

# Similarity of Taylor Swift songs

We will work with the lyrics of the Beatles as our corpus of documents and discover how similar their songs are lyrically. Can we find any redundant songs?

Here is the list of packages we will use for this analysis.

```
# 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
```

Here are some helpful sites to better understand the above packages (if you are not familiar) that are used heavily in this activity.

- <https://blog.rstudio.com/2014/11/24/rvest-easy-web-scraping-with-r/>
- <http://ggplot2.tidyverse.org/reference/>
- <http://dplyr.tidyverse.org>
- <http://tidyr.tidyverse.org>
- <http://purrr.tidyverse.org>
- <https://cran.r-project.org/web/packages/textreuse/vignettes/textreuse-introduction.html>

## Helpful Trick

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.

Remark: You will need to go through this exercise on your own to fully understand it, line by line. At the end you will do a homework exercise to fully re-inforce the concepts that you have learned from the exercise!

We give two simple examples

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

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

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

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

```
# 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
```

## Get the data

The first step is to actually get the 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 here in case you are interested in it.)

```
# get beatles lyrics
links <- read_html("http://www.metrolyrics.com/taylor-swift-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()
}

# 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()) %>%
  filter(num_copy == 1) %>% # remove exact duplicates (by name)
  select(-num_copy) -> songs
```

We end up with the lyrics to 79 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, "Shake It Off".

## Shingling Shake It Off

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 “Shake It Off” using the function `textreuse::tokenize_ngrams()`.

```
# get k = 5 shingles for "Shake It Off"
best_song <- songs %>% filter(name == "Shake It Off") # this song is very popular
shingles <- tokenize_ngrams(best_song$lyrics, n = 5) # shingle the lyrics

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

---

```
i stay up too late
stay up too late got
up too late got nothing
too late got nothing in
late got nothing in my
got nothing in my brain
```

---

These are the  $k = 5$ -shingles for Shake It Off, 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. I think  $k = 3$  may be sufficient.

```
# get k = 3 shingles for "Shake It Off"
shingles <- tokenize_ngrams(best_song$lyrics, n = 3)

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

---

```
i stay up
stay up too
up too late
too late got
late got nothing
got nothing in
```

---

We can now apply the shingling to all 79 Beatles songs in our corpus.

```
# add shingles to each song
# note mutate adds new variables and preserves existing
songs %>%
  mutate(shingles = (map(lyrics, tokenize_ngrams, n = 3))) -> songs
```

## Jaccard similarity

We can now look at the pairwise *Jaccard similarity coefficient* for each song by looking at the shingles for each song as a set ( $S_i$ ) and computing

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

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

```
# create all pairs to compare then get the jaccard 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:3081      Length:3081      Min.   :0.000000
## Class :character  Class :character 1st Qu.:0.000000
## Mode  :character  Mode  :character Median :0.000000
##                                     Mean  :0.002507
##                                     3rd Qu.:0.001980
##                                     Max.   :0.979167
```

Let's look at these similarity scores graphically.

```
# plot of similarity scores
ggplot(song_sim) + # use ggplot2
  geom_raster(aes(song1, song2, fill = jaccard_sim)) + # tile plot
  theme(axis.text.x = element_text(angle = 90, hjust = 1, size = 4),
        axis.text.y = element_text(size = 4),
        aspect.ratio = 1) # fix axis labels
```

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

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

song1	song2	jaccard_sim
Blank Space	Blank Spaces	0.9791667
22	Twenty Two	0.6968326
Safe And Sound	Safe & Sound	0.7086614
A Love Story	Before The Love Story	0.6161369
State Of Grace	State of Grace	0.8412698

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. Perhaps these are different versions of the song.



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

## character(0)

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

## character(0)
```

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?

```
# filter to find the most similar songs
song_sim %>%
  filter(jaccard_sim <= .5) %>%
  arrange(desc(jaccard_sim)) %>% # sort by similarity
  head() %>%
  kable()
```

song1	song2	jaccard_sim
I Knew You Were Trouble	You Were Wide Awake	0.2455243
A Love Story	Baby's Love Story In My Head	0.2358871
Baby's Love Story In My Head	Before The Love Story	0.1686957
Mean	Mean Planes	0.0470383
Sparks Fly	Fearless	0.0167866
Safe And Sound	Superman	0.0138122

It seems like these songs may also be versions of the same group of songs, but either combined together into a multiple-song block or separated into individual 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()`.

```
# 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)

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

```
##               Look What You Made Me Do Blank Space   Dear John
## Look What You Made Me Do               NA  0.00203666 0.000000000
## Blank Space                           NA           NA 0.001788909
## Dear John                             NA           NA           NA
```

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

a	b	score
22	Twenty Two	0.6968326
A Love Story	Baby's Love Story In My Head	0.2358871
A Love Story	Before The Love Story	0.6161369
Baby's Love Story In My Head	Before The Love Story	0.1686957
Blank Space	Blank Spaces	0.9791667
I Knew You Were Trouble	You Were Wide Awake	0.2455243
Safe & Sound	Safe And Sound	0.7086614
State of Grace	State Of Grace	0.8412698

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 3,081 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*.

## 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 “Shake It Off”.

```
# look at the hashed shingles for "Shake It Off"
tokens(corpus)$`Shake It Off` %>% head()
```

```
## [1] "i stay up"      "stay up too"    "up too late"
## [4] "too late got"   "late got nothing" "got nothing in"
```

```
hashes(corpus)$`Shake It Off` %>% head()
```

```
## [1] 1334746969 -2069270204 -1072229794 -754380331 -1072674653 -499472535
```

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 == "Shake It Off"][[1]] %>% head()
```

```
## [1] 1334746969 -2069270204 -1072229794 -754380331 -1072674653 -499472535
```

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
```

As you can see, the Jaccard similarities are all the same whether they are computed on strings or hashes due to the fact that all we are computing are set intersections and unions. This allows us to not have to store the shingles, but rather just the shingled hashes.

Another way that we can make this process a more computationally feasible is by using the combination of *minhash* and *locality-sensitive hashing (LSH)* to compute possible matches only once for each document, so that the cost of computation grows linearly rather than exponentially.

## Minhashing

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.

```
# 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 tidy documentation)

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

item	... Ready For It?	22	A Love Story
-2147481894	0	0	0
-2147216718	0	0	0
-2146926393	0	0	0
-2146906553	0	0	0

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.

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.

```
# 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()
```

... Ready For It?	134
22	70
A Love Story	1
A Place In This World	99

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()
```

... Ready For It?	22	A Love Story	A Place In This World
134	70	1	99
49	6	4	485
21	41	107	265
42	32	28	22

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.

```
# 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 0.0008477 0.0009051 0.0522480
```

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.

## Locality-sensitive hashing (LSH)

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.

The idea is that 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 hashings is called a candidate pair and we only check candidate pairs for similarity.

This is accomplished by using the signature matrix from the minhashing.

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.

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  $s$ ,

$$P(\text{pair of documents with a Jaccard similarity } s \text{ will be marked as potential matches}) = 1 - (1 - s^{m/b})^b$$

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

```

# 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)))

```

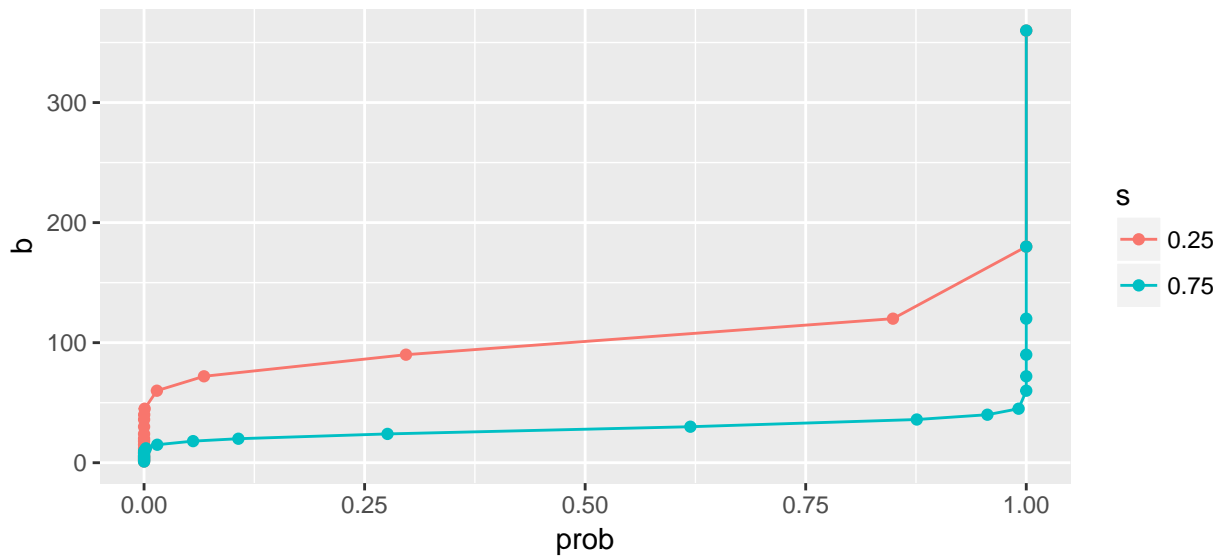


Figure 2: Probability that a pair of documents with a Jaccard similarity  $s$  will be marked as potential matches for various bin sizes  $b$  for  $s = .25, .75$  for the number of permutations we did,  $m = 360$ .

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

```

# look at 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  $b = 120$ , a pair of records with Jaccard similarity of .25 will have a 84.9% chance of being matched as candidates and a pair of records with Jaccard similarity of .75 will have a 100% chance of being matched as candidates. This seems adequate for our application, so I will continue with  $b = 120$ .

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

```
# bin the signature matrix
b <- 120
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	...Ready For It?	134-49-21
1	22	70-6-41
1	A Love Story	1-4-107
1	A Place In This World	99-485-265
1	All Too Well	105-87-37
1	Baby's Love Story In My Head	1-4-55

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

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

song1	song2
22	Twenty Two
A Love Story	Baby's Love Story In My Head
A Love Story	Before The Love Story
Blank Space	Blank Spaces
I Knew You Were Trouble	You Were Wide Awake
Safe & Sound	Safe And Sound
State of Grace	State Of Grace

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
Blank Space	Blank Spaces	0.9791667
State of Grace	State Of Grace	0.8412698
Safe & Sound	Safe And Sound	0.7086614
22	Twenty Two	0.6968326
A Love Story	Before The Love Story	0.6161369
I Knew You Were Trouble	You Were Wide Awake	0.2455243
A Love Story	Baby's Love Story In My Head	0.2358871

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
Blank Space	Blank Spaces	0.9791667
State of Grace	State Of Grace	0.8412698
Safe & Sound	Safe And Sound	0.7086614
22	Twenty Two	0.6968326
A Love Story	Before The Love Story	0.6161369
I Knew You Were Trouble	You Were Wide Awake	0.2455243
A Love Story	Baby's Love Story In My Head	0.2358871

## Your turn!

Now that we've walked through the analysis of Beatles songs, it's your turn to investigate another band.

1. Scrape lyrics from <http://www.metrolyrics.com> for Taylor Swift (<http://www.metrolyrics.com/taylor--swift-lyrics.html>).
  - Don't forget to remove duplicates and broken links
2. Construct shingles of using the song "Shake it Off."
  - Explore lengths of shingles.
3. Construct shingles of all songs in your corpus.
4. Hash all of your shingled lyrics.
5. Compute pairwise Jaccard similarity coefficients for all songs.
  - Do this using the hashed lyrics
  - Approximate using the minhash.
6. Use LSH to compute the Jaccard similarity only for candidates.
  - How many buckets will you use?
  - How many minhashes?
7. Give a paragraph description of your analysis and your conclusions. How did this differ from looking at the Beatles.