Slumbers	Golden Slumbers/Carry That Wieght/Ending	0.3557692
Golden Slumbers/Carry That Wieght/Ending	The End	0.2616822

LSH for the Beatles

There is a much easier way to do this whole process using the built in functions in `textreuse` via the functions `textreuse::minhash_generator` and `textreuse::lsh`.

```
# create the minhash function
minhash <- minhash_generator(n = m, seed = 09142017)

# add it to the corpus
corpus <- rehash(corpus, minhash, type="minhashes")

# 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_compare(candidates, corpus, jaccard_similarity, progress = FALSE) %>%
  arrange(desc(score)) %>%
  kable()
```

a	b	score
When I'm 64	When I'm Sixty-four	0.9823529
Golden Slumbers	Golden Slumbers/Carry That Wieght/Ending	0.3557692
Golden Slumbers/Carry That Wieght/Ending	The End	0.2616822