Detección de similitud de documentos

1er Cuatrimestre - Curso 2016/2017

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Mario Fernández Villalba
Juan Miguel de Haro Ruiz
Carlos Roldán Montaner

ÍNDICE

1. Identificación del problema

Desde tiempos inmemoriales las personas nos hemos aprovechado del éxito ajeno mediante las imitaciones y las copias. Actualmente, con la existencia de Internet, esta práctica ha sido extendida hasta límites insospechables. Es normal entonces de que se desarrollen herramientas para detectar copias. En esta práctica nos centraremos en algoritmos que detectan copias entre dos textos, y su implementación mediante tablas de hash.

2. El algoritmo

Antes de hablar del algoritmo en si, es necesario explicar un coeficiente fundamental en la semejanza de dos documentos: la similitud de Jaccard. Ésta nos indica la similitud que hay entre dos conjuntos A y B, y la definimos como:

$$S = \frac{|A \cap B|}{|A \cup B|}$$

Una vez dicho esto, dado un documento definimos los *k-shingles* del documento como las diferentes cadenas de caracteres de longitud *k* que podemos extraer de dicho documento.

Para saber el grado de similitud entre dos documentos crearemos un conjunto de *k-shingles* para cada uno y mediremos la similitud de Jaccard de estos dos conjuntos. Cabe destacar que la similitud vendrá dada en función de *k*, para un estudio sobre la influencia de *k* en la similitud ver el apartado 6.1.

3. Implementación del algoritmo mediante signaturas de *Minhash*

El algoritmo comentado en el apartado anterior tiene un grave problema en cuanto al coste espacial, ya que hemos de tener en memoria todos los *k-shingles* de cada documento y operar con cadenas de caràcteres es costoso. Una posible optimización es convertir cada *k-shingle* a un entero, por ejemplo de 4 bytes. De esta manera además de comprimir el espacio, las operaciones sobre los *k-shingles* como la comparación, tardarán un tiempo constante ya que se podrán realizar en una sola instrucción de la máquina donde se ejecute.

Pero aun así puede no ser suficiente cuando tenemos muchos textos demasiado grandes, ya que eso implica que habrá muchos *k-shingles* y cada uno ocupará 4 bytes.

Para intentar solucionar esto implementaremos una aproximación del cálculo de la similitud de Jaccard mediante una representación de los documentos con signaturas de *Minhash*. Para más información sobre las signaturas de *Minhash* mirar artículo del enunciado.

4. Implementación mediante LSH

A pesar de que la implementación mediante tablas de signaturas de *Minhash* reduce el espacio en memoria, como se ha comentado antes, documentos lo suficientemente grandes pueden incrementar la dimensionalidad del problema hasta llegar a un punto donde es difícil manejar los datos. Para reducir esta complejidad utilizaremos el *Local Sensitive Hashing* o *LSH*.

La idea básica de *LSH* es dada una matriz de signaturas de *Minhash*, dividir ésta en *b* bandas de *r* filas cada una, y una vez hecho ésto para cada banda, aplicar una función de hash al número formado por los r dígitos de cada columna, las cuales representan una porción de la signatura de *Minhash*. Entonces, los valores que hayan ido a parar al mismo recipiente de valores, a los cuales hemos aplicado la función de hash, nos indican los documentos potencialmente similares. Para una mejor explicación de esto mirar el artículo del enunciado.

En el apartado 6.3 hay un estudio detallado de esta técnica para sus diferentes parámetros.

5. Sets de datos

Antes de pasar a la parte experimental, explicamos a continuación los sets de datos utilizados en cada experimento:

- *texto50palabras*: Este es un conjunto de 20 textos obtenidos permutando las palabras de un texto base de 50 palabras del *Lorem Ipsum*, un texto de relleno escrito en latín.
- *juegodetronos*: Es un conjunto de 20 textos obtenidos a partir de un fragmento del libro *Juego de Tronos,* utilizando un algoritmo que se explicará posteriormente. EL texto ocupa 31.9 kB.
- *textoDummy:* Este conjunto contiene textos con palabras en inglés generados de manera aleatoria, el contenido no tiene sentido alguno. Estos textos se han generado con el mismo algoritmo que en el set *juegodetronos*. El texto ocupa 23 kB.

El algoritmo usado para generar los dos últimos conjuntos de textos funciona de la siguiente manera: a partir del texto original, el algoritmo realizará n pasos, donde n es el número de textos que queremos generar. En la iteración i, se realizarán i*inc + 10 cambios aleatorios en el documento original, donde inc es un parámetro que se establece al inicio del algoritmo, y la constante 10 es para que en la iteración 0 se generen 10 cambios, así se asegura que el primer texto será muy parecido al original. Para los cambios aleatorios, primero se escoge un número aleatorio L entre 1 y 6, entonces con una probabilidad del 90% se realizará un intercambio de L caracteres seguidos con otros L caracteres seguidos del texto, ambos escogidos aleatoriamente. Con el 10% de probabilidad restante, se colocará una palabra de longitud L en una zona del documento escogido aleatoriamente que contendrá cualquier carácter ASCII(0 a 127).
De esta manera los primeros textos serán más parecidos al original que los últimos.

6. Estudio experimental

6.1 Similitud de Jaccard

A la hora de calcular la similitud de Jaccard, como hemos mencionado en el apartado 2, podemos hacerlo de dos maneras, las cuales hemos implementado de la siguiente forma:

- **Versión sin hash:** Se encuentra implementada en la clase KShingleSet y guarda los *kshingles* en forma de strings de longitud k en la clase set de la std. Es decir, los *kshingles* están ordenados lexicográficamente, lo que implica que añadir un elemento en el conjunto tiene un tiempo *O(log(n)*k)* donde n es la cantidad de elementos en el conjunto. El multiplicador k se debe a que la comparación de strings se hace carácter a carácter.
- **Versión con hash:** Se encuentra implementada en la clase KShingleSetHashed. Utiliza la misma estructura de datos que en la otra versión pero guarda los *kshingles* en forma de números enteros positivos de 4 bytes. De esta manera insertar tendrá tiempo *O(log(n))* ya que la comparación de enteros es constante. Para convertir una palabra de k caracteres a un entero de 4 bytes hace lo siguiente: Se divide la palabra en grupos de 4 caracteres (puede haber un grupo de menos si k no es múltiplo de 4). Cada grupo se interpreta como un entero en ascii, se convierte usando el método de Horner y se suma al resultado. Para que las sumas consecutivas no den un número mayor que 4 bytes, en cada suma se guarda el módulo del resultado en base 2^{32} (el número máximo que ocupa 4 bytes).

6.1.1 Experimento

Observación	La similitud de Jaccard para dos sets de *kshingles* formados con la versión sin hash es diferente a la versión con hash para los mismos textos y k. Además la versión con hash tarda menos en hacer el cálculo y ocupa menos en memoria.
Planteamiento	Calcularemos la similitud de Jaccard para dos sets de *kshingles* con ambas versiones.
Hipótesis	La similitud de Jaccard calculada con sets de *kshingles* implementados con la versión con hash es la misma que la calculada con la versión sin hash, tarda menos y ocupa menos espacio.
Método	● Ejecutamos el experimento para cada uno de los sets de datos. ● Creamos dos set de *kshingles* con la versión con hash, uno para cada texto a comparar. ● Calculamos la similitud de Jaccard. ● Medimos el tiempo que ha tardado en crear ambos sets y en calcular la similitud y el espacio que ocupan ambos sets. ● Hacemos lo mismo para la versión sin hash. ● Haremos comparaciones del texto original con los otros 20 textos modificados del set de datos, en total 20 comparaciones.

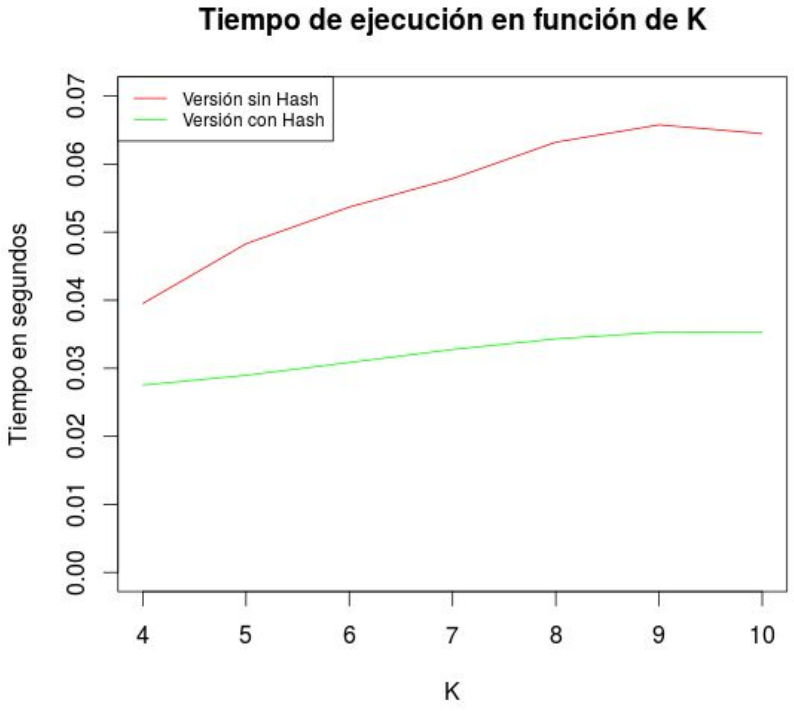

Figura 1: Tiempo de ejecución en función de k para una comparación del set juegodetronos

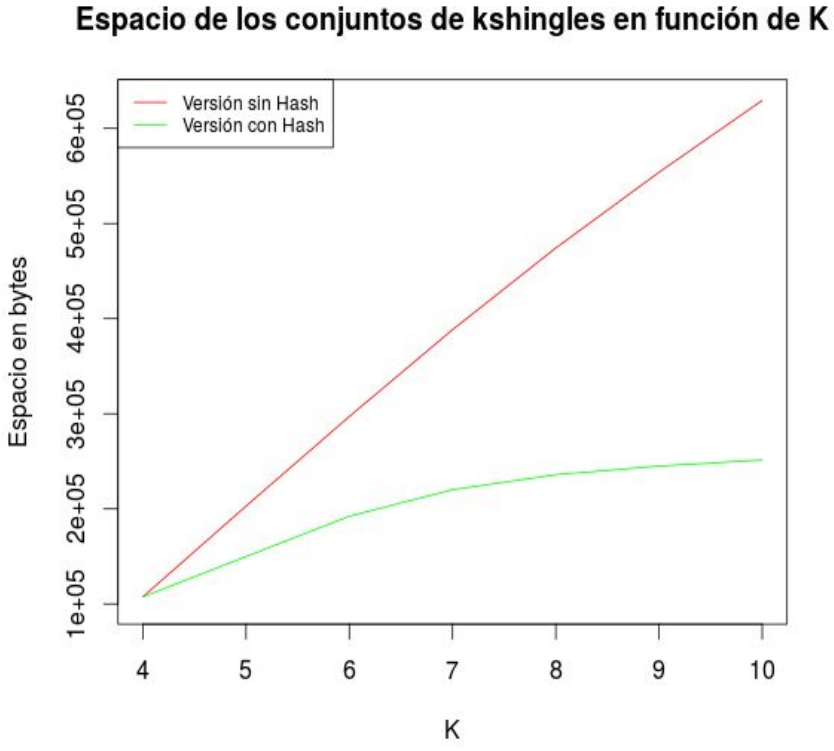

Figura 2: Espacio de los sets en función de k para una comparación del set juegodetronos

Como podemos observar en las *figuras 1 y 2*, tanto el tiempo de ejecución como el espacio siempre es menor en la versión con hash, como ya habíamos deducido anteriormente. También podemos apreciar un aumento del tiempo y espacio conforme aumenta la k, el motivo de esto podría ser el siguiente: En la versión sin hash, los *kshingles* ocupan cada vez más espacio porque aumenta el número de *kshingles* diferentes que puede existir(127^k combinaciones de caracteres) y porque su longitud es mayor. Por ese último motivo se tarda más en insertar los elementos en el conjunto, además al ser mayor la k, el tiempo de extraer cada subpalabra del texto también aumentará. En la otra versión, el espacio aumenta por el primer motivo explicado anteriormente. Aunque el número de *kshingles* sea limitado por usar 4 bytes, el número de elementos de un texto es mucho menor que el máximo que podemos tener (2^{32} posibles valores) por lo tanto tiene sentido que siga aumentando en función de k. Para justificar el tiempo, tenemos el mismo motivo de extraer las subpalabras pero hay que añadirle el tiempo de transformar esa subpalabra en un entero.

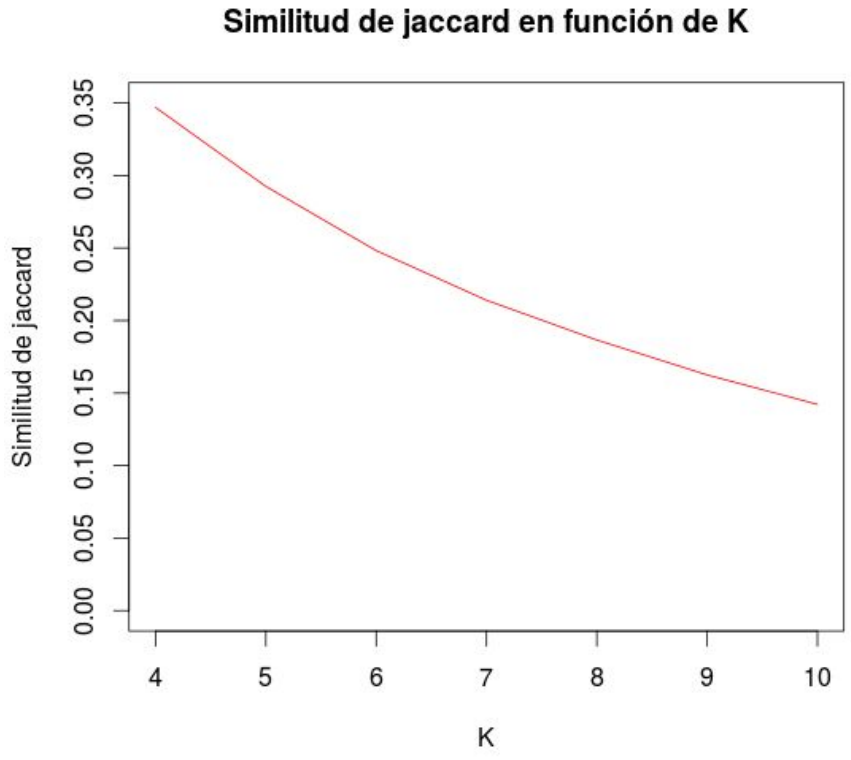

Figura 3: similitud de jaccard en función de k para una comparación del set juegodetronos

En cuanto a la similitud, como podemos ver en la figura 3 la similitud disminuye en función de k. Esto se debe al aumento del número de *kshingles*, que provocará que haya menos probabilidad de que exista el mismo en ambos textos.

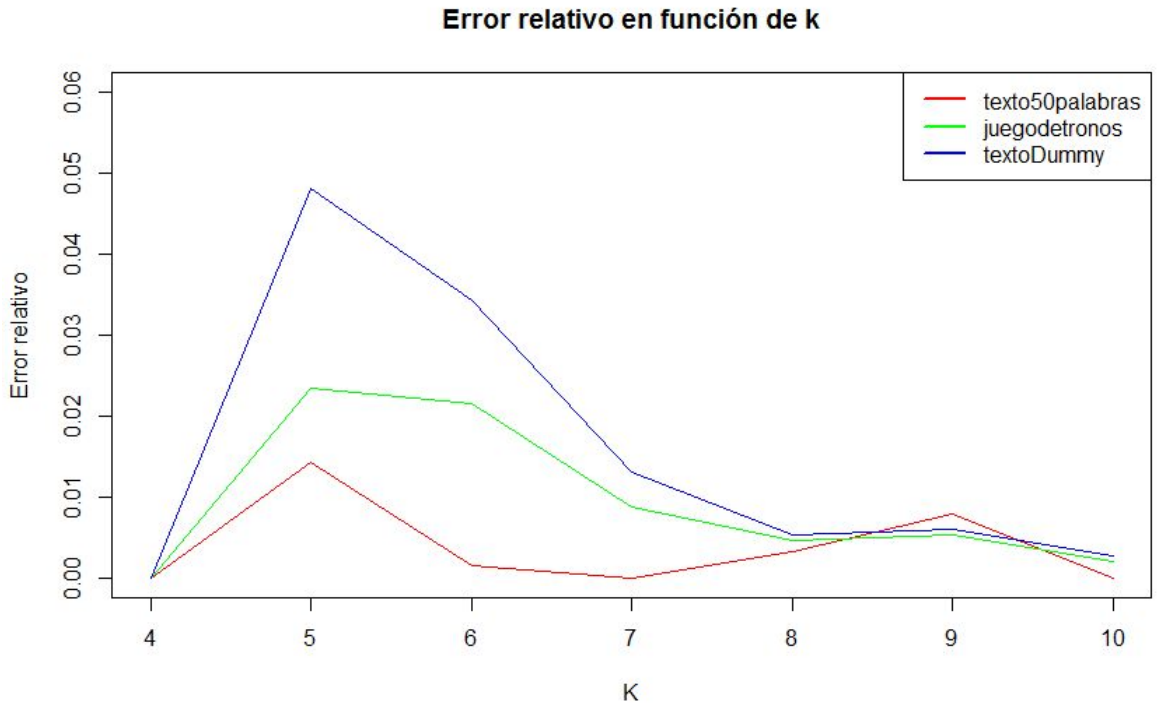

Figura 4: Medias de los errores relativos para los tres sets de datos en función de k

	texto50palabras	***textoDumnmy***	***juegodetronos***
Error relativo medio	0.003855998722	0.01561387626	0.009385614024

Figura 5: Medias de los errores relativos para los tres sets de datos

En cuanto a la aproximación de la similitud de jaccard conseguida por la versión de hash, podemos ver en la figura 5 que en todos los sets de datos el error relativo medio es muy pequeño. Concretamente, podemos ver una relación con el tamaño de conjuntos de *kshingles*, ya que cuantos menos hay, menos colisiones se producirán a la hora de convertir la subpalabra en número. Si miramos la evolución del error en función de k, en la figura 4 podemos observar una tendencia: con k bajas el error relativo es alto, en cambio se estabiliza con valores más altos, por lo tanto unos valores buenos a la hora de calcular la similitud con la versión con hash será a partir de 7.

6.1.2 Versión con hash mal implementada

La primera implementación de la versión con hash interpretaba la subpalabra de longitud k como un número de k dígitos y le aplicaba módulo 2^{32}. Esto implica que para cualquier valor, aplicar el módulo es equivalente a constuir el número con los 4 dígitos más bajos de la subpalabra. Es decir que para la palabra “hola” y “algohola” el resultado sería siempre el número formado por los valores ascii de “hola”. Por eso con k mayor que 5 nos daba la mayoría de las veces que tanto el número de *kshingles* como la similitud de jaccard era la misma, independiente de k. Esto se debe a que siempre utilizaremos los mismos *kshingles* formados por grupos de 4 dígitos, sin tener en cuenta el resto. Por eso decidimos mejorarlo usando un algoritmo en el cual todos los caracteres de la subpalabra influyeran en el resultado final, no solo los 4 últimos. De ahí salió la versión explicada al inicio de este apartado.

6.2 Minhash

No es factible realizar permutaciones de grandes matrices características, ya que el coste temporal sería demasiado elevado. No obstante, podemos simular estas permutaciones mediante la aplicación de funciones de hash aleatorias a la matriz característica, construyendo así la matriz de signaturas. Disponemos de 3 maneras distintas de construir la matriz de signaturas de minhash:

Hash:

Ésta es la manera que se indica en el enunciado. Aplicamos funciones de hash aleatorias de la forma ax + b y utilizamos como módulo el número de kshingles de todos los documentos.

Hash con el siguiente número primo:

Es similar a la manera anterior, con la excepción de que usamos como módulo el siguiente número primo mayor o igual al número de *kshingles*. El objetivo es tratar de reducir el número de colisiones que se producen y acercarnos más a las permutaciones absolutas.

Hash con módulo 2^32:

La manera de generar las signaturas es parecida a las anteriores, pero en este caso utilizamos como módulo 2^{32}, (el número máximo que puede representarse en 4 Bytes). El motivo es que utilizando este módulo en lugar del número de *kshingles* (o del siguiente primo), podemos ahorrarnos guardar los *kshingles* y la matriz característica, construyendo directamente la matriz de signaturas y consiguiendo así un gran ahorro espacial y temporal. La idea principal de este método es imaginarnos una matriz característica que contenga todos los *kshingles* posibles, que será 2^{32} siempre ya que convertimos las subpalabras de longitud k a enteros de 4 bytes. Entonces aplicamos la función de hash para permutar a las filas que tengan un 1 en alguna de sus columnas, que es equivalente a aplicar esta función a cada *kshingles* extraído en cada texto.

6.2.1 Experimento 1

Observación	Las distintas maneras de generar las signaturas de Minhash producen distintos resultados. Queremos ver cuál de las aproximaciones se acerca más a la similitud de Jaccard y también cuál es más eficiente en cuanto a tiempo y memoria.
Planteamiento	Calcularemos la aproximación a la similitud de Jaccard mediante *Minhash* de las tres maneras distintas con 3 sets de datos diferentes. Compararemos el error relativo en la aproximación, el tiempo y la memoria.
Hipótesis	Usar mod número de *kshingles* producirá una aproximación más acertada de la similitud de Jaccard ya que reducirá al máximo el número de colisiones. La versión con mod 2^{32} será la que utilice menos memoria.
Método	● Ejecutaremos el *Minhash* de las tres versiones distintas para cada set de datos. ● Para cada set de datos y cada versión del *Minhash*, comparamos el texto original con los otros 20 textos modificados, calculamos la aproximación a la similitud de Jaccard y medimos tiempo y espacio. ● Para cada set de datos y cada versión del *Minhash*, comparamos la aproximación obtenida para cada par de textos con la similitud de Jaccard real y calculamos el error relativo. ● Obtenemos las medias del error relativo para cada versión con cada set de datos y las comparamos. ● Para generar las funciones de hash aleatorias usamos la misma semilla para cada versión, es decir, los valores de a y b en la función a*x + b mod m serán los mismos.

	Hash	Hash Primo	Hash 2^{32}
Error relativo medio Texto Dummy	**0.2645756765**	**0.1568748224**	**0.08684742904**
Error relativo medio Texto Juego de Tronos	**0.08377827242**	**0.1660906451**	**0.06058939587**
Error relativo medio Texto 50 palabras	**0.4115914607**	**0.2433362701**	**0.0887065419**

Figura 6: Error relativo para los 3 sets de datos con las distintas de maneras de generar signaturas

Como podemos observar en la tabla, para los 3 sets de datos la media de error relativo más pequeña se produce en la versión con hash mod 2^{32}. Esto significa que ésta es la versión que da una aproximación más precisa a la similitud de Jaccard y que el hash con módulo primer primo igual o mayor al número de *kshingles* (a partir de ahora *HashP*) no es tan bueno como pensábamos.

	Hash	Hash Primo	Hash 2^{32}
Tiempo en segundos Texto Dummy	**5.52875**	**5.71312**	**1.86951**
Tiempo en segundos Texto Juego de Tronos	**7.25934**	**7.46266**	**2.57478**
Tiempo en segundos Texto 50 palabras	**0.163821**	**0.138656**	**0.0804698**

Figura 7: Tiempo para los 3 sets de datos con las distintas maneras de generar signaturas.

	Hash	Hash Primo	Hash 2^{32}
Memoria utilizada Texto Dummy	**1847336**	**1847336**	**23000**
Memoria utilizada Texto Juego de Tronos	**2634196**	**2634196**	**23000**
Memoria utilizada Texto 50 palabras	**51348**	**51348**	**23000**

Figura 8: Memoria utilizada para los 3 sets de datos con las distintas maneras de generar signaturas.

En cuanto a tiempos, vemos que *HashP* es ligeramente más lento que la versión normal. Esto es debido al tiempo que tarda en encontrar el número primo. La versión con hash mod 2^{32} es mucho mejor que las otras dos en cuanto a tiempo y espacio, debido a lo explicado al inicio del apartado.

Por tanto, ya que la versión con mod 2^{32} es mejor en tiempo y espacio y su aproximación a la similitud de Jaccard es más precisa, concluimos que ésta es la mejor versión.

Una vez hemos decidido que versión es más efectiva, queremos comprobar cómo afecta la variación del número de funciones de hash (t) a la precisión de la aproximación y al tiempo de ejecución. Para ello diseñamos el siguiente experimento:

6.2.2 Experimento 2

Observación	Variar el número de funciones de hash produce distintas aproximaciones a la similitud de Jaccard con diferentes grados de precisión.
Planteamiento	Calcularemos la aproximación a la similitud de Jaccard mediante *Minhash* usando funciones de hash con módulo 2^{32} para los 3 sets de datos. Repetiremos el experimento varias veces modificando el número de funciones de hash.
Hipótesis	A medida que aumentemos el número de funciones de hash la precisión aumentará también, pero el tiempo crecerá drásticamente..
Método	● Para cada set de datos, ejecutamos el *Minhash* con la versión que aplica funciones de hash con módulo 2^{32}. Empezamos con t = 100 y la vamos aumentando de 50 en 50 hasta 850. ● Para cada t, comparamos el texto original con los otros 20 textos modificados, calculamos la aproximación a la similitud de Jaccard y medimos tiempo y espacio. ● Para cada t, comparamos la aproximación obtenida para cada par de textos con la similitud de Jaccard real y calculamos el error relativo. ● Obtenemos las medias del error relativo para cada t y las comparamos.

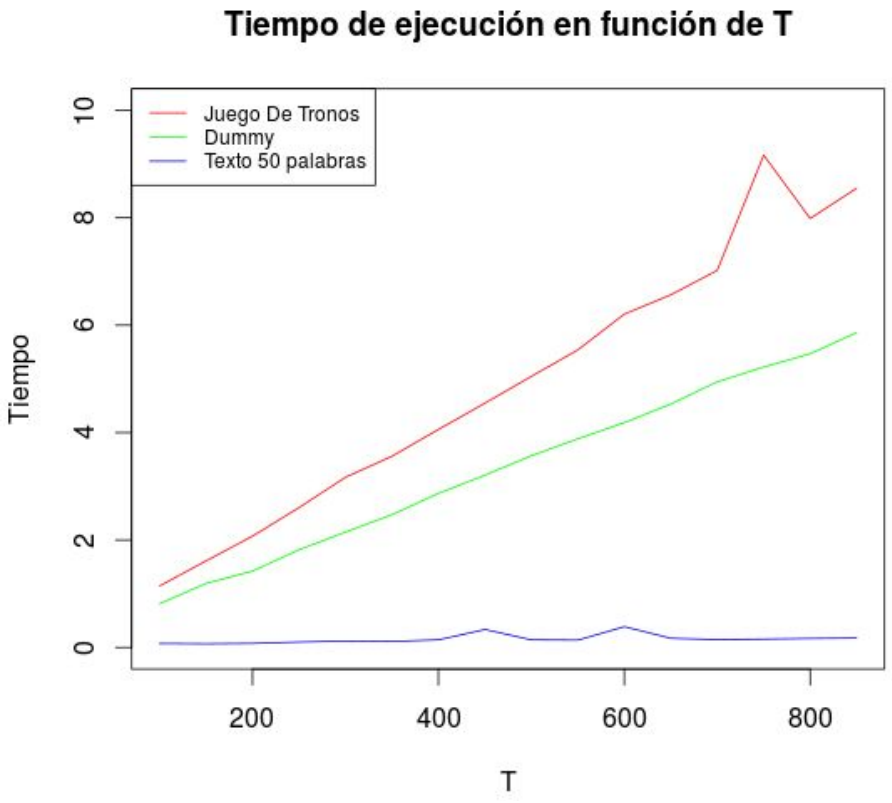

Figura 9: Tiempo en segundos de ejecución en función del número de funciones de Hash

En la gráfica anterior podemos observar que la función del tiempo depende de dos factores, t y el tamaño del documento. A medida que aumentan, también incrementa el tiempo. No obstante, vemos que el tamaño del documento tiene mucha más repercusión que el número de funciones de hash.

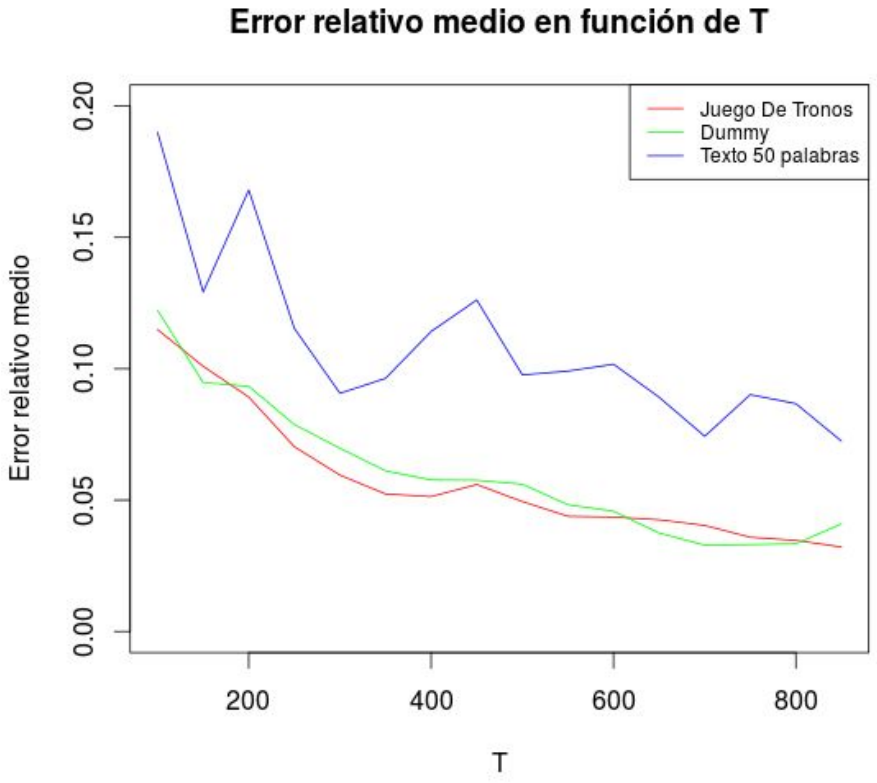

Figura 10: Error relativo medio en función de T

En la gráfica anterior podemos comprobar que a medida que aumenta el número de funciones de hash, el error relativo medio disminuye. No obstante, vemos que a partir de cierto punto la disminución del error es menos pronunciada y por tanto menos significativa, hasta el punto en que no vale la pena incrementar más t.

Concluimos que una t = 250 es suficiente ya que proporciona un error relativo medio inferior a 0.1 y además, a partir de t = 250 el error relativo medio empieza disminuir menos progresivamente.

6.2.3 Permutaciones absolutas

Intentamos realizar una versión que generaba signaturas realizando permutaciones absolutas de la matriz característica. No obstante, comprobamos que a la que el tamaño de ésta empezaba a aumentar, el coste temporal incrementaba de manera inadmisible.

6.3 Estudio de LSH

Como se ha comentado en el apartado 4, *LSH* separa la tabla de signaturas de minhash en *b* bandas de *r* filas cada una para posteriormente hashear en cada banda los valores de las columnas. Durante este proceso aparecen dos nuevos parámetros: *threshold* y *mod*.

Threshold es un valor entre 0 y 1, y nos indica la similitud mínima que han de tener dos documentos para que LSH los detecte como documentos potencialmente similares. A partir de este valor podemos conseguir los valores de *b* y *r* mediante la aproximación de *threshold* a $(1/b)^{(1/r)}$.

Mod es el valor del módulo utilizado en la función de hash para las columnas de cada banda.

Es natural cuestionarse entonces la influencia de estos parámetros sobre los resultados obtenidos por *LSH*. Para ello hemos diseñado dos experimentos. Operaremos en los dos experimentos con el set de datos *juegodetronos*. Esto es debido a que el set de datos *texto50palabras* contiene documentos con similitudes de Jaccard demasiado bajas para ser estudiadas y que *textoDummy* da resultados muy similares a *juegodetronos*.

En el primer experimento estudiaremos el comportamiento de *LSH* mirando diferentes valores de *threshold*. Para esto fijaremos el valor de *mod* a un número primo aleatorio para minimizar el número de colisiones.

En el segundo experimento estudiaremos el comportamiento de *LSH* para diferentes valores de *mod*.

6.3.1 Experimento 1

Observación	El parámetro *threshold* afecta a la calidad de la solución proporcionada por *LSH*.
Planteamiento	Ejecutamos el algoritmo *LSH* con diferentes valores de *threshold*.
Hipótesis	El parámetro *threshold* influye en la calidad de la solución (H_o) o no influye.
Método	● Elegimos el conjunto de textos *juegodetronos*. ● Creamos la tabla de signaturas de minhash con *k = 9, t = 250*. ● Escogemos un primo aleatorio para *mod* dentro del rango [100, 300]. ● Escogemos diferentes valores de *threshold* dentro del rango [0.5, 1]. ● Para cada valor de *threshold* calculamos la similitud de Jaccard real de cada pareja de documentos y guardamos las parejas tales que su similitud >= *threshold*. ● Para cada valor de *threshold* calculamos *b* y *r* de manera que *b*r = t* y que $(1/b)^{(1/r)}$ sea aproximadamente *threshold*. ● Para cada valor de *threshold* ejecutamos el algoritmo *LSH* y medimos los falsos positivos, los falsos negativos, el tiempo de ejecución y la similitud entre los conjuntos calculados por la similitud de Jaccard y *LSH* que contienen las parejas de documentos con similitud >= *threshold*.

THRESHOLD	B	R	FALSOS_POSITIVOS	FALSOS_NEGATVOS	SIMILTUD_SETS	TIEMPO (ms)	ESPACIO (B)
0.5	50	5	66	0	0.204819	1.28804	4864
0.6	25	10	25	5	0.166667	0.677411	2348
0.7	25	10	25	0	0.166667	0.605372	2340
0.8	25	10	17	0	0.15	0.649894	2260
0.9	10	25	10	0	0.0909091	0.413372	928

Figura 11: Resultados del primer experimento de LSH.

En la *Figura 11* podemos observar algunos resultados de este experimento. Debido a la gran cantidad de datos que hay pasaremos a hacer un estudio de las gráficas de diferentes parámetros en función de *threshold*.

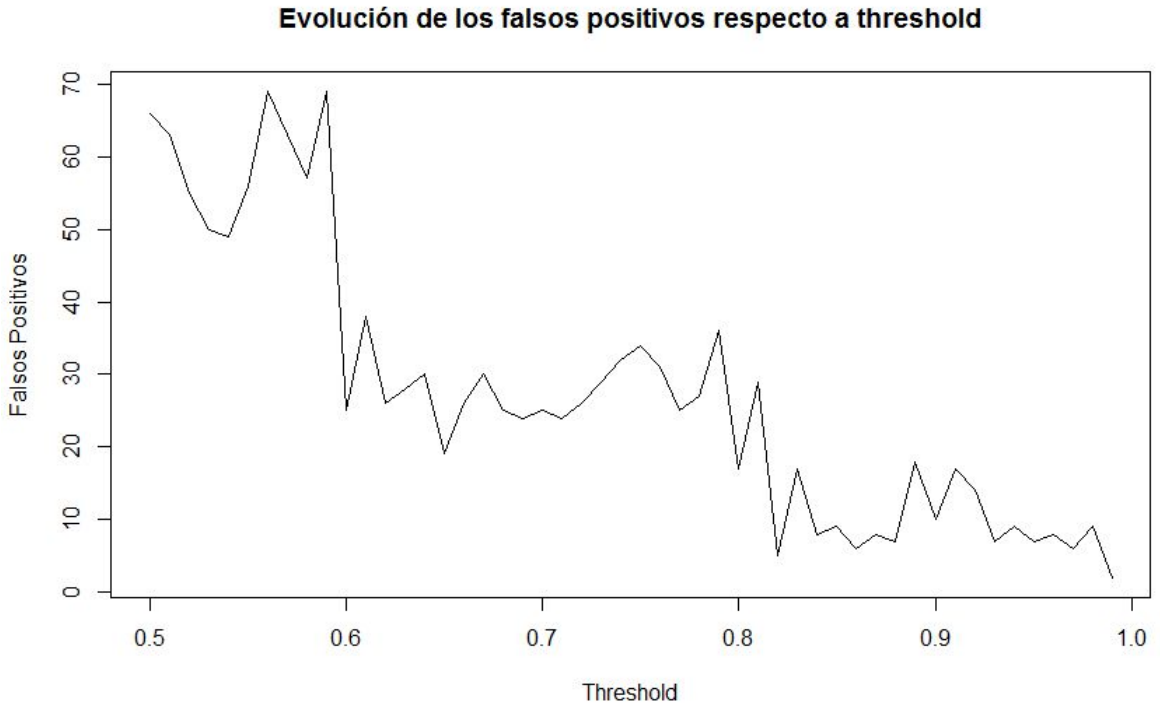

Figura 12: Evolución de los falsos positivos respecto a threshold

En la *Figura 12* observamos la evolución de los falsos positivos en función de *threshold*. Observamos que en general tienden a disminuir a medida que aumenta *threshold,* pero hay grandes bajadas de falsos positivos en los valores de *threshold* 0.6 y 0.82. Esto es debido a que en estos valores *b* y *r* cambian de manera que la aproximación $(1/b)^{(1/r)}$ es mayor que *threshold*, cosa que limita el número de falsos positivos.

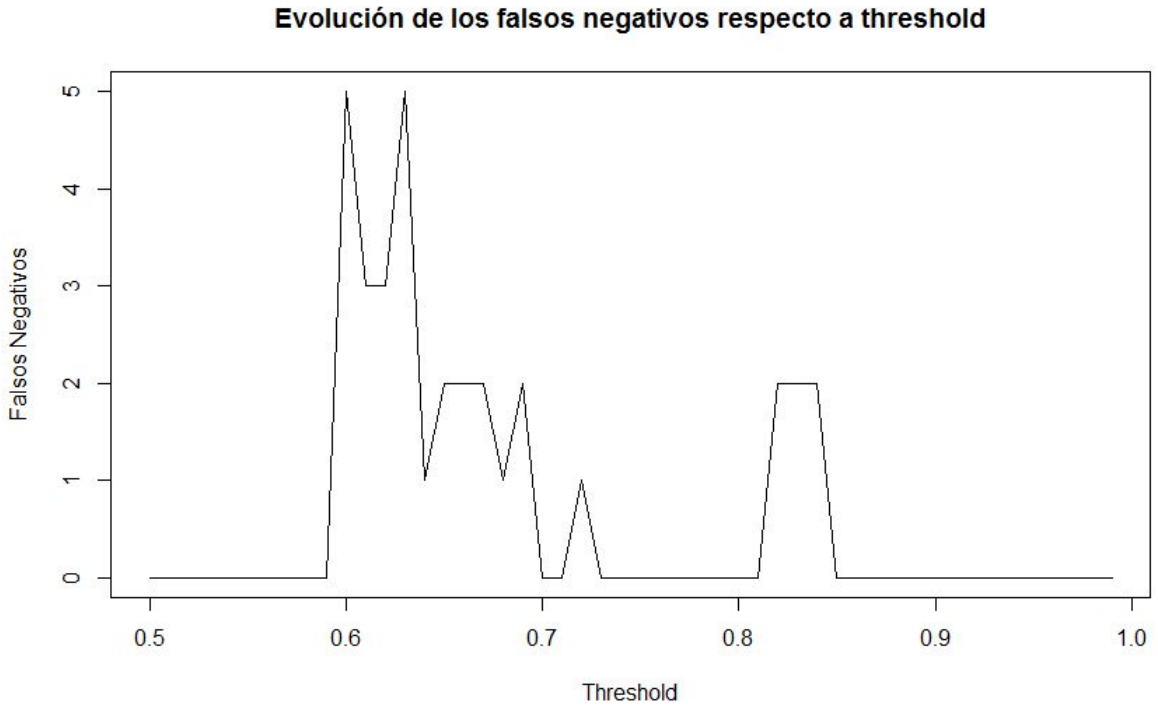

Figura 13: Evolución de los falsos negativos respecto a threshold

En la *Figura 13* observamos la evolución de los falsos negativos en función de *threshold*. Observamos que en general se mantienen constantes en 0, pero en los rangos de valores de *threshold* [0.6,0,72] y [0.81,0.85] aumentan. Esto es debido a que en estos valores *b* y *r* cambian de manera que la aproximación $(1/b)^{(1/r)}$ es menor que *threshold*, cosa que limita el número de falsos negativos.

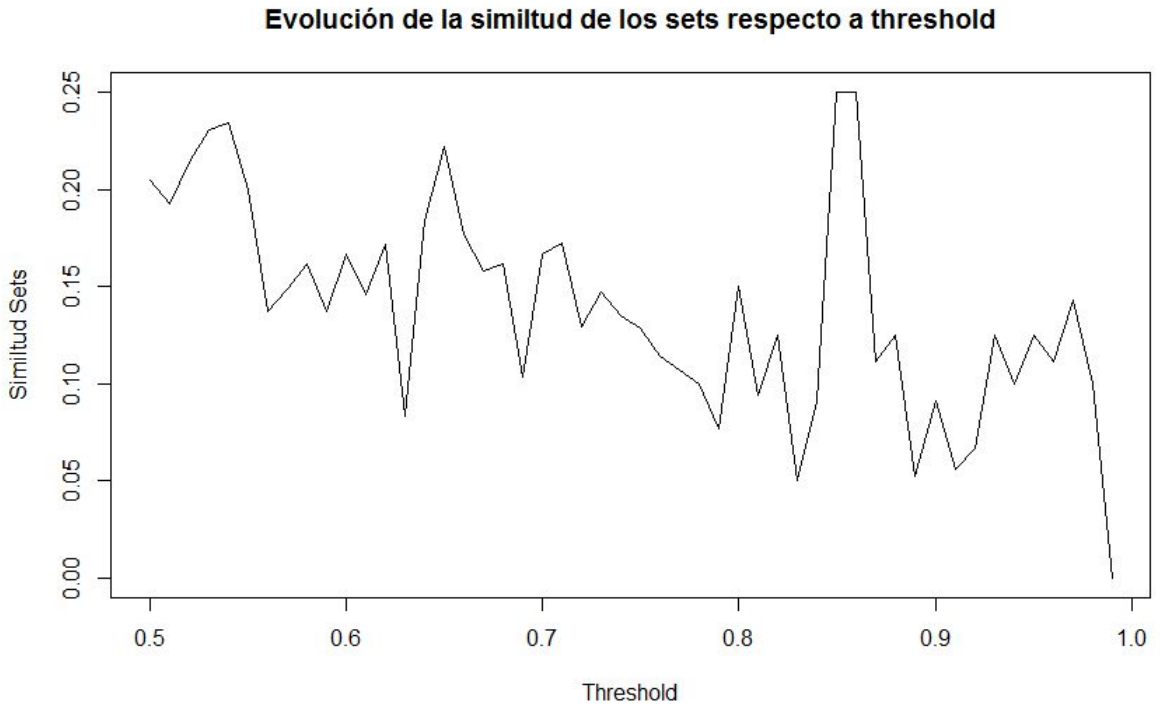

Figura 14: *Evolución de la similitud de los sets respecto a threshold*

En la *Figura 14* observamos la evolución de la similitud de los sets en función de *threshold*. Observamos una breve tendencia a disminuir, pero no es concluyente. Esto puede ser debido a que la similitud de los sets pueda ser influenciada por los falsos positivos.

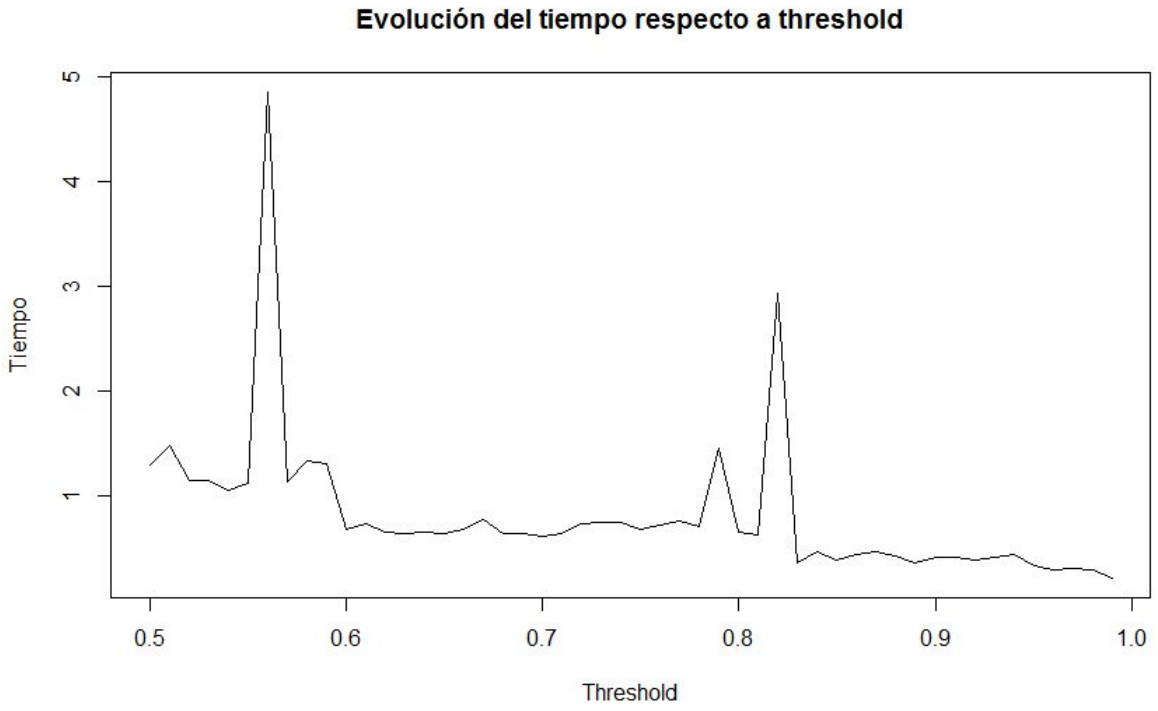

Figura 15: Evolución del tiempo respecto a threshold

En la *Figura 15* observamos la evolución del tiempo en función de *threshold*. Observamos que en general tiende a disminuir a excepción de algunos picos puntuales. Estos pequeños incrementos pueden ser debidos a irregularidades de *hardware* durante la ejecución del programa. La tendencia a disminuir es debida a que a medida que aumenta *threshold* disminuye el número de parejas potencialmente similares.

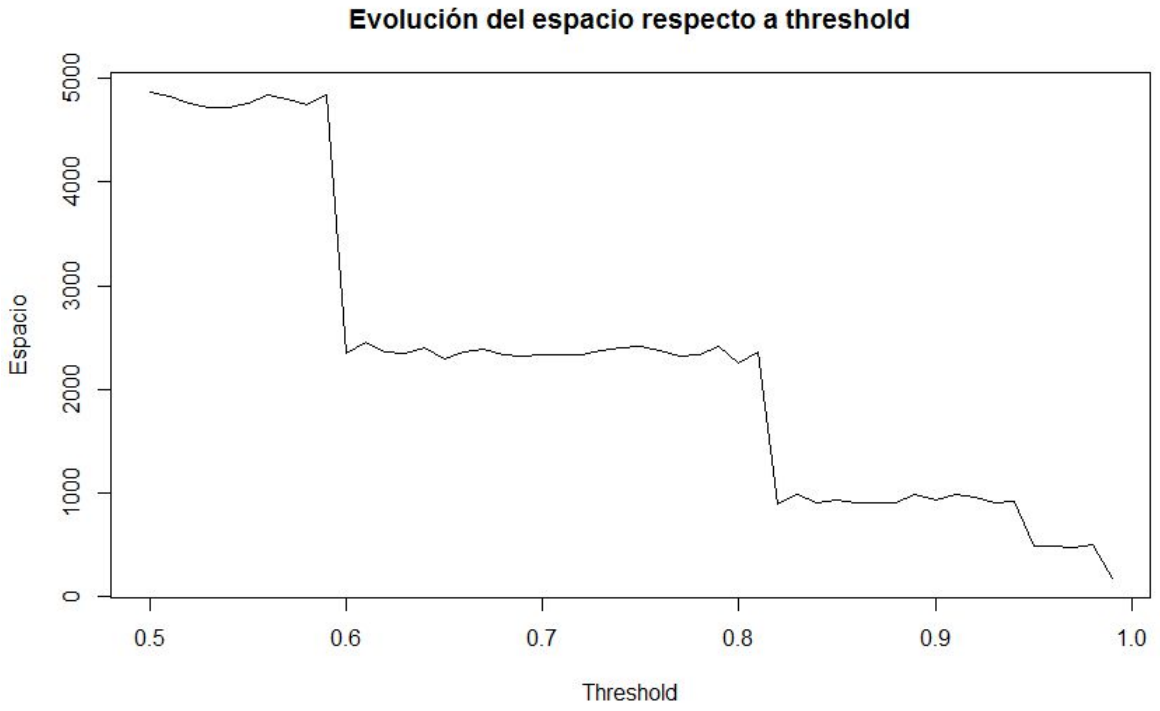

Figura 16: Evolución del espacio respecto a threshold

En la *Figura 16* observamos la evolución del espacio en función de *threshold*. Observamos que en general tiende a disminuir, y se observan grandes bajadas en los valores de *threshold* 0,6 y 0.82. Como hemos comentado antes, en estos valores cambian los valores de *b* y *r* de manera que hay menos valores a hashear.

6.3.2 Experimento 2

Observación	El parámetro *mod* afecta a la calidad de la solución proporcionada por *LSH*.
Planteamiento	Ejecutamos el algoritmo *LSH* con diferentes valores de *mod*.
Hipótesis	El parámetro *mod* influye en la calidad de la solución (H_o) o no influye.
Método	● Elegimos el conjunto de textos *juegodetronos*. ● Creamos la tabla de signaturas de minhash con *k = 9, t = 250*. ● Escogemos *threshold = 0.7*. ● Escogemos valores primos de *mod*. Estos valores seran los 100 primeros numeros primos empezando desde 13. ● Calculamos la similitud de Jaccard real de cada pareja de documentos y guardamos las parejas tales que su similitud >= *threshold*. ● Calculamos *b* y *r* de manera que *b*r = t* y que $(1/b)^{(1/r)}$ sea aproximadamente *threshold*. ● Para cada valor de *mod* ejecutamos el algoritmo *LSH* y medimos los falsos positivos, los falsos negativos, el tiempo de ejecución y la similitud entre los conjuntos calculados por la similitud de Jaccard y *LSH* que contienen las parejas de documentos con similitud >= *threshold*.

MOD	FALSOS_POSITIVOS	FALSOS_NEGATVOS	SIMILITUD_SETS	TIEMPO (ms)	ESPACIO (B)
13	172	0	0.0282486	0.491904	3516
101	35	0	0.125	0.507053	2420
311	22	0	0.185185	1.19714	2316
571	10	0	0.333333	1.33534	2220

Figura 17: Resultados del segundo experimento de LSH

En la *Figura 17* podemos observar algunos resultados de este experimento. Debido a la gran cantidad de datos que hay pasaremos a hacer un estudio de las gráficas de diferentes parámetros en función de *mod*.

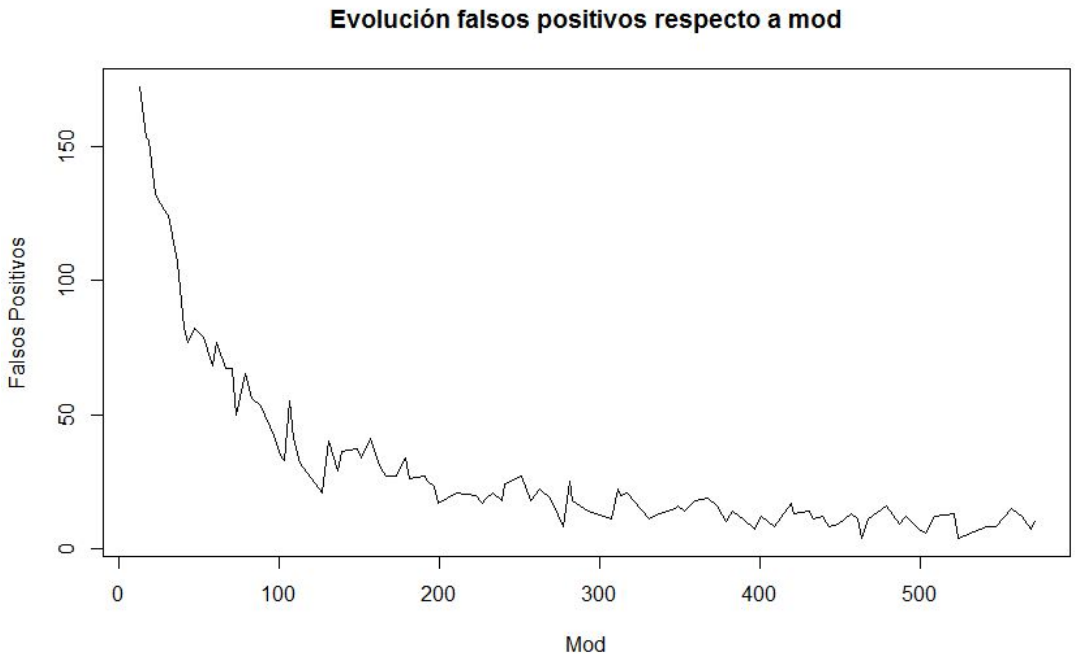

Figura 18: Evolución de los falsos positivos respecto a mod

En la *Figura 18* observamos la evolución de los falsos positivos en función de *mod*. Observamos que en general tienden a disminuir a medida que aumenta *mod* hasta llegar a un punto donde se mantienen constantes. Esto es debido a que al

aumentar *mod* aumentamos el espacio de valores y por lo tanto disminuimos el número de colisiones, en las cuales puede haber los falsos positivos.

No incluimos ninguna gráfica con falsos negativos debido a que en toda la ejecución siempre han sido 0.

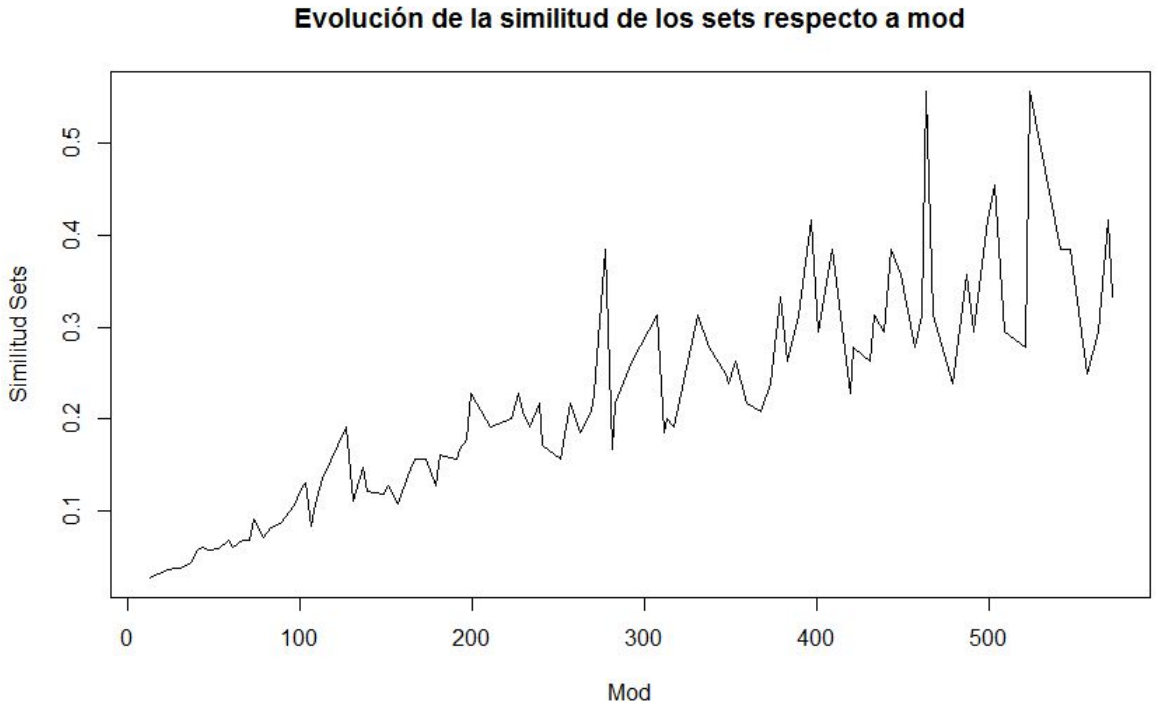

Figura 19: Evolución de la similitud de los sets respecto a mod

En la *Figura 19* observamos la evolución de la similitud de los sets en función de *mod*. Observamos que la similitud tiende a crecer a medida que lo hace *mod*. Esto es debido a que la similitud de los sets vesta influenciada por el número de falsos positivos, que como hemos comentado antes disminuyen a medida que lo hace *mod*.

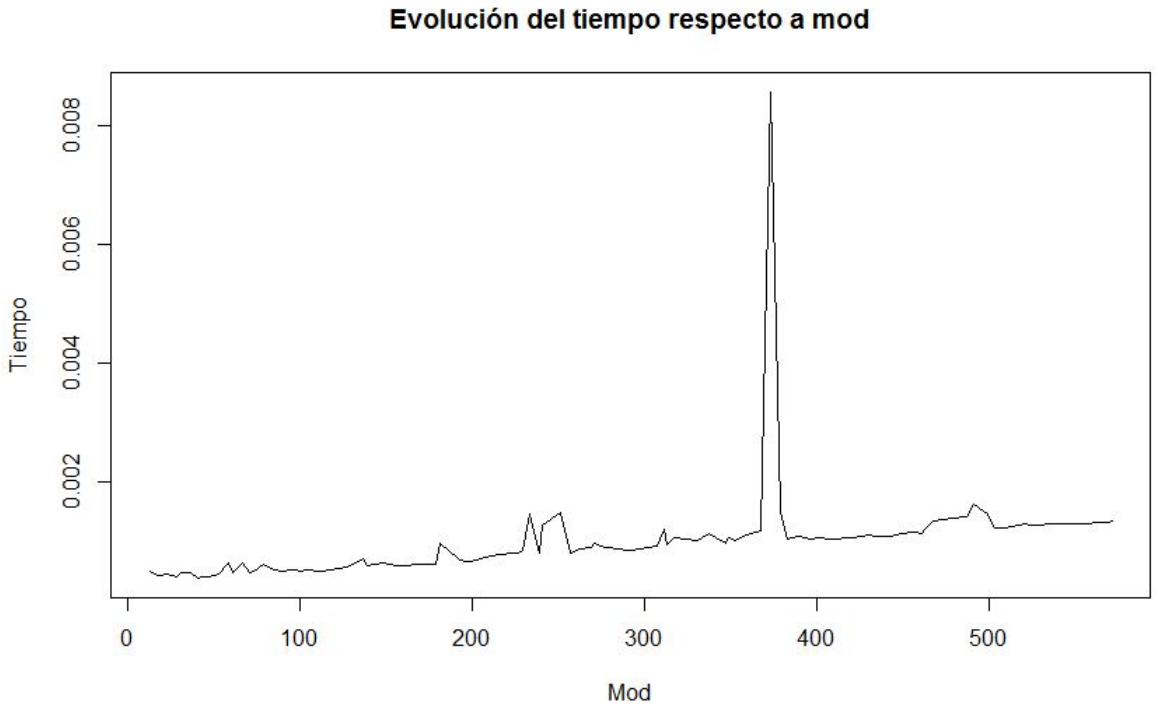

Figura 20: Evolución del tiempo respecto a mod

En la *Figura 20* observamos la evolución del tiempo en función de *mod*. Observamos que en general tiende a aumentar a excepción de un pico puntual. Este pico puede ser debido a irregularidades de *hardware* durante la ejecución del programa. La tendencia a aumentar es debida a que a medida que aumenta *mod* aumentamos el espacio de valores de las diferentes tablas de hash y tardamos más en recorrerlas.

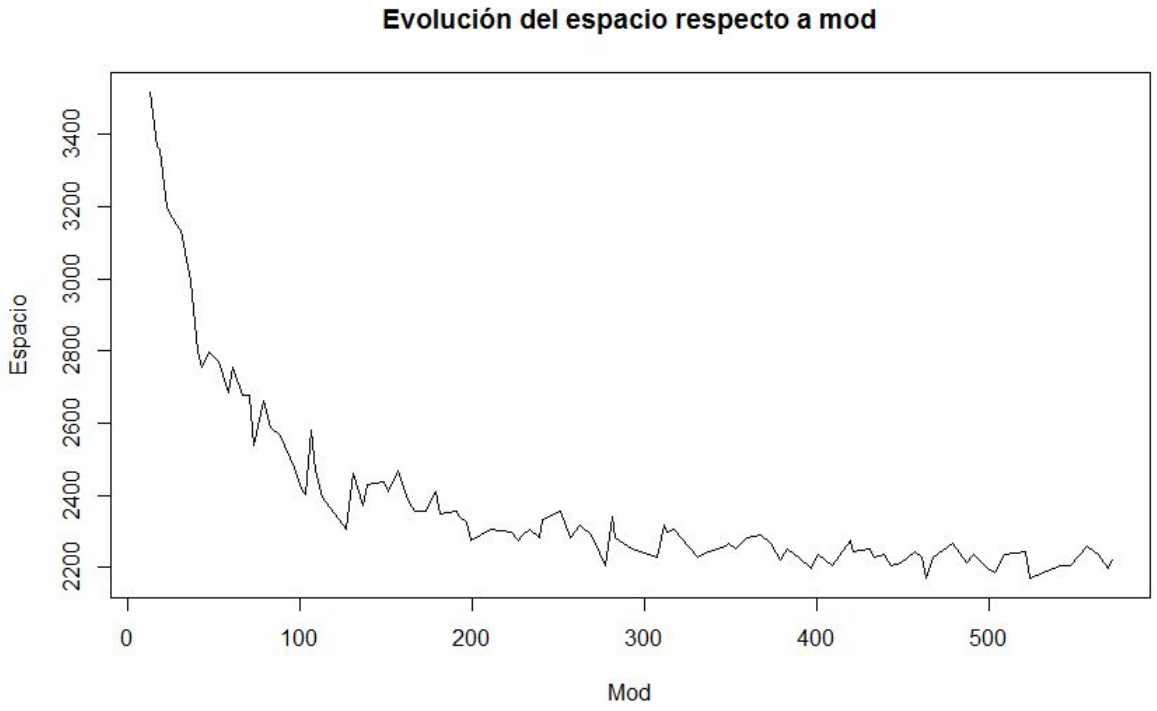

Figura 21: Evolución del espacio respecto a mod

En la *Figura 21* observamos la evolución del espacio en función de *mod*. Observamos que en general tiende a disminuir y finalmente mantenerse constante. Esto puede parecer contradictorio respecto al incremento de tiempo, pero podemos justificar la disminución del espacio porque este está ligado al número de falsos positivos, los cuales disminuyen con *mod*. Es más, si observamos la *Figura 18* podemos ver que se comporta de manera muy similar a esta figura.