

2 Similitud: Minhashing

En esta parte trata de un problema fundamental en varias tareas de minería de datos: ¿cómo medir similitud, y cómo encontrar vecinos cercanos en un conjunto de elementos?

Algunos ejemplos son:

- Encontrar documentos similares en una colección de documentos (este es el que vamos a tratar más). Esto puede servir para detectar plagio, deduplicar noticias o páginas web, etc.
- Encontrar imágenes similares en una colección grande.
- Encontrar usuarios similares (Netflix), en el sentido de que tienen gustos similares. O películas similares, en el sentido de que le gustan a las mismas personas
- Uber: rutas similares que indican (fraude o abusos) [<https://eng.uber.com/lsh/>].

Estos problemas no son triviales por dos razones:

- Los elementos que queremos comparar muchas veces están naturalmente representados en espacios de dimensión muy alta, y es relativamente costoso comparar un par (documentos, imágenes, usuarios, rutas).
- Si la colección de elementos es grande (N), entonces el número de pares posibles es del orden de N^2 , y no es posible hacer todas las posibles comparaciones para encontrar los elementos similares (por ejemplo, comparar 100 mil documentos, con unas 10000 comparaciones por segundo, tardaría alrededor de 10 días).

El tema principal de esta parte es el siguiente:

- Podemos usar reducción probabilística de dimensión (usando funciones hash) para reducir la dimensionalidad del problema de similitud, sin perder mucha precisión en el cálculo de similitudes.
- Podemos usar métodos probabilísticos para agrupar elementos similares (encontrar vecinos cercanos), sin necesidad de calcular TODAS las similitudes posibles.

2.1 Similitud de conjuntos

Muchos de estos problemas de similitud se pueden pensar como problemas de similitud entre conjuntos. Por ejemplo, los documentos son conjuntos de palabras, pares de palabras, sucesiones de caracteres, etc, una película como el conjunto de personas a las que le gustó, o una ruta como un conjunto de tramos, etc.

Hay muchas medidas que son útiles para cuantificar la similitud entre conjuntos. Una que es popular, y que explotaremos por sus propiedades, es la similitud de Jaccard:

La similitud de Jaccard de los conjuntos A y B está dada por

$$\text{sim}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Esta medida cuantifica qué tan cerca está la unión de A y B de su intersección. Cuanto más parecidos sean $A \cup B$ y $A \cap B$, más similares son los conjuntos. En términos geométricos, es el área de la intersección entre el área de la unión.

Ejercicio



Calcula la similitud de jaccard entre los conjuntos $A = \{5, 2, 34, 1, 20, 3, 4\}$ y $B = \{19, 1, 2, 5\}$

```

library(tidyverse)
library(textreuse)

sim_jaccard <- function(a, b){
  length(intersect(a, b)) / length(union(a, b))
}

sim_jaccard(c(0,1,2,5,8), c(1,2,5,8,9))
## [1] 0.6666667
sim_jaccard(c(2,3,5,8,10), c(1,8,9,10))
## [1] 0.2857143
sim_jaccard(c(3,2,5), c(8,9,1,10))
## [1] 0

```

2.2 Representación en tejas para documentos

En primer lugar, buscamos representaciones de documentos como conjuntos. Hay varias maneras de hacer esto.

Consideremos una colección de textos cortos:

```

textos <- character(4)
textos[1] <- 'el perro persigue al gato.'
textos[2] <- 'el gato persigue al perro'
textos[3] <- 'este es el documento de ejemplo'
textos[4] <- 'el documento con la historia del perro y el gato'

```

Los métodos que veremos aquí se aplican para varias representaciones:

- La representación más simple es la bolsa de palabras, que es conjunto de palabras que contiene un documento. Podríamos comparar entonces documentos calculando la similitud de jaccard de sus **bolsas de palabras (1-gramas)**

```
 tokenize_words(textos[1])

## [1] "el"       "perro"     "persigue"  "al"       "gato"
```

- Podemos generalizar esta idea y pensar en **n-gramas** de palabras, que son **sucesiones de *n* palabras** que ocurren en un documento.

```
 tokenize_ngrams(textos[1], n = 2)
```

```
## [1] "el perro"      "perro persigue" "persigue al"    "al gato"
```

- Otro camino, es el **k-tejas**, que son k-gramas de **caracteres**

```
shingle_chars <- function(string, lowercase = FALSE, k = 4){

  # produce shingles (con repeticiones)

  if(lowercase) {

    string <- str_to_lower(string)

  }

  shingles <- seq(1, nchar(string) - k + 1) %>%
    map_chr(function(x) substr(string, x, x + k - 1))

  shingles

}

ejemplo <- shingle_chars('Este es un ejemplo', 4)
ejemplo
## [1] "este" "ste " "te e" "e es" " es " "es u" "s un" " un " "un e" "n ej"
## [11] " eje" "ejem" "jemp" "empl" "mplo"
```

Si lo que nos interesa principalmente similitud textual (no significado, o polaridad, etc.) entre documentos, entonces podemos comparar dos documentos considerando que sucesiones de caracteres de tamaño fijo ocurren en ambos documentos, usando *k*-tejas. Esta representación es **flexible** en el sentido de que se puede adaptar para documentos muy cortos (mensajes o tweets, por ejemplo), pero también para documentos más grandes.

Tejas (shingles)

Sea $k > 0$ un entero. Las k -tejas (k -shingles) de un documento d es el conjunto de todas las corridas (distintas) de k caracteres sucesivos.

Es importante escoger suficientemente grande, de forma que la probabilidad de que una teja particular tenga probabilidad baja de ocurrir en un texto dado. Si los textos son cortos, entonces basta tomar valores como $k = 4, 5$, pues hay un total de 27^4 tejas de tamaño 4, y el número de tejas de un documento corto (mensajes, tweets) es mucho más bajo que 27^4 (nota: ¿puedes explicar por qué este argumento no es exactamente correcto?)

Para documentos grandes, como noticias o artículos, es mejor escoger un tamaño más grande, como $k = 9, 10$, pues en documentos largos puede haber cientos de miles de caracteres, si k fuera más chica entonces una gran parte de las tejas aparecería en muchos de los documentos.

Ejemplo

Documentos textualmente similares tienen tejas similares:

```
textos <- character(4)
textos[1] <- 'el perro persigue al gato, pero no lo alcanza'
textos[2] <- 'el gato persigue al perro, pero no lo alcanza'
textos[3] <- 'este es el documento de ejemplo'
textos[4] <- 'el documento habla de perros, gatos, y otros animales'
tejas_doc <- lapply(textos, shingle_chars, k = 4)
sim_jaccard(tejas_doc[[1]], tejas_doc[[2]])
## [1] 0.7391304
sim_jaccard(tejas_doc[[1]], tejas_doc[[3]])
## [1] 0
sim_jaccard(tejas_doc[[4]], tejas_doc[[3]])
## [1] 0.1666667
```

Observación: las n -tejas de palabras se llaman usualmente n -gramas. Lo que veremos aquí aplica para estos dos casos.

2.3 Reducción probabilística de dimensión.

La representación de k-tejas de documentos es una representación de dimensión alta (pues hay muchas tejas), pues cada documento se escribir como un vector de 0s y 1s:

```
todas_tejas <- Reduce('c', tejas_doc) %>% unique %>% sort
vector_1 <- as.numeric(todas_tejas %in% tejas_doc[[1]])
vector_1

## [1] 1 1 0 0 0 0 0 0 1 0 1 1 0 1 0 0 1 0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 0 0 0
## [36] 1 0 0 0 0 0 0 1 0 0 1 1 1 0 0 1 1 0 1 0 0 0 1 1 0 1 1 0 0 0 0 1 0 1 0
## [71] 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 1
## [106] 0 0
```

Para esta colección chica, con k relativamente chico, el vector que usamos para representar cada documento es de tamaño 107, pero en otros casos este número será mucho más grande.

Podemos construir explícitamente la matriz de tejas-documentos de las siguiente forma (OJO: esto normalmente **no** queremos hacerlo, pero lo hacemos para ilustrar):

```
df <- data_frame(id_doc = paste0('doc_',
                                    seq(1, length(tejas_doc))),
                   tejas = tejas_doc) %>%
  unnest %>%
  unique %>%
  mutate(val = 1) %>%
  spread(id_doc, val, fill = 0)
df
```

```
## # A tibble: 107 x 5
##   tejas    doc_1  doc_2  doc_3  doc_4
##   * <chr>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 "al"     1.00    1.00    0       0
## 2 "alc"    1.00    1.00    0       0
## 3 "ani"    0       0       0       1.00
## 4 "de"     0       0       1.00    1.00
## 5 "doc"    0       0       1.00    1.00
## 6 "eje"    0       0       1.00    0
## 7 "el"     0       0       1.00    0
## 8 "es"     0       0       1.00    0
## 9 "gat"    1.00    1.00    0       1.00
## 10 "hab"   0      0       0       1.00
## # ... with 97 more rows
```

¿Cómo calculamos la similitud de Jaccard usando estos datos?

Calcular la unión e intersección se puede hacer haciendo OR y AND de las columnas, y entonces podemos calcular la similitud

```
inter_12 <- sum(df$doc_1 & df$doc_2)
union_12 <- sum(df$doc_1 | df$doc_2)
similitud <- inter_12/union_12
similitud # comparar con el número que obtuvimos arriba.
```

```
## [1] 0.7391304
```

Ahora consideramos una manera probabilística de reducir la dimensión de esta matriz sin perder información útil para calcular similitud. Queremos obtener una matriz con menos renglones (menor dimensión) y las mismas columnas.

Las proyecciones que usaremos son escogidas al azar, y son sobre el espacio de enteros.

- Sea π una permutación al azar de los renglones de la matriz.
- Permutamos los renglones de la matriz tejas-documentos según π .

- Definimos un nuevo descriptor del documento: para cada documento (columna) d de la matriz permutada, tomamos el entero $f_{\pi}(d)$, que da el número del primer renglón que es distinto de 0

2.3.0.1 Ejercicio

Considera la matriz de tejas-documentos para cuatro documentos y cinco tejas dada a continuación, con las permutaciones $(2, 3, 4, 5, 1)$ (indica que el renglón 1 va al 2, el 5 al 1, etc.) y $(2, 5, 3, 1, 4)$.

##	d_1	d_2	d_3	d_4		d_1	d_2	d_3	d_4		d_1	d_2	d_3	d_4
## abc	1	0	0	1	1	0	0	1	0	1	0	1	1	0
## ab	0	0	1	0	2	1	0	0	1	2	1	0	0	1
## xyz	0	1	0	1	3	0	0	1	0	3	0	1	0	1
## abx	1	0	1	1	4	0	1	0	1	4	0	0	2	0
## abd	0	0	1	0	5	1	0	1	1	5	0	0	1	0

2 4 1 2 1 3 1 1

Ejemplo

Ordenamos al azar:

```
set.seed(321)
df_1 <- df %>% sample_n(nrow(df))
head(df_1, 14)
```



```
## # A tibble: 14 x 5
##   tejas doc_1 doc_2 doc_3 doc_4
##   <chr>  <dbl> <dbl> <dbl> <dbl>
## 1 tos,     0     0     0    1.00
## 2 to h     0     0     0    1.00
## 3 anza    1.00  1.00  0     0
## 4 "ato "   0     1.00  0     0
## 5 el d    0     0     1.00  1.00
## 6 docu    0     0     1.00  1.00
## 7 "ero "   1.00  1.00  0     0
## 8 atos    0     0     0     1.00
## 9 ento    0     0     1.00  1.00
## 10 pero   1.00  1.00  0     0
## 11 l pe   1.00  1.00  0     0
## 12 "to, "  1.00  0     0     0
## 13 o no   1.00  1.00  0     0
## 14 " doc"  0     0     1.00  1.00
```

```
primer_uno <- function(col){
  purrr::detect_index(col, function(x) x > 0)
}

df_1 %>% summarise_if(is.numeric, primer_uno)

## # A tibble: 1 x 4
##   doc_1 doc_2 doc_3 doc_4
##   <int> <int> <int> <int>
## 1     3     3     5     1
```

Ahora repetimos con otras permutaciones:

```

set.seed(32)

num_hashes <- 10

permutaciones <- sapply(1:num_hashes, function(i){

  sample(1:nrow(df), nrow(df))

})

firmas_df <- lapply(1:num_hashes, function(i){

  df_1 <- df[order(permutaciones[,i]),]

  df_1 %>% summarise_if(is.numeric, primer_uno)

}) %>% bind_rows()

firmas_df <- firmas_df %>% add_column(firma = paste0('f_', 1:num_hashes),
                                         .before = 1)

firmas_df

## # A tibble: 10 x 5
##   firma doc_1 doc_2 doc_3 doc_4
##   <chr> <int> <int> <int> <int>
## 1 f_1     7     7     1     2
## 2 f_2     1     1     4     2
## 3 f_3     2     2     7     1
## 4 f_4     3     5     1     1
## 5 f_5     7     3     5     1
## 6 f_6     1     1     8     2
## 7 f_7     2     2     5     1
## 8 f_8     1     1     2     2
## 9 f_9     2     2     3     1
## 10 f_10   1     1     2     1

```

A esta nueva matriz le llamamos **matriz de firmas** de los documentos. La firma de un documento es una sucesión de enteros.

Cada documento se describe ahora con 10 entradas, en lugar de 107.

Nótese que por construcción, cuando dos documentos son muy similares, es natural que sus columnas de firmas sean similares, pues al hacer las permutaciones es altamente probable que el primer 1 ocurra en la misma posición.

Resulta que podemos cuantificar esta probabilidad. Tenemos el siguiente resultado simple pero sorprendente:

Sea π una permutación escogida al azar, y a y b dos columnas dadas. Entonces

$$\underline{P(f_{\pi}(a) = f_{\pi}(b)) = sim(a, b)}$$

donde sim es la similitud de jaccard basada en las tejas usadas.

Sean $\pi_1, \pi_2, \dots, \pi_n$ permutaciones escogidas al azar de manera independiente. Si n es grande, entonces por la ley de los grandes números

$$\underline{sim(a, b) \approx \frac{|\{\pi_j : f_{\pi_j}(a) = f_{\pi_j}(b)\}|}{n}},$$

es decir, la similitud de jaccard es aproximadamente la proporción de elementos de las firmas que coinciden.

Ejemplo

Antes de hacer la demostración, veamos como aplicaríamos a la matriz de firmas que calculamos arriba. Tendríamos, por ejemplo :

```
mean(firmas_df$doc_1 == firmas_df$doc_2)
## [1] 0.8
mean(firmas_df$doc_1 == firmas_df$doc_3)
## [1] 0
mean(firmas_df$doc_3 == firmas_df$doc_4)
## [1] 0.2
```

Ahora veamos qué sucede repetimos varias veces:

```

firmas_rep <- lapply(1:50, function(i){
  firmas_df <- lapply(1:20, function(i){
    df_1 <- df %>% sample_n(nrow(df))
    df_1 %>% summarise_if(is.numeric, primer_uno)
  }) %>% bind_rows()
  firmas_df$rep <- i
  firmas_df
})

sapply(firmas_rep, function(mat){
  mean(mat[, 1] == mat[,2])
}) %>% quantile(probs = c(0.1,0.5,0.9))
##   10%   50%   90%
## 0.600 0.725 0.850

sapply(firmas_rep, function(mat){
  mean(mat[, 3] == mat[,4])
}) %>% quantile(probs = c(0.1,0.5,0.9))
##   10%   50%   90%
## 0.05 0.15 0.25

```

Observación: si la similitud de dos documentos es cero, entonces este procedimiento siempre da la respuesta exacta. ¿Por qué?

Ahora damos un argumento de este resultado. Consideraremos dos columnas a, b de la matriz de 0's y 1's, con conjuntos de tejas asociados A, B .

- Supongamos que entre las dos columnas a y b , el primer 1 ocurre en el renglón k .
- El renglón k puede ser de tipo $(1, 0), (0, 1), (1, 1)$. Todos estos renglones tienen la misma probabilidad de aparecer en el renglón k . El número de estos renglones es el tamaño de $A \cup B$, pues este número cuenta cuántas tejas en común tienen estos conjuntos.
- El número de renglones de tipo $(1, 1)$ es el tamaño de $A \cap B$, el número de tejas en común de los dos documentos.
- Entonces, la probabilidad condicional de que el renglón k sea de tipo $(1, 1)$, dado que es de algún tipo de $(1, 0), (0, 1), (1, 1)$, es

$$\frac{|A \cap B|}{|A \cup B|},$$

que es la similitud de Jaccard de los dos documentos.

2.4 Mejoras al método de permutaciones

En la sección anterior propusimos una manera probabilística de reducir dimensionalidad para el problema de calcular similitud de Jaccard usando proyecciones aleatorias de los datos basadas en permutaciones de las tejas. Esto nos da una representación más compacta, que es más fácil de almacenar en memoria.

El problema con el procedimiento de arriba es el costo de calcular las permutaciones y permutar la matriz característica (tejas-documentos).

Primero escribimos un algoritmo para hacer el cálculo de la matriz de firmas dado que tenemos las permutaciones, sin permutar la matriz y recorriendo por renglones.

Supongamos que tenemos π_1, \dots, π_k permutaciones. Denotamos por $SIG_{i,c}$ el elemento de la matriz de firmas para la i -ésima permutación y el documento c , y escribimos $h_i = \pi_i$.

Cálculo de matriz de firmas

Inicializamos la matriz de firmas como $SIG_{i,c} = \infty$. Para cada renglón r :

- Para cada columna c :
 1. Si c tiene un cero en el renglón r , no hacemos nada.
 2. Si c tiene un uno en el renglón r , ponemos $SIG_{i,c} = \min\{SIG_{i,c}, h_i(r)\}$.

Ejercicio

Aplicar este algoritmo al ejercicio 2.3.0.1.

	d_1	d_2	d_3	d_4	H_1	H_2
abc	1	0	0	1	2	2
ab	0	0	1	0	3	5
xyz	0	1	0	1	1	3
abx	1	0	1	1	1	2
abd	0	0	1	0	1	4

Renglón 1
z cero? minhash 2
 H_1 2 ∞ ∞ 3 H_2
 H_2 2 ∞ ∞ 2

Renglón 2
z cero? minhash 2
 H_1 2 ∞ 3 2
 H_2 2 ∞ 5 2

Renglón 3
z cero?
 H_1 2 1 3 2

Renglón 4
z cero?
 H_1 2 3 2 1

Renglón 5
z cero?
 H_1 2 9 1 2
 H_2 3 1 1

Consideramos el ejemplo que vimos antes

df

```
## # A tibble: 107 x 5
##   tejas doc_1 doc_2 doc_3 doc_4
##   * <chr>  <dbl> <dbl> <dbl> <dbl>
## 1 "al"   1.00  1.00  0     0
## 2 "alc"  1.00  1.00  0     0
## 3 "ani"  0     0     0     1.00
## 4 "de"   0     0     1.00  1.00
## 5 "doc"  0     0     1.00  1.00
## 6 "eje"  0     0     1.00  0
## 7 "el"   0     0     1.00  0
## 8 "es"   0     0     1.00  0
## 9 "gat"  1.00  1.00  0     1.00
## 10 "hab" 0    0     0     1.00
## # ... with 97 more rows
```

```
mat_df <- df %>% select(-tejas) %>% as.matrix
calc_firmas <- function(mat_df, permutaciones){
  firmas <- list()
  num_hashes <- ncol(permutaciones)
  firmas <- sapply(1:ncol(mat_df), function(r) rep(Inf, num_hashes))
  for(r in 1:nrow(df)){
    indices <- mat_df[r, ] > 0
    firmas[, indices] = pmin(firmas[, indices], permutaciones[r, ])
  }
  firmas
}
calc_firmas(mat_df, permutaciones)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    7    7    1    2
## [2,]    1    1    4    2
## [3,]    2    2    7    1
## [4,]    3    5    1    1
## [5,]    7    3    5    1
## [6,]    1    1    8    2
## [7,]    2    2    5    1
## [8,]    1    1    2    2
## [9,]    2    2    3    1
## [10,]   1    1    2    1
```

2.5 Min-hashing

Cuando vemos este algoritmo, nos damos cuenta de que las funciones h_i no necesariamente tienen que ser una permutación de los renglones. Simplemente estamos buscando en cada columna el mínimo entero que corresponde a una teja que aparezca en la columna. Podemos **simular** estas permutaciones de las siguiente forma:

Si h es una función que envía los renglones (tejas) a un rango grande de enteros, podríamos aplicar el algoritmo con los enteros que produce esta función. Para estar cerca de simular las permutaciones, necesitamos:

- Una familia de funciones que sean fáciles de calcular, y que podamos escoger al azar entre ellas.
- Si escogemos una función al azar de esta familia, necesitamos que la probabilidad de que $h(x) = h(y)$ para un par x,y de tejas sea muy baja (baja probabilidad de colisión al mismo entero). En las permutaciones no tenemos colisiones.

Estas son, entre otras, propiedades de funciones hash, y hay varias maneras de construirlas.

En (Leskovec, Rajaraman, and Ullman 2014), por ejemplo, una sugerencia es construir una familia como sigue: Si tenemos m posibles tejas (renglones), escogemos un primo mayor a m . En nuestro ejemplo con 107 tejas, podríamos tomar el primo 113 y hacer:

```

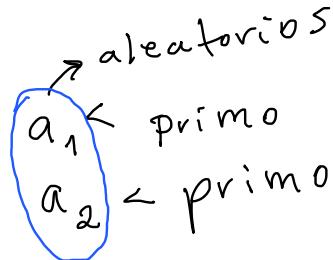
num_renglones <- nrow(mat_df)
hash_simple <- function(...){
  primo <- 113
  a <- sample.int(primo - 1, 2)
  out_fun <- function(x) {
    ((a[1]*(x-1) + a[2]) % primo) + 1
  }
  out_fun
}

```

`set.seed(1323)``hash_f <- lapply(1:20, hash_simple)`

hash_f contiene 20 funciones con

Observación: distintas (a_1, a_2)



x input

$$(a_1(x-1) + a_2 \bmod \text{primo}) + 1$$

esta función manda las
 x 's a un número

- Usamos un primo en la congruencia para evitar casos con muchas colisiones, por ejemplo: si por azar escogemos $10x + 7 \bmod 110$, entonces todos los múltiplos de 11 caen en la misma cubeta 7.

Veamos cómo funciona en nuestro ejemplo:

`hashes <- sapply(hash_f, function(f) f(1:num_renglones))``dim(hashes)``## [1] 107 20``hashes[1:10, 1:5]`

se aplican las funciones
hash a los números de
renglones (107)

obtenemos 107 renglones
por cada función
(20)

```

## [,1] [,2] [,3] [,4] [,5]
## [1,] 30   69   107  106  62
## [2,] 109  56   34   36   106
## [3,] 75   43   74   79   37
## [4,] 41   30   1    9    81
## [5,] 7    17   41   52   12
## [6,] 86   4    81   95   56
## [7,] 52   104  8    25   100
## [8,] 18   91   48   68   31
## [9,] 97   78   88   111  75
## [10,] 63   65   15   41   6

```

$h(1) = 30$
 $h(2) = 109$

Estas son nuestra permutaciones simuladas. Ahora aplicamos el algoritmo de arriba:

```

firmas_2 <- calc_firmas(mat_df, hashes)
firmas_2

```

[,1] [,2] [,3] [,4] → ↗ documentos

## [1,]	10	13	1	2
## [2,]	2	2	1	1
## [3,]	2	2	1	1
## [4,]	1	1	2	8
## [5,]	2	2	1	2
## [6,]	2	2	4	1
## [7,]	8	5	4	1
## [8,]	3	3	1	1
## [9,]	6	4	5	2
## [10,]	5	1	11	2
## [11,]	3	3	2	1
## [12,]	3	8	6	1
## [13,]	2	2	1	2
## [14,]	3	2	6	4
## [15,]	1	2	3	3
## [16,]	3	3	16	1
## [17,]	1	5	13	2
## [18,]	4	4	3	1
## [19,]	8	2	1	3
## [20,]	2	2	5	4

↳ firmas

```
mean(firmas_2[,1]==firmas_2[,2])
```

```
## [1] 0.55
```

```
mean(firmas_2[,1]==firmas_2[,3])
```

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

```
mean(firmas_2[,3]==firmas_2[,4])
```

```

## [1] 0.2

hash_f <- lapply(1:20, hash_simple)
hashes <- sapply(hash_f, function(f) f(1:num_renglones))
firmas_2 <- calc_firmas(mat_df, hashes)
mean(firmas_2[,1]==firmas_2[,2])

## [1] 0.85

mean(firmas_2[,1]==firmas_2[,3])

## [1] 0

mean(firmas_2[,3]==firmas_2[,4])

## [1] 0.15

```

Y estas son nuestras estimaciones de la similitud de Jaccard.

Podemos usar mejores funciones hash, que no requieren de ajustar parámetros para cada problema, como en el paquete *textruese* (Mullen 2016), que utiliza hashes de cadenas (que serán las tejas) a los enteros, y y utiliza como base función hash ampliamente probada (De librerías de Boost para C++). Por ejemplo, la función *hash_string* del paquete *textruese*:

```

hash_string('a')
## [1] 1424956863

hash_string('El perro persigue al gato')
## [1] -128122489

hash_string('El perro persigue al gat')
## [1] 1678899309

```

Y por ejemplo, si mapeamos las tejas de un documento, los hashes correspondientes para esta función son:

```
hash_string(shingle_chars('El perro persigue al gato'))
```

```
## [1] -1551000127 -231718157 1927879825 98111507 1474359759
## [6] -1016524329 1824378030 821460964 1927879825 -1912846923
## [11] 2087484239 946985811 -1470902616 593787156 2023930193
## [16] -135735541 -2145380970 -1361166998 -1642026314 2020970990
## [21] -1561732516 1021355903
```

Min-hashing (con permutaciones) Para obtener la estimación de min-hashing de la similitud de dos documentos:

1. Convertimos los documentos a tejas
 2. Escogemos al azar funciones hash h_1, h_2, \dots, h_k que mapean tejas a un rango grande enteros.
 3. Aplicamos el algoritmo anterior para encontrar la matriz de firmas.
 4. Calculamos la fracción de coincidencias de las dos firmas.
- $\rightarrow \min(h, \infty)$

Observación: El paso 3 también podemos hacerlo por columna: simplemente hay que calcular los hashes de las tejas y tomar el mínimo valor.

```

library(textreuse)
set.seed(253)
options("mc.cores" = 4L)
minhash <- minhash_generator(50)
corpus <- TextReuseCorpus(text = textos,
                           tokenizer = shingle_chars,
                           minhash_func = minhash,
                           keep_tokens = TRUE)

# En este objeto:
# hashes: Los hashes de las tejas, con la función hash base - puede ser útil para # almacenar
# minhashes: contiene los valores minhash bajo las funciones hash
# que escogimos al azar en minhash_generator.

str(corpus[[1]])

## List of 5
## $ content :Class 'String' chr "el perro persigue al gato, pero no lo alcanza"
## $ tokens  : chr [1:42] "el p" "l pe" " per" "perr" ...
## $ hashes   : int [1:42] -1487917609 -231718157 1927879825 98111507 1474359759 -10165243
## $ minhashes: int [1:50] -2063752536 -2142368545 -2123813470 -2139507477 -1855378276 -20
## $ meta     :List of 4
##   ..$ hash_func  : chr "hash_string"
##   ..$ id         : chr "doc-1"
##   ..$ minhash_func: chr "minhash"
##   ..$ tokenizer  : chr "shingle_chars"
## - attr(*, "class")= chr [1:2] "TextReuseTextDocument" "TextDocument"

minhashes_corpus <- minhashes(corpus)



Lista de 4 objetos: cada objeto de 50 entradas


mean(minhashes_corpus[[1]]==minhashes_corpus[[2]])
```

[1] 0.66

```
mean(minhashes_corpus[[1]]==minhashes_corpus[[3]])
```

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

```
mean(minhashes_corpus[[4]]==minhashes_corpus[[3]])
```

```
## [1] 0.14
```

Observación:

- El cálculo de minhashes es fácilmente escalable: por ejemplo, podemos procesar grupos de documentos en paralelo (enviando las funciones hash a los trabajadores) para obtener las firmas de ese bloque de documentos. ¿Cómo se podría hacer esto si los datos estuvieran distribuidos por renglones (tejas)?

Ejemplo

Consideramos un ejemplo de unos 2000 tweets:

```
minhash <- minhash_generator(50)

x <- scan("../datos/similitud/gamergate_antigg.txt", what="", sep="\n")

# este caso ponemos en hash_func Los minhashes, para después
# usar la función pairwise_compare (que usa Los hashes)
system.time(
  corpus_tweets <- TextReuseCorpus(text = x,
    tokenizer = shingle_chars,
    k = 5,
    lowercase = TRUE,
    hash_func = minhash,
    keep_tokens = TRUE,
    keep_text = TRUE, skip_short = FALSE))
```

```
##      user  system elapsed
## 1.960   0.180   0.808
```

Busquemos tweets similares a uno en particular

```
corpus_tweets[[16]]$content
```

```
## @femfreq Are you really this stupid? Yes you are. The Suffragettes/first wave also broug
```

```
mh <- hashes(corpus_tweets)
similitud <- sapply(mh, function(x) mean(mh[[16]]==x))
indices <- which(similitud > 0.5)
names(indices)
```

```
## [1] "doc-11" "doc-16"
```

```
corpus_tweets[['doc-16']]$content
```

```
## @femfreq Are you really this stupid? Yes you are. The Suffragettes/first wave also broug
```

```
corpus_tweets[['doc-11']]$content
```

```
## @femfreq Are you really this stupid? Yes you are. The Suffragettes also brought in The F
```

```
similitud <- sapply(mh, function(x) mean(mh[[186]]==x))
indices <- which(similitud > 0.35)
names(indices)
```

```

## [1] "doc-107"  "doc-186"  "doc-545"  "doc-859"  "doc-1657"

lapply(names(indices), function(nom) corpus_tweets[[nom]]$content)

## [[1]]
## Reply to @femfreq #GamerGate explained https://t.co/AJzRp5qQqa
##
## [[2]]
## @femfreq #gamergate explained https://t.co/AJzRp5qQqa
##
## [[3]]
## @femfreq @xoxo #GamerGate explained https://t.co/AJzRp5qQqa
##
## [[4]]
## @femfreq https://t.co/al1xLZlikc__^_^ #GamerGate
##
## [[5]]
## @femfreq @gameinformer #gamergate explained http://t.co/FLlnImHVxo

```

¿Cuáles son las verdaderas distancias de jaccard? Por ejemplo,

```

jaccard_similarity(
  shingle_chars(corpus_tweets[["doc-545"]]$content, lowercase=TRUE, k = 5),
  shingle_chars(corpus_tweets[["doc-1657"]]$content, lowercase=TRUE, k = 5)
)

## [1] 0.3764706

```

Este es un falso positivo (no tiene similitud mayor a 0.35):

```
jaccard_similarity(
  shingle_chars(corpus_tweets[["doc-545"]])$content, lowercase=TRUE, k = 5),
  shingle_chars(corpus_tweets[["doc-859"]])$content, lowercase=TRUE, k = 5)
)

## [1] 0.3026316
```

Observación: Una vez que calculamos los que tienen similitud aproximada > 0.35 , podemos calcular la función de jaccard exacta para los elementos similares resultantes.

2.6 Buscando vecinos cercanos

Aunque hemos reducido el trabajo para hacer comparaciones de documentos, no hemos hecho mucho avance en encontrar todos los pares similares de la colección completa de documentos. Intentar calcular similitud para todos los pares (del orden n^2) es demasiado trabajo:

```
system.time(
  pares <- pairwise_compare(corpus_tweets[1:200], ratio_of_matches) %>%
    pairwise_candidates()

##      user    system elapsed
##     8.660   0.010   8.708

  pares <- pares %>% filter(score > 0.20) %>% arrange(desc(score))

  pares
```

```
## # A tibble: 125 x 3
##   a         b      score
##   <chr>    <chr>    <dbl>
## 1 doc-45   doc-47   1.00
## 2 doc-1     doc-2    0.740
## 3 doc-107   doc-186  0.740
## 4 doc-11    doc-16   0.700
## 5 doc-52    doc-53   0.660
## 6 doc-3     doc-4    0.640
## 7 doc-129   doc-130  0.600
## 8 doc-104   doc-32   0.440
## 9 doc-117   doc-8    0.440
## 10 doc-199  doc-200  0.440
## # ... with 115 more rows
```

`corpus_tweets[["doc-107"]]$content`

`## Reply to @femfreq #GamerGate explained https://t.co/AJzRp5qQqa`

`corpus_tweets[["doc-186"]]$content`

`## @femfreq #gamergate explained https://t.co/AJzRp5qQqa`

Y si quisieramos entender esto, todavía faltaría hacer clusters basados en los scores, para agrupar todos los tweets similares.

En la siguiente parte veremos como aprovechar estos minhashes para hacer una búsqueda eficiente de pares similares.

2.7 Locality sensitive hashing (LSH) para documentos

Como discutimos arriba, calcular todas las posibles similitudes de una colección de un conjunto no tan grande de documentos es difícil. Sin embargo, muchas veces lo que nos interesa es simplemente agrupar colecciones de documentos que tienen alta similitud (por ejemplo para deduplicar, hacer clusters de usuarios muy similares, etc.).

Una técnica para encontrar vecinos cercanos de este tipo es LSH. Comenzamos construyendo LSH basado en las firmas de minhash. La idea general es:

- Recorremos la matriz de firmas documento por documento
- Asignamos el documento a una cubeta dependiendo de sus valores minhash (su firma).
- Todos los pares de documentos que caen en una misma cubeta son candidatos a pares similares. Generalmente tenemos mucho menos candidatos que el total de posibles pares.
- Checamos todos los candidatos calculando su similitud exacta para eliminar falsos positivos, para tener nuestra colección final de pares similares.

Veremos formas de diseñar las cubetas para obtener candidatos con la similitud que busquemos (por ejemplo, mayor a 0.5, mayor a 0.9, etc.). Primero veamos algunas posibilidades construidas a mano para lograr nuestro objetivo.

Ejemplo: todos los minhashes son iguales

Consideremos la siguiente matriz de firmas, con un mini-ejemplo:

```

textos <- c('el perro persigue al gato',
          'el perro persigue al gato.',
          'mi mascota es divertida',
          'mi mascota es divertida, más mi perro',
          'mi mascota preferida es divertida')

set.seed(834)

num_hashes <- 16

minhash <- minhash_generator(num_hashes)

corpus <- TextReuseCorpus(text = textos,
                           tokenizer = shingle_chars,
                           k = 3, lowercase = TRUE,
                           minhash_func = minhash,
                           keep_tokens = TRUE)

pairwise_compare(corpus, jaccard_similarity)

```

	doc-1	doc-2	doc-3	doc-4	doc-5
## doc-1	NA	0.9545455	0	0.07843137	0.0000000
## doc-2	NA	NA	0	0.07692308	0.0000000
## doc-3	NA	NA	NA	0.61764706	0.7000000
## doc-4	NA	NA	NA	NA	0.4883721
## doc-5	NA	NA	NA	NA	NA

```

mhashes <- minhashes(corpus) %>%
  lapply(function(x) x %% 97) # esto solo es para hacer el ejemplo
                                # más simple

df_firmas <- bind_rows(mhashes) %>%
  mutate(hash = paste0('h_', 1:num_hashes)) %>%
  gather(documento, minhash, -hash)

df_firmas

```

```

## # A tibble: 80 x 3
##   hash documento minhash
##   <chr> <chr>     <dbl>
## 1 h_1   doc-1     18.0
## 2 h_2   doc-1     14.0
## 3 h_3   doc-1     5.00
## 4 h_4   doc-1    79.0
## 5 h_5   doc-1    10.0
## 6 h_6   doc-1    81.0
## 7 h_7   doc-1    6.00
## 8 h_8   doc-1    50.0
## 9 h_9   doc-1    30.0
## 10 h_10  doc-1   34.0
## # ... with 70 more rows

```

Primero calculamos cada cubeta usando toda la firma:

```

firmas_colapsadas <-
  df_firmas %>%
  group_by(documento) %>%
  arrange(hash) %>%
  summarise(cubeta = paste(minhash, collapse = '-'))
firmas_colapsadas

## # A tibble: 5 x 2
##   documento cubeta
##   <chr>     <chr>
## 1 doc-1     18-34-11-61-42-10-81-51-14-5-79-10-81-6-50-30
## 2 doc-2     18-34-11-61-42-10-81-51-14-5-79-10-81-6-50-30
## 3 doc-3     77-74-40-53-54-75-79-7-15-10-13-19-38-7-34-20
## 4 doc-4     77-74-40-61-54-75-79-7-15-10-65-19-38-7-34-30
## 5 doc-5     43-13-40-53-54-75-79-66-73-10-78-60-38-7-34-20

```

Observación: veremos técnicas mejores (otra vez usando funciones hash) para evitar estos nombres de cubeta poco convenientes.

Y ahora agrupamos los documentos por cubeta:

```
cubetas_df <- firmas_colapsadas %>%
  group_by(cubeta) %>%
  summarise(docs = list(documento))

cubetas_df

## # A tibble: 4 x 2
##   cubeta           docs
##   <chr>        <list>
## 1 18-34-11-61-42-10-81-51-14-5-79-10-81-6-50-30 <chr [2]>
## 2 43-13-40-53-54-75-79-66-73-10-78-60-38-7-34-20 <chr [1]>
## 3 77-74-40-53-54-75-79-7-15-10-13-19-38-7-34-20 <chr [1]>
## 4 77-74-40-61-54-75-79-7-15-10-65-19-38-7-34-30 <chr [1]>

cubetas_lista <- cubetas_df$docs
names(cubetas_lista) <- cubetas_df$cubeta
cubetas_lista
cubetas

## $`18-34-11-61-42-10-81-51-14-5-79-10-81-6-50-30` → nombre de la cubeta
## [1] "doc-1" "doc-2"          (en este ejemplo es toda
##                                → elementos por
##                                cubeta)
## $`43-13-40-53-54-75-79-66-73-10-78-60-38-7-34-20`
## [1] "doc-5"
## 
## $`77-74-40-53-54-75-79-7-15-10-13-19-38-7-34-20` → nombre de la cubeta
## [1] "doc-3"
## 
## $`77-74-40-61-54-75-79-7-15-10-65-19-38-7-34-30` → nombre de la cubeta
## [1] "doc-4"
```

Y esto nos da un par de candidatos solamente, el documento 1 y 2 que sabemos que son muy similares. Podemos calcular la similitud de este par (exacta) para verificar. **No tuvimos que hacer todas las posibles comparaciones**, solamente agrupar los documentos en cubetas según su firma completa.

Observación: si la similitud de un par de documentos es igual a s , la probabilidad de que caigan en la misma cubeta es s^8 . Si s es muy cercano a 1, esta probabilidad es alta, pero es baja en otro caso (solo encontramos pares muy similares).

Ejemplo: algún minhash igual

Nótese que para que dos documentos caigan en la misma cubeta según el ejemplo anterior, la similitud realmente tiene que ser muy alta, pues todos los minhashes deben coincidir. ¿Qué pasa si queremos encontrar documentos con similitud de 0.3 o más, por ejemplo?

Si queremos similitud más baja de casi 1
Podríamos, por ejemplo, pedir que al menos un minhash coincida. Probamos:

```
cubetas_df <-  
  df_firmas %>% rowwise %>%  
  mutate(cubeta = paste(hash, minhash, sep = '-'))  
cubetas_df
```

```
## Source: local data frame [80 x 4]
## Groups: <by row>
##
## # A tibble: 80 x 4
##   hash   documento minhash cubeta
##   <chr>  <chr>      <dbl> <chr>
## 1 h_1    doc-1      18.0  h_1-18
## 2 h_2    doc-1      14.0  h_2-14
## 3 h_3    doc-1      5.00   h_3-5
## 4 h_4    doc-1     79.0   h_4-79
## 5 h_5    doc-1     10.0   h_5-10
## 6 h_6    doc-1     81.0   h_6-81
## 7 h_7    doc-1      6.00   h_7-6
## 8 h_8    doc-1     50.0   h_8-50
## 9 h_9    doc-1     30.0   h_9-30
## 10 h_10   doc-1    34.0   h_10-34
## # ... with 70 more rows

docs_agrupados <- cubetas_df %>%
  group_by(cubeta) %>%
  summarise(docs = list(documento))

## Warning: Grouping rowwise data frame strips rowwise nature

docs_agrupados
```

```
## # A tibble: 39 x 2
##   cubeta  docs
##   <chr>    <list>
## 1 h_1-18  <chr [2]>
## 2 h_1-43  <chr [1]>
## 3 h_1-77  <chr [2]>
## 4 h_10-13 <chr [1]>
## 5 h_10-34 <chr [2]>
## 6 h_10-74 <chr [2]>
## 7 h_11-11 <chr [2]>
## 8 h_11-40 <chr [3]>
## 9 h_12-53 <chr [2]>
## 10 h_12-61 <chr [3]>
## # ... with 29 more rows
```

```
cubetas_lista <- docs_agrupados$docs
names(cubetas_lista) <- docs_agrupados$cubeta
cubetas_lista <- keep(cubetas_lista, function(x) length(x) > 1)
```

Y ahora podemos extraer candidatos:

```
extraer_pares <- function(candidatos){
  candidatos %>%
    map(function(x) combn(sort(x), 2, simplify = FALSE)) %>%
    flatten %>%
    unique
}
cubetas_lista %>% extraer_pares()
```

```

## [[1]]
## [1] "doc-1" "doc-2"
##
## [[2]]
## [1] "doc-3" "doc-4"
##
## [[3]]
## [1] "doc-3" "doc-5"
##
## [[4]]
## [1] "doc-4" "doc-5"
##
## [[5]]
## [1] "doc-1" "doc-4"
##
## [[6]]
## [1] "doc-2" "doc-4"

```

si al menos un hash es igual, se

Y obtenemos 6 pares, que excluye a los que tienen similitud 0 o cercana a 0.

Observación: Si la similitud de un par de documentos es s , la probabilidad de que no coincidan en ningún hash es $(1 - s)^8$, así que la probabilidad de coinciden en al menos un hash es $1 - (1 - s)^8$. Si s es chico, entonces puede ser que esta probabilidad sea considerable de todas formas, así que capturamos pares de similitud baja.

Ejemplo: bandas de minhashes

Para poder tener un resultado intermedio, por ejemplo, que capture similitud mayor a 0.15, podríamos combinar los hashes en grupos. Si tenemos 4 grupos de 4 hashes cada uno, podemos pedir que en al menos uno de los 4 grupos todos los hashes interiores coincidan. Esto no es tan exigente como pedir que todos los 16 hashes coincidan, ni tan laxo como poder que al menos uno de los 16 hashes.

Hacemos 4 grupos de 4 hashes. Primero construimos el grupo y nombres para las cubetas individuales:

```

cubetas_df <-
  df_firmas %>%
  mutate(grupo = as.integer(substr(hash, 3)) - 1) %% 4) %>%
  mutate(grupo = paste0('g_', grupo)) %>%
  mutate(cubeta = paste(hash, minhash, sep = '-'))

cubetas_df

## # A tibble: 80 x 5
##   hash documento minhash grupo cubeta
##   <chr> <chr>     <dbl> <chr> <chr>
## 1 h_1   doc-1      18.0  g_0   h_1-18
## 2 h_2   doc-1      14.0  g_0   h_2-14
## 3 h_3   doc-1      5.00  g_0   h_3-5
## 4 h_4   doc-1      79.0  g_0   h_4-79
## 5 h_5   doc-1      10.0  g_1   h_5-10
## 6 h_6   doc-1      81.0  g_1   h_6-81
## 7 h_7   doc-1      6.00  g_1   h_7-6
## 8 h_8   doc-1      50.0  g_1   h_8-50
## 9 h_9   doc-1      30.0  g_2   h_9-30
## 10 h_10  doc-1     34.0  g_2   h_10-34
## # ... with 70 more rows

```

Y ahora agrupamos los 4 hashes dentro de cada grupo para formar una nueva cubeta:

```

cubetas_df <- cubetas_df %>%
  group_by(documento, grupo) %>%
  arrange(hash) %>%
  summarise(cubeta = paste(cubeta, collapse = '-')) %>%
  mutate(cubeta = paste(grupo, cubeta))

cubetas_df

```

```

## # A tibble: 20 x 3
## # Groups:   documento [5]
##   documento grupo cubeta
##   <chr>     <chr> <chr>
## 1 doc-1     g_0   g_0 h_1-18-h_2-14-h_3-5-h_4-79
## 2 doc-1     g_1   g_1 h_5-10-h_6-81-h_7-6-h_8-50
## 3 doc-1     g_2   g_2 h_10-34-h_11-11-h_12-61-h_9-30
## 4 doc-1     g_3   g_3 h_13-42-h_14-10-h_15-81-h_16-51
## 5 doc-2     g_0   g_0 h_1-18-h_2-14-h_3-5-h_4-79
## 6 doc-2     g_1   g_1 h_5-10-h_6-81-h_7-6-h_8-50
## 7 doc-2     g_2   g_2 h_10-34-h_11-11-h_12-61-h_9-30
## 8 doc-2     g_3   g_3 h_13-42-h_14-10-h_15-81-h_16-51
## 9 doc-3     g_0   g_0 h_1-77-h_2-15-h_3-10-h_4-13
## 10 doc-3    g_1   g_1 h_5-19-h_6-38-h_7-7-h_8-34
## 11 doc-3    g_2   g_2 h_10-74-h_11-40-h_12-53-h_9-20
## 12 doc-3    g_3   g_3 h_13-54-h_14-75-h_15-79-h_16-7
## 13 doc-4    g_0   g_0 h_1-77-h_2-15-h_3-10-h_4-65
## 14 doc-4    g_1   g_1 h_5-19-h_6-38-h_7-7-h_8-34
## 15 doc-4    g_2   g_2 h_10-74-h_11-40-h_12-61-h_9-30
## 16 doc-4    g_3   g_3 h_13-54-h_14-75-h_15-79-h_16-7
## 17 doc-5    g_0   g_0 h_1-43-h_2-73-h_3-10-h_4-78
## 18 doc-5    g_1   g_1 h_5-60-h_6-38-h_7-7-h_8-34
## 19 doc-5    g_2   g_2 h_10-13-h_11-40-h_12-53-h_9-20
## 20 doc-5    g_3   g_3 h_13-54-h_14-75-h_15-79-h_16-66

```

Observación: Nótese que cada cubeta está hecha de 4 hashes.

Ahora agrupamos por cubeta y vemos los pares resultantes:

```

docs_agrupados <- cubetas_df %>%
  group_by(cubeta) %>%
  summarise(docs = list(documento))
docs_agrupados$docs %>% keep(function(x) length(x) > 1) %>% extraer_pares

```

```
## [[1]]
## [1] "doc-1" "doc-2"
##
## [[2]]
## [1] "doc-3" "doc-4"
```

Y vemos que de esta forma obtenemos los dos documentos con similitud más alta. En la siguiente sección veremos con detalle esta técnica de hacer bandas de minhashes para filtrar pares similares por encima de algún umbral predefinido.

Tarea

1. (Ejercicio de (Leskovec, Rajaraman, and Ullman 2014)) Considera la siguiente matriz de tejas-documentos:

```
mat <- matrix(c(0,1,0,1,0,1,0,0,1,0,0,1,0,0,1,0,0,0,1,1,1,0,0,0),
               nrow = 6, byrow = TRUE)

colnames(mat) <- c('d_1','d_2','d_3','d_4')

rownames(mat) <- c(0,1,2,3,4,5)

mat

##      d_1 d_2 d_3 d_4
## 0    0   1   0   1
## 1    0   1   0   0
## 2    1   0   0   1
## 3    0   0   1   0
## 4    0   0   1   1
## 5    1   0   0   0
```

- Sin permutar esta matriz, calcula la matriz de firmas minhash usando las siguientes funciones hash: $h_1(x) = 2x + 1 \pmod{6}$, $h_2(x) = 3x + 2 \pmod{6}$,

$h_3(x) = 5x + 2 \pmod{6}$. Recuerda que $a \pmod{6}$ es el residuo que se obtiene al dividir a entre 6, por ejemplo $14 \pmod{6} = 2$, y usa la numeración de renglones comenzando en 0.

- Compara tu resultado usando el algoritmo por renglón que vimos en clase, y usando el algoritmo por columna (el mínimo hash de los números de renglón que tienen un 1).
 - ¿Cuál de estas funciones hash son verdaderas permutaciones?
 - ¿Qué tan cerca están las similitudes de Jaccard estimadas por minhash de las verdaderas similitudes?
2. Calcula la similitud de jaccard de las cadenas “Este es el ejemplo 1” y “Este es el ejemplo 2”, usando tejas de tamaño 3.

3. Funciones hash. Como vimos en clase, podemos directamente hacer hash de las tejas (que son cadenas de texto), en lugar de usar hashes de números enteros (número de renglón). Para lo siguiente, puedes usar la función `hash_string` del paquete `textreuse` (o usar la función `pyhash.murmur3_32` de la librería `pyhash`):

- Calcula valores hash de algunas cadenas como ‘a’, ‘Este es el ejemplo 1’, ‘Este es el ejemplo 2’.
- Calcula los valores hash para las tejas de tamaño 3 de ‘Este es el ejemplo 1’. ¿Cuántos valores obtienes?
- Calcula el valor minhash de la cadena anterior. Repite para la cadena ‘Este es el ejemplo 2’, y usa este minhash para estimar la similitud de jaccard (en general usamos más funciones minhash para tener una buena estimación, no solo una!).
- Para hacer en clase: repite usando 10 funciones minhash (puedes usar `minhash_generator` de `textreuse`, o usar distintas semillas para `pyhash.murmur3_32`).

References

Leskovec, Jure, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of Massive Datasets*. 2nd ed. New York, NY, USA: Cambridge University Press.

Mullen, Lincoln. 2016. *Textreuse: Detect Text Reuse and Document Similarity*. <https://CRAN.R-project.org/package=textreuse>.