Similaridad entre documentos

Gerard Otin
Felix Axel Gimeno
Ali Muhammad
15/11/2016
A-FIB-UPC

Índice

Introducción

En esta práctica implementaremos algoritmos para medir similaridades entre dos documentos o más. De los algoritmos implementados, se analizará la eficiencia, efectividad y los tiempos.

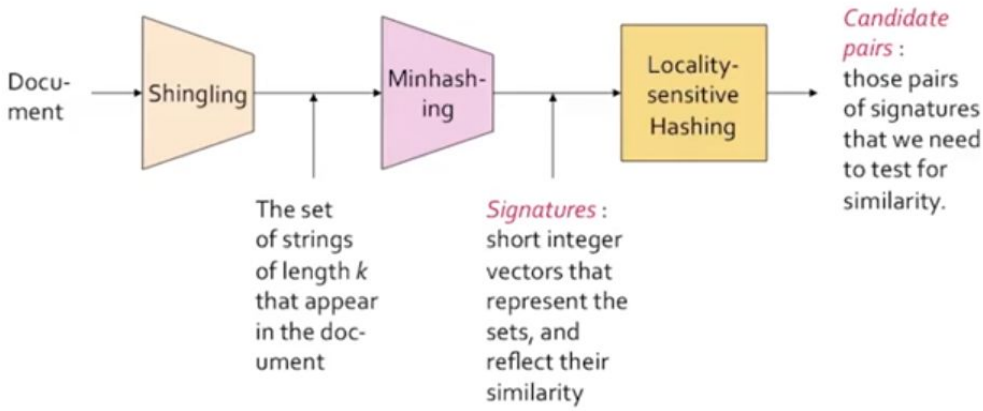
El programa calculará, en orden, k-shingles (n-gramas) de los documentos, la similaridad de Jaccard directamente a partir de los k-shingles, una signature matrix usando minhashing de los k-shingles, a partir de esta última obtenemos también la similaridad de Jaccard, y finalmente usamos Local-Sensitive Hashing para obtener una lista de documentos potencialmente similares.

Tal como se ha descrito en el párrafo anterior, tenemos dos formas de calcular la similaridad de los documentos, la definición de Jaccard para conjuntos, y la aproximación usando la signature matrix obtenida por minhashing.

La definición de similaridad de Jaccard entre dos conjuntos S y T es:

$$J(S,T) = \frac{|S \cap T|}{|S \cup T|}$$

En total el pipeline hasta el LSH sería:



Después de describir nuestra implementación hablaremos de los experimentos que hemos hecho.

Implementaciones

Generador de Pruebas

Tenemos un documento llamado **50words.txt**. Este documento contiene 50 palabras que representan una pequeña "base de datos de palabras". Estas palabras son las posibles palabras que pueden aparecer en cada uno de los documentos (permutaciones de palabras).

Para generar un documento, cogemos de forma random palabras de la "base de datos de palabras". Repetimos el proceso **ndocs** veces, ya que queremos **ndocs** documentos. Como **mínimo** hay **20** y con las permutaciones de las palabras de la base de datos.

Al final, acabamos obteniendo **ndocs** documentos con permutaciones de palabras de entre la "base de datos de palabras". Si se quiere ejecutar el programa main.cpp sin generar cada vez los documentos, se tiene que poner la variable *const int generation = true***; a false.**

En un principio, la base de datos (**50words.txt**), contiene 50 palabras de longitud media **3.4**. La variable **ndocs** está fijada a **20** y la variable **nwords** está fijada a 50. Esto quiere decir que generamos 20 documentos, con 50 palabras por documento.

Hemos generado también dos archivos (**50words_avglong8.txt** y **500words_avglong5.txt**) que contienen 50 y 500 palabras, de longitud media **8.48** y **5.4**, respectivamente.

Si queremos generar colecciones de documentos más grandes se tienen que cambiar la base de datos de palabras(con palabras de longitud más larga o corta), se puede cambiar también, el número de documentos a generar con las permutaciones de la base de datos. Para generar diferente colección de documentos se tiene que modificar en el archivo **50words.txt** y parámetros(**ndocs,nwords**) del **main.cpp**.

K-shingles

Dado un documento, lo pasamos a una string S, y consideramos únicamente palabras y espacios en blanco entre éstas. K-shingling consistiría en obtener un conjunto de strings donde cada substring de S de tamaño k (también contando espacios) está contenida.

No ha habido ningún problema relevante para hacer esto.
Nuestra implementación recorrer S, obtener sus substrings de tamaño k, y finalmente las insertamos en un ordered set, la función tiene de nombre "get_k_shingles_from_string"
El coste es O(k*n*log(n))

Similaridad de Jaccard

Versión directa Jaccard

Función: double jaccard_similarity(const k_shingles& A, const k_shingles& B);

La definición de la similaridad de Jaccard para dos conjuntos es la medida de la intersección dividido la medida de la unión. En nuestro caso queremos aplicarla a documentos expresados como k-shingles.

Implementación: Recorremos uno de los conjuntos de k-shingles y comprobamos si está en el otro conjunto, actualizamos unos contadores, si estaba en el otro conjunto lo borramos del segundo conjunto, cuando no queden más elementos en el primer conjunto, actualizamos los contadores con el tamaño actual del segundo conjunto.

El coste es O(n+n*log(n)+n*log(n))=O(n*log(n))

Versión aproximada de Jaccard usando minhashing

Signature matrix

Función: compute_signature_matrix

Implementación: Igual que la sección 3.3.5 del capítulo tres del libro [4]

Coste: O(r*(h + n*r)) donde n es el número de documentos, r el número de k-shingles distintos en total, h el número de funciones de hash

Calcular la similaridad de Jaccard a partir de la signature matrix

Función: double jaccard_similarity(size_t index1, size_t index2, signature_matrix& sm)

Implementación: Recorrer y comparar los elementos de dos columnas de sm

Coste: O(h) donde h es el número de filas de la matriz, o el número de funciones de hash

Generación de funciones de hash para minhashing

Funciones: get_hash_function , get_vector_of_hash_functions

Implementación: Generar números positivos aleatorios a b c d, con c primo >= d, entonces ((a*x+b mod c) mod d) es una función de hash (ver [6]), la d será el número de filas de la matriz característica La comprobación de que es primo lo hacemos con un test de primalidad de Miller determinista para primos acotados (ver [7]).

Coste: O(1) generar y comprobar, lineal recorrer hasta encontrar un primo.

Locality-Sensitive Hashing (LSH)

Función: lsh

Implementación: Seguimos el capítulo del libro, escogemos una banda de una columna,hacemos hash y lo guardamos, después vemos si ha habido coincidencias y actualizamos unos contadores, después si los contadores son suficientemente altos los añadimos a un set y os devolvemos.

Coste: el hashing cuesta O(n*c) donde n son las filas, c el número de columnas, guardar y acceder a los “buckets” es O(c*log(c)), actualizar los contadores es O(c*c*log(c)), filtrar la información es O(c*c*log(c)). En total es O(n*c + c*c*log c)

Generación de funciones de hash que tengan de entrada vectores

Funciones: get_hash_function_for_vectors , get_vector_of_hash_function_for_vectors

Implementación: La idea es usar un generador de números pseudoaleatorios que sea repetible, hemos escogido la familia “Linear congruential generator” ver [8], es decir, generamos constantes a b c y devolveremos (a*(a*(...)+b+x1)+b+x0) mod c donde x0 x1 ... son las entradas a la función de hash de vectores.

Coste: Generar 3 números aleatorios, constante crea una función de hash, que es lineal en el tamaño de la entrada

Nota: Tenemos una versión de código (main.cpp) antigua, dónde se usa otra forma para generar la funciones de hash. Todo la relativo a esta “variante” está en la carpeta “OLD_VERSION”

Experimentos

Ahora vamos a realizar experimentos con algunos parámetros que pueden modificar tiempo de cómputo, eficiencia y el grado de similaridad de los documentos.

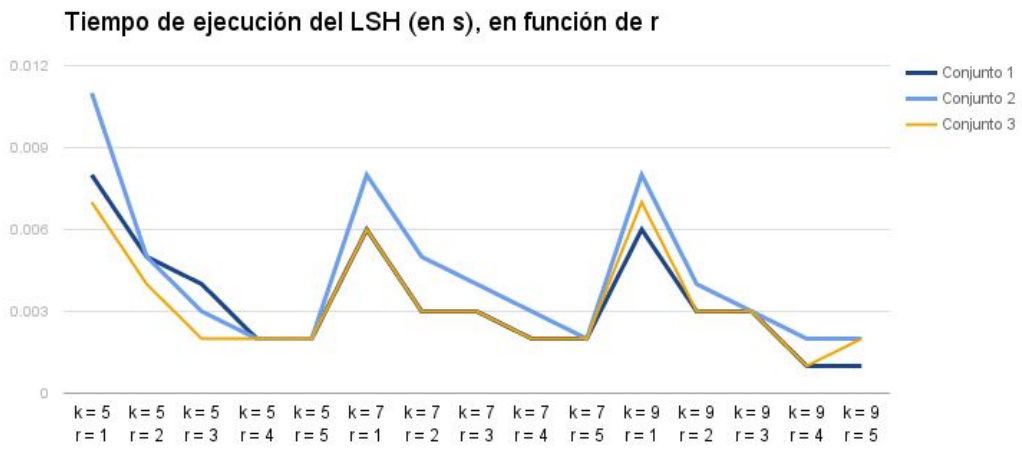
Vamos a realizar experimentos con 3 variables. La **"k"** que indica la longitud de los substrings de los k-shingles. La **"r"** indica el número de filas que se cogen de la signature matrix y que son hasheadas con la misma función de hash (usada para el cómputo del lsh). Finalmente, tenemos **"t" (threshold)** que es un umbral para considerar documentos como iguales.

Para los experimentos tenemos colecciones de documentos diferentes. En la colección 1 tenemos 20 documentos de longitud media por palabra de 3.4. En la colección 2 tenemos 50 palabras de longitud media 8.48 y en la colección 3 tenemos 500 palabras de longitud media 5.4.

Variabilidad del número de filas por banda (r)

La "r" indica de cuantas filas tiene que tener cada banda (para el cómputo del LSH). Vamos a ver si la variabilidad de r afecta al cálculo de la similaridad de documentos (en tiempo y resultados).
Para medir la r por separado, fijamos las demás variables k y t (t = 0.1)

Resultados: tiempo



Tiempo de ejecución del LSH en función de r para los 3 conjuntos de documentos.

Observamos que con r = 1 el efecto es igual que hacer minhashing. La tendencia es similar para los 3 conjuntos de documentos, pero varia un poco el tiempo ya que el número de palabras, comparaciones y cómputos varía según cada conjunto. El tiempo de ejecución va

bajando a medida que la r va aumentando, seguramente debido a que se procesan menos elementos. Cuanto mayor sea la r habrá menos probabilidad de colisión, ya que se hace un hash de varios k-shingles (mayor variabilidad) antes compararlo con otro documento.

Resultados: similaridad

Diferencias de los resultados del LSH (parejas de documentos candidatas a ser iguales) para cada k y cada r, fijando t = 0.1.

Cuando la r es 1, aumenta el coste computacional, ya que se hashea cada k-shingle por separado (el tamaño de la banda es de 1 fila) y se compara k-shingle a k-shingle.
Al tener que hacer más computo, hace que el tiempo de ejecucion tambien augmente(tal como se ha mostrado en el gráfico anterior). A medida que la r va aumentando, el número de funciones de hash aumenta, hay menos comparaciones y coincidencias, el tiempo de cómputo también baja porque hay que realizar menos comparaciones.

Creemos que el hecho de que haya menos coincidencias a mayor es debido a que los documentos generados con los que hemos hecho las pruebas son muy dispares entre sí, y al combinar más de un k-shingle para comparar la probabilidad de que sean iguales se reduce drásticamente.

Variabilidad del threshold (t)

La variable que controla el *threshold* o umbral de similaridad (t) es la que nos indica qué porcentaje de k-shingles respecto al documento entero tienen que ser iguales en dos documentos para ser considerados candidatos a ser similares.

Como para medir los efectos en el tiempo y resultados del threshold, son independientes de k y r, tomamos los valores medios del tiempo y resultados del LSH para todas las k's y r's.

La única afectación que tiene la variable t en el tiempo son las inserciones a una estructura de datos de tipo Set, por lo que obviamos la medida del tiempo en función de t.

Resultados: similaridad

Comparación de los resultados del LSH (número de parejas candidatas a ser iguales) para cada valor posible de t.
Gráfico obtenido realizando la media aritmética de t, independientemente de k y r.

.

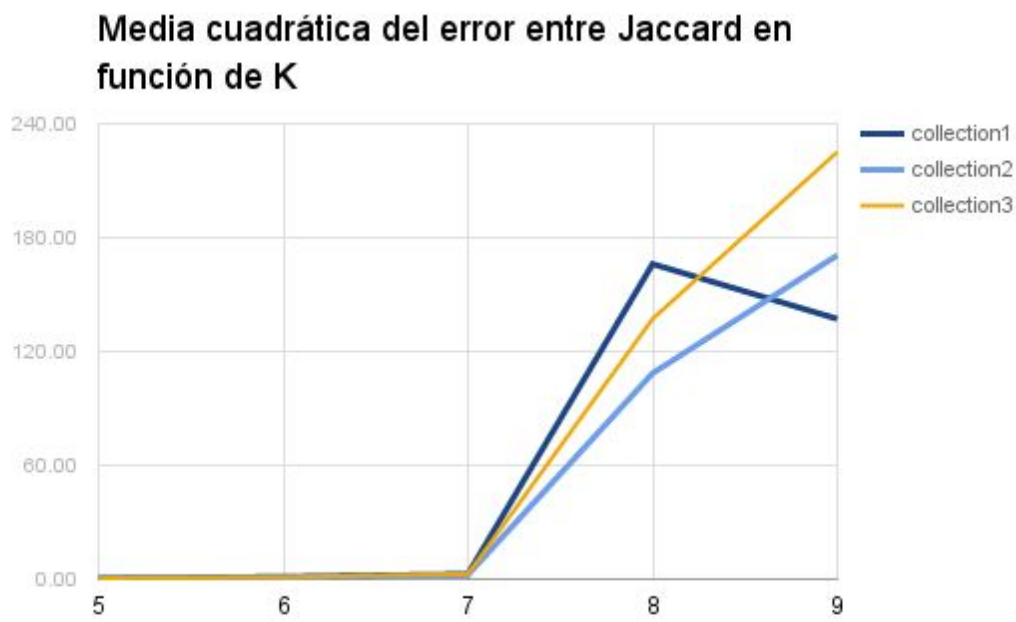
Variabilidad del tamaño de los k-shingles (k)

La k indica cuántos caracteres tiene que contener cada k-shingle.

Vamos a ver cómo afecta la k en el cálculo de la similaridad de los documentos, para ello mediremos una media del error entre Jaccard y su aproximación por minhashing, el coste temporal y el tamaño de salida del LSH.

Media del error entre calcular Jaccard directamente y aproximarlo por minhashing:

La media cuadrática de los errores relativos es :
sqrt(1/n * sum(e_i)) donde e_i = abs(js_sm - js)/js ,
Donde js es la similaridad de jaccard para la pareja i, y, js_sm es análoga por minhashing



Para cada k, media cuadrática del error entre la función de Jaccard (comparando el documento entero) y la función de Jaccard usando LSH (comparando solo un chunk).
Este gráfico nos sirve de aproximación para conocer la fiabilidad del LSH para cada k. A mayor error, menor fiabilidad del algoritmo.

Tal como podemos observar, la diferencia entre los errores se dispara cuando k es mayor que 7.

Si miramos las tendencias, el error crece conforme lo hace k, conjeturamos que linealmente (a comprobar en futuros experimentos).

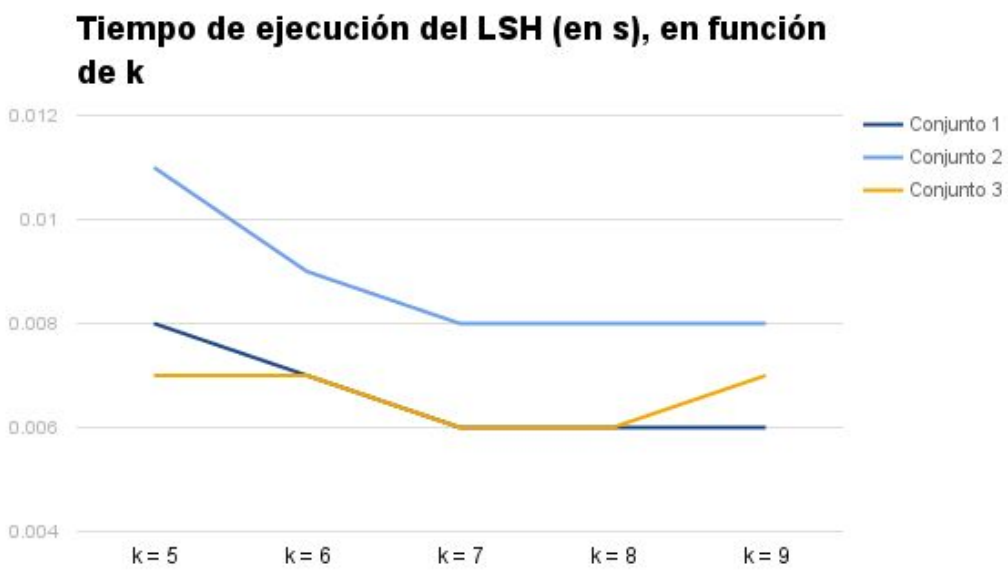
Nuestra conjetura es que este aumento del error proviene de que, al aumentar k, los posibles k-shingles aumentan (A^k diferentes, A es el tamaño del alfabeto), mientras que el número de k-shingles que describe un documento disminuye (n-k, n longitud documento), como los errores son aditivos y hacemos un error por cada k-shingle posible, que haya más k-shingles distintos implica mayor error.
Es decir, aumentar k implica aumentar las filas de la matriz característica (ver [4]), cómo el sampling (número de funciones de hash) se mantiene constante, durante el proceso de minhashing la aproximación/compresión de la matriz característica en signature matrix tendrá mayor pérdida de información.

Resultados: tiempo

Para hacer los **gráficos de tiempo de ejecucion (en segundos)** hemos hecho lo siguiente: Para observar qué efectos tiene la variación de la k únicamente, fijamos un valor bajo (que no restrinja los resultados) de los otros dos parámetros r y t (r = 1; t = 0.1).

Diferencia entre el tiempo de ejecución de las dos funciones de Jaccard (usando el documento entero, o por chunks dados por el LSH)

Tal como podemos ver en él gràfico, el cómputo de Jaccard sin usar LSH tarda mucho más que usando los resultados del LSH. Esto es debido a que en el cómputo del Jaccard “directo”, se compara todo el documento. En cambio, con el LSH se usan tablas de hash para determinar las franjas del documento sobre las que realizar las comparaciones. Jugando siempre con la probabilidad y variabilidad de los datos y los hashes se logran hacer menos comparaciones que aplicando el método de Jaccard “directo”, es decir, haciendo comparaciones dos a dos entre el/los documentos enteros.



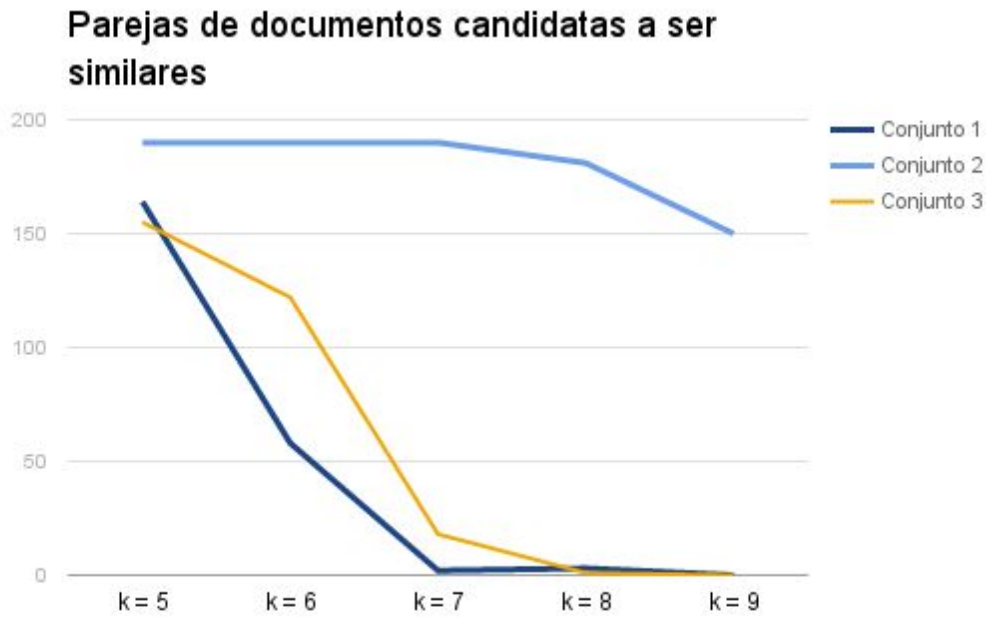
Tiempo de ejecución que tarda el LSH en las 3 colecciones de documentos que hemos usado para hacer los experimentos para cada k.

Vemos que a medida que aumenta la k, el tiempo de ejecución disminuye (hasta k = 8). Tal como podemos ver, está relacionada con el conjunto de documento que utilicemos. Depende de la longitud media de las palabras de cada documento. Cuanto menor es la k, más k-shingles hay, por lo que los cómputos a realizar aumentan y consecuentemente el tiempo de ejecución.

La cuestión es, entonces, encontrar un equilibrio óptimo entre el coste de computación para k grandes, y los beneficios que nos trae el usar k-shingles.

Observando los resultados, pensamos que k = 7 podría ser un buen candidato óptimo.

Resultados: Tamaño salida LSH



Resultado del LSH (número de parejas candidatas a ser iguales) para cada k, fijando r = 1 y t = 0.1.

En el gráfico podemos ver que a medida que aumenta la k, disminuye el número de parejas candidatas a ser iguales, debido a que es menos probable encontrar dos conjuntos grandes de letras iguales, ya que la generación de los documentos ha sido arbitraria.

No obstante, creemos que el caso del conjunto con longitud media de palabras elevada merece una profundización en estudios posteriores.

Conclusiones

- En k = 7 es el mínimo tiempo de ejecución que hemos encontrado (para el cómputo del LSH)
- El error relativo (diferencia entre diferencia de Jaccard por la signature matrix y por el documento) crece linealmente respecto k a partir de la k = 7.
- A partir de k = 7 las parejas de documentos candidatas a ser similares disminuyen drásticamente (excepto para el conjunto con una longitud media de palabra más grande).
- K = 7 es el valor ideal que hemos encontrado para realizar las mediciones con los documentos que hemos usado, ya que es el último valor entero de k donde se producen menos errores, donde el tiempo es menor y donde se producen mayores resultados de similaridad.
- Cuando la r es menor, se realizan más comparaciones. Es por esto que el tiempo de ejecución crece. En el caso de r = 1, es como hacer minhashing .
- A medida que la r aumenta hay menos parejas candidatas a ser similares (debido a que se reduce la probabilidad al mirar de r k-shingles en r k-shingles). También se reduce el tiempo de ejecución (hay menos elementos a comparar)
- Hemos comprobado que el threshold es muy restrictivo, ya que aumentando de 0.1 apenas produce algún resultado. Hemos comprobado que no es ningún error de implementación ya que sí detecta documentos iguales para threshold con valor 1 y menor. Por tanto, para comprobar la similaridad entre documentos, conviene mantener el threshold muy bajo.
- El threshold no produce variaciones significativas en el tiempo de ejecución; aunque hemos comprobado que a medida que t aumenta, disminuye de manera poco notable el tiempo de ejecución (por haber menos elementos a procesar).
- La longitud de las palabras de la BBDD hará cambiar los resultados de la similaridad (por la composición de los k-shingles)
- Usar LSH para luego aplicar Jaccard es una técnica sutil, pero hay que tener en cuenta que se están usando funciones de hash universales aleatorias. Si fuéramos puristas, sería preferible aplicar directamente la fórmula de Jaccard sobre los k-shingles de los documentos. No obstante, como no tenemos poder computacional para realizar el cálculo “puro” para muchos documentos, y documentos largos, podemos contar con la técnica de “filtrado” previa del LSH.

Bibliografía

[1]Video 1:Sobre k-shingles, Minhashing, Local sensitive Hash
https://www.youtube.com/watch?v=c6xK9WgRFhI

[2]Video 2:Sobre Jaccard_Similarty, Signature Matrix
https://www.youtube.com/watch?v=96WOGPUgMfw

[3]Video 3:Sobre LSH
https://www.youtube.com/watch?v=_1D35bN95Go

[4]Capítulo 3 del libro
http://infolab.stanford.edu/~ullman/mmds/ch3.pdf

[5] Libro Algoritmos Minhashing y LSH
Jure Leskovec, Anand Rajaraman and Jeffrey D. Ullman Mining of Massive Datasets Cambridge University Press

[6] Familias de hash para enteros
https://en.wikipedia.org/wiki/Universal_hashing#Hashing_integers

[7] Primos probables fuertes
http://primes.utm.edu/prove/prove2_3.html

[8] Generador de números pseudoaleatorios
https://en.wikipedia.org/wiki/Linear_congruential_generator