Datos

Hashing

Introducción

Podemos pensar a las funciones de hashing como $h : \mathbb{U} \to \{0, 1, ..., m-1\}$, donde $\mathbb{U}$ es el espacio de claves (no necesariamente acotado).
Nos permiten trabajar con números en vez de strings, texto, imágenes, audio, etc. Son funciones muy rápidas y seguras.
Nos interesa que:

1. Que sea rápida.
2. Que no tenga muchas colisiones.

Funciones de hash no criptográficas

Son aquellas funciones que cumplen las siguientes propiedades:

- Deben ser muy eficientes de calcular.
- Deben producir la menor cantidad de colisiones posibles.

> Ejemplos: FNV, Jenkins, Murmur, Pearson.

Funciones de hash criptográficas

Son aquellas funciones que cumplen las siguientes propiedades:

- Dado $h(x)$ tiene que ser muy difícil hallar x.
- Tiene que ser muy difícil hallar x e y tal que $h(x) = h(y)$
- La función tiene que producir la menor cantidad de colisiones posibles
- Debe producir **efecto avalancha**. Esto significa que un cambio muy chico en x produce un cambio muy grande en $h(x)$.

> Ejemplos: MD5, SHA-256/512, Blake2

Como está implementado SHA-256

Construcción de Merkle-Damgard

Transforma una función de hash que recibe tamaños fijos a una que acepte tamaños variables. Para hacerlo parte el mensaje en bloques de tamaño fijo para pasarlas a las funciones de hash. Luego el último bloque que, en general, no es del tamaño esperado, es rellenado con un *padding* con la longitud del archivo. Luego, además de pasarle el bloque con el mensaje le pasa el estado de la función de compresión anterior. Se puede demostrar que si

las funciones de compresión son buenas y se utiliza un padding ocn la longitud del archivo, entonces la construcción Merkle-Damgard es resistente a colisiones.

Función de compresión

Se utiliza un algoritmo de encripción que recibe el hash anterior y el mensaje. El resultado de la encripción se junta con el hash anterior mediante una compuerta *XOR* para conseguir el resultado de la función de compresión. En SHA-256, el algoritmo de encripción que se utiliza es *Shacal-2*.

Hashing universal

H es una familia de funciones de hashing universal si se cumple que

$$\forall x, y \in \mathbb{U},\ h \in H,\ x \neq y$$

$$P(h(x) = h(y)) \leq \frac{1}{m}$$

Para valores numéricos

$$h(x) = (a.x + b(mod p))(mod m)$$

Con m el espacio de direcciones y p un número primo mayor o igual a m

> $a : \{1, 2, ..., p-1\},\ b : \{0, 1, ..., p-1\}$

Para claves de longitud fija

$$h(x) = \sum_{i=0}^{r} (a_i.x_i(mod p))(mod m)$$

Para claves de longitud variable

$$h(x) = h_{int}((\sum_{i=1}^{l} x_i.a^i)(mod p))$$

Donde p debe ser un número primo muy grande.

Hashing perfecto

Es una función de hash donde no hay colisiones y garantiza $O(1)$ en búsqueda.

FKS

Vamos a resolver colisiones buscando funciones de hash que diferencien las claves.
Lo mostramos con un ejemplo:

Intentamos almacenar $\{1, 5, 6, 9, 12, 13\}$ (como son 6 elementos entonces $m = 6$).
Elegimos la función $h(x) = x \ mod \ k$ con $k = 7$ (es primo mayor y más cercano a m).
Nuestra tabla quedaría algo como

Tabla	**0**	**1**	**2**	**3**	**4**	**5**	**6**
		1	9			5;12	6;13

Cuando hay colisiones de tamaño m_i elegimos un k_i primo más cercano a m_i^2 y elegimos $h_i(x)$ tal que no colisione. En nuestro caso tenemos dos colisiones de tamaño 2. Probamos con una nueva función de hashing $h_1(x) = x \ mod \ 5$.
Para la colision en la posición 5 tenemos la nueva tabla

Tabla	**0**	**1**	**2**	**3**	**4**	**5**
	5		12			

Haciendo lo mismo para la colision en la posición 6

Tabla	**0**	**1**	**2**	**3**	**4**	**5**
		6		13		

Si alguno de estas "segundas tablas" me sigue generando una colisión entonces hay que cambiar la función de hashing para esa tabla. Además para mantener la estructura de una tabla con otras tablas adentro podemos usar $h(x) = x \ mod \ 1$ para obtener la segunda tabla con una sola posición.

Entonces nuestra tabla orignial quedaría algo como

Tabla	**0**	**1**	**2**	**3**	**4**	**5**	**6**
		Tabla1	Tabla2			Tabla5	Tabla6

Las tablas con elementos que no colisionaron tienen la misma estructura:

Tabla1 ($x \ mod \ 1$)	**0**
	1

Tabla2 ($x \ mod \ 1$)	**0**
	9

Las tablas con elementos que colisionaron pueden tener distintas cantidades de elementos adentro y usar funciones de hashing diferentes (en este caso usan la misma):

Tabla5 ($x \ mod \ 7$)	**0**	**1**	**2**	**3**	**4**	**5**

Tabla5 ($x \ mod \ 7$)	0	1	2	3	4	5
	5		12			

Tabla6 ($x \ mod \ 7$)	0	1	2	3	4	5
		6		13		

Si al agregar nuevos elementos no se generan colisiones entonces podemos mantener la estructura. Si al agregar generamos una colisión entonces hay que rearmar la tabla.

Hashing perfecto y mínimo

Además de ser perfecto queremos acercarnos al óptimo en cuanto a espacio (no queremos dejar espacios vacíos en la tabla).

HDC

Necesitamos una familia del estilo $h(i, x)$ con $i = 0, 1, ..., m - 1$. Todos los slots de la tabla quedarán ocupados por una clave.
Tenemos m claves a almacenar, y tenemos un hash *parametrizable* $h(i, x)$. Los pasos a seguir son los siguientes:

1. Hasheamos todas las claves con $h(0, x)$ (va a haber colisiones).
2. Marcamos en un **bitmap** las posiciones ocupadas por una sola clave.
3. Partiendo del bucket con más colisiones, aumentamos i hasta que las claves se ubiquen en espacios vacíos, según el bitmap. Repetimos, nunca decrementando i, y actualizadno el bitmap.
4. En una tabla G anotamos los valores de i que se usaron para resolver x; si con $h(0, x)$ no hubo colisión almacenamos $-\mathbf{h(0, x)}$.

> Hay un video con un ejemplo explicado aquí.

Cuckoo hashing

Tenemos dos tablas con funciones de hash asociadas. Un dato siempre está en $h_1(x)$ o en $h_2(x)$ (tiempo constante). Ante una colisión, **la clave nueva echa la vieja** a la otra tabla. Si se genera un ciclo donde nunca se termina de hashear elementos entonces hay que rehashear todo en un espacio más grande.

Cuckoo hashing recargado

El problema de la versión orginal es que tiene un factor de carga muy bajo (hay que rehashear muchas veces). En vez de tener dos tablas distintas, podemos juntar las dos tablas en una y utilizar dos funciones de hashing distintas para simular. Además se puede utilizar un *stash* como alternativa para rehashear todos los elementos cuando encontramos un ciclo.

> Podemos encontrar un ejemplo aquí.

Hopscotch hashing

Queremos evitar rehashear tanto (comparado con Cuckoo) y evitar la mezcla de grupos sinónimos que tiene el sondeo lineal (esto pasaba cuando en una tabla de hashing normal tenías que hacer búsqueda lineal cuando no encontrabas el elemento en la posición del hash). Lo que no queremos es que si un elemento es hasheado a la posición 5 y esta misma está ocupada,
que no termine quedando en la posición 25 (muy alejado de la original). Es por esto que se introduce el concepto de *vecindario*.

Cada posición tendra asociada un vecindario de tamaño V. Si la posición $h(x)$ está ocupada, sólo puede ubicarse entre $[h(x),\ h(x)+V-1]$. El problema con esto es que se genera un solapamiento de vecindarios. Para resolver esto vamos a mantener un bitmap que nos diga cuáñes de las siguientes $V-1$ posiciones tiene elementos que hashean a h(x) (con esto nos evitamos tener que revisar elementos que no hashean a $h(x)$).
Si el vecindario se llena con elementos que deberían pertenecer al mismo entonces hay que rehashear. Si el vecindario se llena con elementos que **no** deberían pertenecer al mismo entonces vamos a tratar de reordenar la tabla para poder agregar el elemento nuevo. Los pasos a seguir si eso ocurre son los siguientes:

1. Buscamos la primera posición vacía.
2. Vamos hacia atrás viendo si podemos mover algún dato a esa posición, repetimos las veces que sea necesario.
3. Si no se puede, rehasheamos.

> Un ejemplo del reordenamiento se puede ver aquí.

Feature Hashing

Vamos a aplicar funciones de hash para reducir las dimensiones de nuestro set de datos. La idea es mantener las distancias entre los puntos.

Lema de Johnson-Lindenstrauss

Si tenemos n puntos en$\mathbb{R}^N$, y definimos un error ε con $0<\varepsilon<1$

$$\exists k = O(ln(n)) \wedge k > \frac{24ln(n)}{\varepsilon^2},\ f:\mathbb{R}^N \to \mathbb{R}^k$$

$$(1-\varepsilon)\|u-v\|^2 \leq \|f(u)-f(v)\|^2 \leq (1+\varepsilon)\|u-v\|^2,\ \forall u,v \in \mathbb{R}^N$$

> En otras palabras, este lema nos dice que podemos hallar una función f que nos reduzcade N a k dimensiones, con k en el orden de $ln(n)$ y una cota mínima. Esta función va a mantener las distancias parecidas. Que tan parecidas van a ser las nuevas distancias depende de ε.

Una matriz de proyección aleatoria puede ser una matriz A de tamaño $\mathbb{R}^{N\times k}$ tal que A_{ij} sea igual al valor aleatorio normal.
Proyectamos los datos:

$$XA = \hat{X}$$

$$X \in \mathbb{R}^{n\times N},\ A \in \mathbb{R}^{N\times k},\ \hat{X} \in \mathbb{R}^{n\times k}$$

Esta nueva matriz preserva aproximadamente las distancias ente los datos.

¿Que tipo de proyección vamos a usar en feature hashing?

Para la reducción de dimensiones vamos a utilizar una matriz aleatoria donde sólo un elemento por fila es 1 y el resto 0. Se puede probar que hashear los elementos de un vector y sumar las componentes que hashean al mismo lugar tiene el mismo efecto práctico que multiplicar por la matriz aleatoria. Un ejemplo de esto se puede ver aquí.

Podemos escribir x pasado por feature hashing como

$$\phi^{(h)}(x)$$

Las componentes del nuevo vector son

$$\phi_i^{(h)}(x) = \sum_{j:h(j)=i} x_j$$

Si agregamos **otra función de hash** que actúa de signo mejora los resultados

$$\xi : X \to \{-1, 1\}$$

$$\phi_i^{(h,\xi)}(x) = \sum_{j:h(j)=i} \xi(j)x_j$$

> Esta proyeccipon **sí** cumple el lema de Johnson-Lindenstrauss.

The hashing trick

El producto interno entre dos datos nos da un indicio de similaridad. En general es caro calcularlo cuando tenemos muchos puntos de dimensiones muy grandes. En este caso podemos usar feature hashing para calcular un producto interno aproximado. Si queremos que $k(x, y)$ sea nuestra función de producto interno aproximado entonces

$$k(x, y) = \phi(x) \cdot \phi(y)$$

LSH

Introducción

Locality sensitive hashing es una solución muy eficiente al problema de los vecinos más cercanos. En particular buscamos hallar dado un determinado punto cuales son los vecinos más cercanos con dada una determinada noción de distancia. La idea es que nosotros apliquemos una función de hashing con el objetivo de que los puntos que son parecidos (es decir, cercanos) colisionen. De esta forma, cada vez que nosotros busquemos algo obtendremos un bucket con todos los elementos que colisionan con el query.

Caracterización de LSH

- Queremos que nuestra función de hashing pueda procesar cualquier punto en R^d dimensiones y darnos como resultado el número de bucket.
- Queremos que cosas similares caigan en el mismo bucket.
- Queremos que cosas que son diferentes **NO** caigan en el mismo bucket.
- Queremos que se mantenga la complejidad de O(1) para búsqueda.

Hay 4 parámetros que van a caracterizar a un LSH:

1. d_1: La distancia máxima para considerar a dos elementos como similares.
2. d_2: La distancia mínima para considerar a dos elementos como distintos.
3. p_1: La probabilidad que dos elementos similares colisionen en el mismo bucket.
4. p_2: La probabilidad que dos elementos distintos colisionen en el mismo bucket.

Entonces una tenemos a la función $H(d_1, d_2, p_1, p_2)$ que representa una LSH.

> Naturalmente, $p_1 > p_2$ y además vamos a pedir que $d_1 = c.d_2$

Minhashing

Llamaremos a una función h minhash si cumple

$$P(h(x) = h(y)) = f(||x - y||)$$

> En otras palabras una función es un minhash si la probabilidad de que dos elementos colisionen es depende de la distancia entre los dos elementos.

Queremos que el minhash sea rápido de calcular y que sea parametrizable (osea que podemos definir una familia de minhashes).

Si tenemos una función

$$H(d_1, d_2, p_1, p_2)$$

Vamos a definir a d_1 y d_2 como distancias normalizadas entre 0 y 1. Definimos a p_1 y p_2 como

$$p_1 = 1 - d_1,\ p_2 = 1 - d_2$$

Vamos a asumir que la probabilidad de colisión depende linealmente de la distancia (como dicen las fórmulas de arriba). El problema es que nosotros no queremos una relación lineal, idealmente buscamos que si dos elementos son similares entonces la probabilidad de colisión sea 1 y que si son distintos entonces esa probabilidad sea 0 (una función escalonada).

Amplificación de familias LSH

Falsos positivos y falsos negativos

- Llamamos positivos a los casos que colisionan.
  - Falso positivo es cuando hay una colisión entre elementos no similares.
- Llamamos negativos a los casos que no colisionan.
  - Falso negativo es cuando no hay colisión entre dos elementos similares.

Recordemos que $p = 1 - d$ (osea que la probabilidad de colisión disminuye linealmente con la distancia). Esto significa que si nuestra $d_1 = 0,2$ entonces elementos que estén a $d = 0,15$ tienen un 15% de no colisionar (falso negativo). Lo mismo ocurre del otro lado, nuestro minhash no se comporta como nosotros queremos entonces vamos a tener algunos elementos con $d = 0,90$ que van a colisionar (falsos positivos).

Reducción de falsos positivos

Para reducir los falsos positivos vamos a pedir que los dos elementos colisionen en más de un minhash. Esto significa que si la probabilidad de que colisionen en un minhashes de p, entonces la probabilidad de que colisionen en dos a la vez es de $p.p$, ...

Si tenemos r minhashes entonces la probabilidad de que colisionen en todos es de p^r.

El problema con esto es que si somos más estrictos no solo vamos a disminuir la chance de colisión entre elementos lejanos sino también para los cercanos. Vamos a aumentar la cantidad de falsos negativos.

Reducción de falsos negativos

Para reducir la cantidad de falsos negativos vamos a pedir que dos elementos colisionen en algún hash. Esto significa que si tenemos b hashes, la probabilidad de *no* que colisionen en ninguno es de $(1 - p)^b$ (la probabilidad de colisión sería $1 - (1 - p)^b$).

Nuevamente, esto nos genera un problema porque si somos más laxos para evitar tener falsos negativos entonces vamos a terminar aumentando la cantidad de falsos positivos.

Combinando todo

Vamos a usar el **OR** de grupos de **AND** de minhashes.

$$M_i : "Los\ elementos\ colisionan\ en\ el\ minhash\ i"$$

$$(M_{11} \wedge M_{12} \wedge ... \wedge M_{1r}) \vee (M_{21} \wedge M_{22} \wedge ... \wedge M_{2r}) \vee ... \vee (M_{b1} \wedge M_{b2} \wedge ... \wedge M_{br})$$

La cantidad de minhashes que vamos a necesitar entonces es $b.r$.
La nueva probabilidad de que dos elementos a distancia d colisionen es de $1 - (1 - (1 - d)^r)^b$.

Esto significa que nuestra familia queda definida como

$$H(d_1, d_2, p_1, p_2)$$

con

$$p_1 = 1 - (1 - (1 - d_1)^r)^b$$

$$p_2 = 1 - (1 - (1 - d_2)^r)^b$$

Siendo b la cantidad de tablas de hashing o "bandas" a utilizar y r la cantidad de minhashes por cada tabla.

Es decir, nuestra funcion de minhash queda caracterizada por d_1, d_2, r y b.

Buscando r y b

Primero tenemos que decidir como vamos a definir la distancia entre dos elementos que vamos a hashear. Luego elegimos d_1, d_2, p_1 y p_2.

Luego vamos a realizar un Grid Search para buscar r y b tal que nuestros valores de p_1 y p_2 de las formulas de la sección anterior nos den menores a los que elegimos.
El método de Grid Search es una forma metódica de encontrarlos por fuerza bruta.

¿Cómo opera el LSH?

Vamos a tener 2 valores:
-b es la cantidad de "bandas" donde vamos a tener un bucket por cada posición en la banda.
-r es la cantidad de minhashes que van a poblar nuestras bandas.

Esto significa que vamos a tener $b.r$ funciones distintas de minhash (las llamaremos M_i).

Vamos a mostrar como opera el LSH con un ejemplo a pequeña escala para entender como funciona. En nuestro ejemplo $b = 2$ y $r = 2$, esto significa que tenemos 2 bandas y cada banda tiene 2 minhashes. Además el LSH va a tener 4 elementos dentro, estos son A, B, C y D ¿Cómo determinamos la estructura interna del LSH?

Vamos a suponer que nuestros minhashes devuelven números entre 0 y 3. Esto significa que nuestras bandas van a tener 4 buckets para que cada posible resultado de M_i tenga su respectivo bucket.

	0	1	2	3
B_1				

	0	1	2	3
B_2				

Ahora vamos a mostrar los resultados de todos los minhashes para cada elemento que tenemos dentro. Empecemos por los minhashes que fueron asignados a la banda B_1

B_1	M_1	M_2
A	1	1
B	0	0
C	1	3
D	0	0

Ahora para los de la banda B_2

B_2	M_3	M_4
A	0	2
B	1	0
C	1	0
D	1	0

Ahora vamos a poblar las bandas. Vamos a añadir a cada elemento en las posiciones que nos marcan los minhashes, si ambos devolvieron la misma posición entonces solo lo ponemos una vez en el bucket. Si primero agregamos los elementos a B_1 nos quedaría algo como

	0	1	2	3
B_1	$B,\ D$	$A,\ C$		C

De la misma manera agregamos los elementos a B_2

	0	1	2	3
B_2	$A,\ B,\ C,\ D$	$B,\ C,\ D$	A	

Ahora vemos que pasa si hacemos un query del elementl Q con valores de minhash

	M_1	M_2	M_3	M_4
Q	0	0	0	1

Por cada una de las bandas voy a obtener un conjunto de la intersección de todos los buckets a los que iría Q. Si Q_1 es el conjunto que consigo para la banda 1 entonces

$$Q_1 = \{B, D\}$$

De la misma manera, Q_2 sería

$$Q_2 = \{A, B, C, D\} \cap \{B, C, D\} = \{B, C, D\}$$

El resultado del query sería la unión de Q_1 y Q_2

$$Q_1 \cup Q_2 = \{B, C, D\}$$

Este ejemplo se puede extender a cualquier valor de r y b con cualquier cantidad de elementos.

Distancia de Jaccard

Definición de distancia y shingles

La distancia de Jaccard se usa para definir distancias entre conjuntos. Vamos a definir un minhash para la **similaridad de Jaccard** como

$$J(A, B) = \frac{|A \cap B|}{|A \cup B} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

La **distancia de Jaccard** va a quedar definida como $D_J(A, B) = 1 - J(A, B)$.

Un n-grama es una "ventana" de n caracteres que pasa por el documento y nos da un conjunto con todos los segmentos de n caracteres que se vieron.
Por ejemplo, el bigrama de la cadena "jaccard" sería: {"$j", "ja", "ac", "cc", "ca", "ar", "rd", "d$"}.
A cada elemento **único** del bigrama lo llamaremos shingle. Esto significa que podemos obtener un conjunto con todos los shingles de un documento de texto y compararlo con el de otro documento para obtener la distancia de Jaccard.

Construcción de minhashes

Vamos a crear una tabla donde las columnas son los documentos y las filas los shingles. En cada posición un 1 implica que ese shingle se encuentra en el documento y un 0 lo opuesto.

	d_1	d_2	d_3	d_4
s_1	1	0	1	0

	d_1	d_2	d_3	d_4
s_2	1	0	0	1
s_3	0	1	0	1
s_4	0	1	0	1

> No es necesario constriur la tabla a la hora de implementar el minhash.

Vamos a reordenar las filaz al azar y el minhash de cada documento es la posición en la que aparece el primer 1. Cada permutación de filas diferente va a ser un minhash nuevo en la familia de funciones.

M_1	d_1	d_2	d_3	d_4
s_1	1	0	1	0
s_2	1	0	0	1
s_4	0	1	0	1
s_3	0	1	0	1

Entonces los valores de $M_1(d)$ para cada documento serían

	$M_1(d)$
d_1	0
d_2	2
d_3	0
d_4	1

> Una propiedad que se puede demostrar de este minhash es que la probabilidad de que dos documentos coincidan es igual a la semejanza de Jaccard para ambos documentos.

A la hora de implementar la función de minhash, no vamos a usar este método de elegir la primera fila con un 1. Cada funcion M_i tendrá asignada un hash H_i y el resultado del minhash será el mínimo resultado que obtengo al hashear todos los shingles del documento. Para ponerlo de otra forma, sea N_d el conjunto de todos los shingles de un documento d,

$$M_i(d) = min(\{H_i(s) / \forall s \in N_d\})$$

> Por más que no sea lo mismo, esta implementación del minhash tiene las mismas propiedades que la explicada anteriormente.

Distancia angular

Concepto y semejanza

Es una métrica muy popular para cuando los datos que tenemos se representan como un vector. Estos vectores pueden llegar a ser de muchas dimensiones. La idea es que cuanto menor es el ángulo, más cercanos son los vectores.
Vamos a utilizar al coseno como medida de semejanza.

$$cos(\theta) = \frac{X \cdot Y}{|X||Y|}$$

Y como vamos a normalizar a todos los vectores

$$cos(\theta) = X \cdot Y$$

Como todos los vectores con los que vamos a trabajar tendrán norma 1 esto significa que vamos a estar trabajando en la hiperesfera unitaria.

> Una hiperesfera es una generalización de la esfera a un espacio de una determinada dimensión arbitraria.

Cálculo minhash y método de hiperplanos

AGREGAR COSAS CON LO EXPLICADO EN https://youtu.be/HeHsEgHQ3oE?list=PLeo_qKwGPZYfzT8fm2I7U-i0LXyR-olH8&t=765
Para calcular el minhash entre dos vectores primero utilizaremos una función auxiliar m_i que calcula el producto interno entre el elemento y un vector aleatorio y devuelve 1 si es positivo o -1 si es negativo. Podemos pensarlo como una función que tiene un vector aleatorio y el hiperplano normal a este vector (lo llamaremos hiperplano aleatorio) y dado un elemento devuelve 1 si el elemento y el vector aleatorio están del mismo lado del hiperplano aleatorio o -1 en el caso contrario.

> En vez de tomar vectores aleatorios basta con utilizar *scketches* (vectores aleatorios cuyas componentes son 1 o -1).

La función de minhash entonces va a tomar k funciones m diferentes y va a utilizar sus resultados como bits para el número a devolver. Esto significa que nuestra función M va a devolver valores entre 0 y 2^k.

> A mayores k mayores probabildades de falsos negativos ya que necesitamos que los puntos estén del mismo lado de los k hiperplanos usados para que el valor de minhash coincida.

La probabilidad de que los dos vectores, con distancia angular θ, estén en el mismo lado es de $1 - \frac{\theta}{180}$. Esto significa que nuestra familia de LSH queda definida como $H(d_1, d_2, 1 - \frac{d_1}{\pi}, 1 - \frac{d_2}{\pi})$ (recordemos que d_1 y d_2 son distancias angulares).

Distancia euclídea

Espacios euclídeos vs espacios no euclídeos

Los espacios euclídeos tienen múltiples dimensiones y un valor real ubica cada punto en cada una de ellas. Existe una noción de promedio entre los puntos que no se da en otros espacios. En los espacios no euclídeos las mediciones de distancia se basan en propiedades de los puntos pero no en su posición.

> La distancia angular y la de Jaccard son ejemplos de distancias en espacios no euclídeos.

Método de las proyecciones

La idea es generar hiperplanos aleatorios y dividirlos en buckets de ancho w. Luego proyectamos nuestro punto en el hiperplano y devolvemos el número de bucket en el que cae.
Uno de los problemas con este método es que nos pone una restricción en d_1 y d_2.
Entonces nuestra familia LSH queda definida como $H(w/2, 2w, p \geq 1/2, p \leq 1/3)$

> En la práctica es difícil encontrar un parámetro w bueno.

Information Retrieval

Introducción

Es la rama de la computación que se encarga de recuperar información (no estructurada) almacenada. En general lo que nos va a importar es como vamos a buscar un término específico en una lista de documentos.

Índices Invertidos

Es una estructura que contiene todos los términos y los documentos en los que aparece. Para poder buscar términos rápidamente vamos a utilizar búsqueda binaria. Es por esto que vamos a necesitar que los términos estén ordenados y que todos los índices tengan tamaño fijo. Un esquema para la estructura del índice es la siguiente:

Términos	Documentos
papa	1, 2, 4
papel	1, 4
paprika	2
pepa	3, 4

Vemos que los términos están ordenados pero no todos los índices tienen el mismo tamaño. El primero tiene un término de 4 letras con 3 documentos y el tercero tiene 7 letras con 1 documento. Es por eso que vamos a estar modificando esta estructura para que todos los indices tengan el mismo tamaño.

Almacenamiento del léxico

Léxico concatenado

La idea es tener un arreglo con todos los terminos concatenados. Siguiendo con el ejemplo anterior tendría el arreglo

```
papapapelpaprikapepa
|   |    |      |
0   4    9      16
```

Y en vez de guardar cada termino por separado guardamos las posiciones donde empieza la cadena en nuestro arreglo concatenado. Como son todos enteros van a tener todos el mismo tamaño. Nuestro léxico quedaría así:

Términos
0
4
9
16

Front Coding

La idea es la misma que con léxico concatenado pero aprovechando que cuando ordenamos los términos alfabéticamente, cada elemento tiene las primeras letras muy parecidas al elemento anterior. Entonces en vez de almacenar todas las letras por cada término solo almacenamos las letras diferentes al anterior. Además tenemos que guardar cuantas letras comparten. Nuestro nuevo léxico concatenado quedaría algo como:

```
papaelrikaepa
|   | |   |
0   4 6   10
```

Nuestro nuevo léxico tiene que guardar la posición y cuantos caracteres son iguales al anterior

Posicion	Caracteres iguales
0	0
4	3
6	3
10	1

El problema con esto es que para conseguir uno de los términos tengo que reconstruir todos los anteriores. Para solucionar esto vamos a hacer front coding parcial. Esto significa que cada

n términos vamos a poner toda la palabra en el arreglo concatenado y los caracteres iguales en cero. Si tomamos $n = 2$ nuestro nuevo léxico quedaría como:

```
papaelpaprikaepa
|   | |      |
0   4 6      13
```

Posicion	Caracteres iguales
0	0
4	3
6	0
13	1

Almacenamiento de punteros a documentos

Para almacenar los punteros a documentos vamos a usar la misma técnica que con el léxico. Si tenemos los siguientes documentos:

Documentos
1, 2, 4
1, 4
2
3, 4

Los tenemos que concatenar para conseguir una estructura de tamaño fijo

```
12414234
|  | ||
0  3 56
```

Y nuestra nueva estructura sería

Documentos
0
3
5
6

El único cambio que vamos a hacer es que en vez de guardar los punteros a documentos vamos a guardar las distancias entre punteros. Hacemos esto porque nuestra codificación va a favorecer números más chicos.

> La lista de documentos 1, 2, 4 quedaría como 1, 1, 2.