Búsqueda y Recuperación de Información en Textos

Víctor Mijangos de la Cruz

II. Métodos formales en RI

Búsqueda de información con expresiones regulares

Alfabetos y cadenas

Alfabeto

Un alfabeto es un conjunto $\Sigma = \{a_1, a_2, ..., a_n\}$, donde cada $a_i \in \Sigma, i = 1, ..., n$, es un elemento al que llamamos *símbolo*.

Cadena

Una cadena sobre un alfabeto es un conjunto ordenado de símbolos $a = (a_1, a_2, ..., a_n)$, con $a_i \in \Sigma$.
La longitud de la cadena $|a|$ es el número n de símbolos.

Σ^*

Sea Σ un alfabeto. Al conjunto de todas las cadenas sobre Σ se le denota como Σ^* (sigma estrella).

Operaciones sobre cadenas

Las cadenas pueden operarse de distinta forma, la operación más importante es la concatenación:

Concatenación

Sean $a = (a_1, ..., a_n)$ y $bn = (b_1, ..., b_n)$ dos cadenas sobre el alfabeto Σ, entonces la concatenación de a con b, denotada como $a \cdot b$, es la cadena $(a_1, ..., a_n, b_1, ...b_n)$.

Podemos definir un elemento nulo, que funciona como el 0 en el espacio de cadenas:

Elemento nulo

El elemento nulo ϵ cumple que, para toda cadena a, se tiene que:

$$\epsilon \cdot a = a \cdot \epsilon = a$$

Operador estrella

El operador estrella es un operador de suma importancia, pero para definirlo, requerimos definir el **operador exponente** sobre una cadena $a \in \Sigma^*$ de la siguiente forma:

- $a^0 = \epsilon$
- $a^n = a^{n-1} \cdot a$

Operador estrella

El **operador estrella** o estrella de Kleen es un operador sobre una cadena $a \in \Sigma^*$ que puede definirse de la siguiente forma:

$$a^* := \{a^i : i \geq 0\}$$

Un operador similar es el operador suma que se define como: $a^+ := \{a^i : i > 0\}$

Expresiones regulares

Las expresiones regulares se definen enumerativamente de la siguiente forma:

Expresión regular

Dado un alfabeto Σ y el conjunto de sus cadenas Σ^*, el conjunto de expresiones regulares E, se define de la siguiente manera:

1. $\emptyset$ es una expresión regular.
2. ϵ es una expresión regular.
3. $a \in \Sigma^*$ es una expresión regular.
4. Si $r_1 \in E$ y $r_2 \in E$ entonces $(r_1 \cdot r_2) \in E$
5. Si $r \in E$ entonces $(r)^* \in E$

Las expresiones regulares denotan **patrones textuales** y, por tanto, pueden fungir como queries de búsquedas.

Denotación de expresión regular

Ligado al concepto de expresión regular, está el concepto de denotación:

Denotación

Dada una expresión regular r su denotación, $[[r]]$, es el conjunto de cadenas definido de la siguiente forma:

1. $[[\emptyset]] = \emptyset$
2. $[[\epsilon]] = \{\epsilon\}$
3. $[[a]] = \{a\}$, con $a \in \Sigma^*$
4. Si $[[r_1]] = r_1$ y $[[r_2]] = r_2$ entonces $[[(r_1 \cdot r_2)]] = [[r_1]] \cdot [[r_2]]$
5. Si $[[r]] = r$ entonces $[[(r)^*]] = [[r]]^*$

La denotación es el conjunto de **resultados** concretos del patrón expresado en la regex.

Operadores sobre expresiones regulares

Las expresiones regulares permiten ciertos operadores que facilitan su expresión. Algunos de ellos son:

Operador OR

El operador booleano OR entre dos cadenas $a, b \in \Sigma^*$ se expresa como $a|b$ e implica a OR b (encuentra la cadena a o la cadena b).

La denotación de el operador OR puede expresarse como:

$$[[a|b]] = \{a\} \cup \{b\}$$

Este operador puede usarse con **agrupamiento**, el cual se denota con paréntesis. Dadas tres cadenas $a, b, c \in \Sigma^*$:

$$[[c(a|b)]] = \{ca\} \cup \{cb\}$$

Cuantificadores

Los símbolos cuantificadores más comúnes son ?, $+$ y $\{n\}$. Estos expresan la repetición de una símbolo:

RegEx	Explicación	Denotación
?	0 ó 1 ocurrencia de símbolo	$[[s?]] = \{s^i : i = 0, 1\}$
$+$	1 ó más ocurrencias de símbolo	$[[s^+]] = \{s^i : i > 0\}$
$\{n\}$	Símbolo a la n	$[[s\{n\}]] = \{s^n\}$

El operador estrella también es parte de los cuantificadores.

Se puede aplicar los cuantificadores a **agrupaciones** de símbolos utilizando los paréntesis:

$$[[(abc)^+]] = \{(abc)^i : i > 0\}$$

Otras expresiones regulares

Algunos símbolos utilizados en las expresiones regulares son:

RegEx	Explicación	Denotación
.	Cualquier caracter	$[[.]] = \Sigma \subset \Sigma^+$
$\widehat{}$	El comienzon de una cadena	$[[\widehat{}s]] = \{s : \nexists a \text{ tal que } a \cdot s\}$
\$	El final de una cadena	$[[s\$]] = \{s : \nexists a \text{ tal que } s \cdot a\}$
$\backslash d$	Un número entre 0 y 9	$[[\backslash d]] = \{0, 1, ..., 9\}$

Por ejemplo:

$$[[\widehat{}.+]] = \{w \in \Sigma^* : |w| > 0, w \text{ inicio de cadena}\}$$

Expresiones regulares y lenguajes regulare

Los **lenguajes regulares** son lenguajes que se pueden describir por medio de expresiones regulares. Formalmente se pueden definir como:

Lenguaje regular

Un lenguaje es regular si existe una expresión regular r tal que $L = [[r]]$.

Un lenguaje regular es, pues, la **denotación de una expresión regular**. Por ejemplo:

$$L = [[. + / \ @. + / \ .com]] = \{\text{correos electrónicos}\}$$

El símbolo / es un símbolo de **escape**.

Propiedades de lenguajes regulares

Los lenguajes regulares tiene la propiedad de ser cerrados bajo ciertas operaciones (Hopcroft, 2001):

Proposición

Sean L_1 y L_2 dos lenguajes regulares, entonces L_1 y L_2 son cerrados bajo la operación de concatenación y estrella de Kleene.

Si $L_1 = [[gat(a|o)]]$ y $L_2 = [[(\epsilon|e)s]]$, entonces la concatenación de los lenguajes es un lenguaje regular:

$$L_1 \cdot L_2 = [[gat(a|o)(\epsilon|e)s]]$$

Procesadores de regex

Las expresiones regulares se expresan con **operadores simbólicos** que deben ser procesados por una máquina. Para esto se usan los procesadores de regex.

Procesador de regex

Un procesador de regex traduce los operadores simbólicos de las expresiones regulares en representaciones que una máquina pueda utilizar para realizar una búsqueda del patrón dentro de una cadena.

Un posible procesador de regex se basa en el **algoritmo de construcción de Thompson**, que requiere de autómatas finitos.

Autómata finito

El **autómata** es una forma de representar a las expresiones regulares; éste se define como:

Autómata de estados finitos

Un autómata de estados finitos es una 5-tupla $A = (Q, \Sigma, q_0, F, \delta)$, tal que $Q = \{q_0, q_1, ..., q_T\}$ es un conjunto de estados; Σ es el alfabeto de entrada; $q_0 \in Q$ es el estado inicial; $F \subseteq Q$ es el conjunto de estados finales; y $\delta : Q \times \Sigma \rightarrow Q$ es la función de transición.

Ejemplo: autómata

- Alfabeto: $\Sigma = \{c, a, t, r\}$
- Conjunto de estados: $Q = \{q_0, q_1, q_2, q_3\}$
- Estado inicial: $q_0 \in Q$
- Estados finales: $F = \{q_3\}$
- Función de transición:

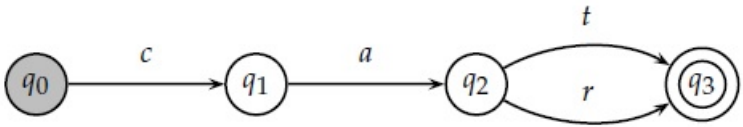
$$\delta(q_0, c) = q_1$$
$$\delta(q_1, a) = q_2$$
$$\delta(q_2, t) = q_3$$
$$\delta(q_2, r) = q_3$$

Ejemplo: autómata

Otra forma de expresar la función de transición es por una matriz:

	c	a	t	r
0	1	0	0	0
1	0	2	0	0
2	0	0	3	3
3:	0	0	0	0

Gráficamente, el autómata es:



Aplicación del autómata finito a cadenas

Un autómata rechaza o acepta una cadena a partir de recorrer los símbolos (caracteres) que lo componen. Una función que acepte o rechace cadenas regresará un valor booleano: Verdadero (True) si acepta la cadena y Falso si no la acepta.

Algorithm Aceptación de cadenas por autómata finito

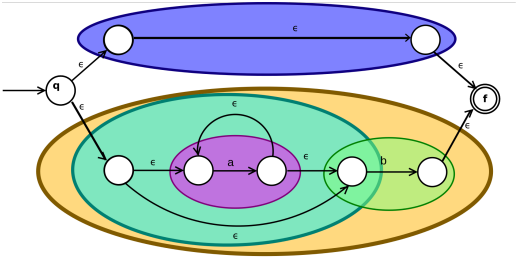
```
1: procedure ACCEPTS-STRING(A, string)
2:     q ← q_0
3:     for c in string do
4:         q ← δ(q, c)
5:     end for
6:     return q ∈ F
7: end procedure
```

Algoritmo de construcción de Thompson

Con la ayuda de los autómatas finitos, podemos usar las expresiones regulares para hacer búsqueda de patrones dentro de cadenas.

Algoritmo de construcción de Thompson

El algoritmo de construcción de Thompson es un algoritmo que compila una expresión regular y la convierte en un autómata (no-determinístico).



Algoritmo de construcción de Thompson

Algorithm Algoritmo de Construcción de Thompson

1: **procedure** THOMPSON-CONSTRUCTION(regex, Σ)
2: **if** regex $= \epsilon$ **then**
3: $\delta \leftarrow^{+} (q, \epsilon, q_f \in F)$
4: **return** $N(\epsilon) = \{\Sigma, Q, q, F, \delta\}$
5: **end if**
6: **if** regex $= a \in \Sigma$ **then**
7: $\delta \leftarrow^{+} (q, a, q_f \in F)$
8: **return** $N(a) = \{\Sigma, Q, q, F, \delta\}$
9: **end if**
10: **if** regex $= a|b,\ a, b \in \Sigma^*$ **then**
11: $\delta \leftarrow^{+} (q, \epsilon, q')$ **and** $\delta \leftarrow^{+} (q, \epsilon, q'')$ **and** $\delta \leftarrow^{+} (q', N(a), q_t)$ **and** $\delta \leftarrow^{+} (q'', N(b), q'_t)$
12: $\delta \leftarrow^{+} (q_t, \epsilon, q_f \in F)$ **and** $\delta \leftarrow^{+} (q'_t, \epsilon, q_f \in F)$
13: **return** $N(a|b) = \{\Sigma, Q, q, F, \delta\}$
14: **end if**
15: **end procedure**

Algoritmo de construcción de Thompson

Algorithm Algoritmo de Construcción de Thompson

1: **procedure** THOMPSON-CONSTRUCTION(regex,A)
2: **if** regex $= a \cdot b,\ a, b \in \Sigma^*$ **then**
3: $\delta \leftarrow^+ (q, N(a), q')$
4: $\delta \leftarrow^+ (q', N(b), q_f \in F)$
5: **return** $N(a \cdot b) = \{\Sigma, Q, q, F, \delta\}$
6: **end if**
7: **if** regex $= a^*$ **then**
8: $\delta \leftarrow^+ (q, \epsilon, q_f \in F)$ **and** $\delta \leftarrow^+ (q, \epsilon, q')$
9: $\delta \leftarrow^+ (q', N(a), q'')$ **and** $\delta \leftarrow^+ (q'', \epsilon, q')$ **and** $\delta \leftarrow^+ (q'', \epsilon, q_f \in F)$
10: **return** $N(a^*) = \{\Sigma, Q, q, F, \delta\}$
11: **end if**
12: **end procedure**

Procesador de regex

Por ejemplo, si tomamos la siguiente expresión regular

$$. + @. + / \ .com$$

Obtenemos el siguiente conjunto de reglas:

$$\begin{array}{c} \delta(q_0, \epsilon, q_1) \ ; \ \delta(q_4, \epsilon, q_5) \\ \delta(q_1, s \in \Sigma, q_2) \ ; \ \delta(q_5, s \in \Sigma, q_6) \\ \delta(q_2, \epsilon, q_1) \ ; \ \delta(q_6, \epsilon, q_5) \\ \delta(q_2, \epsilon, q_3) \ ; \ \delta(q_5, \epsilon, q_7) \\ \delta(q_3, @, q_4) \ ; \ \delta(q_7, .com, q_8) \end{array}$$

Nuestro autómata está conformado por los símbolos del alfabeto, los estados $Q = \{q_i : i = 0, ..., 8\}$, $F = \{q_8\}$, q_0 como estado inicial, y las reglas que hemos definido.

Stemming y lematización

Patrones morfológicos y stemming

Cuando consideramos una lista como:

- niña
- niño
- niñas
- niñitos

Notamos que las palabras expresan un mismo significado básico o **significado léxico**. En este caso, existe un **patrón común** en estas palabras, que podemos llamar **stem**.

Afijos y stem

En una cadena podemos definiri ciertos elementos según su posición.

Afijos y stem

Un afijo es una subcadena $x \in \Sigma^*$ de una cadena mayor w. Si $w = x \cdot b$ se llama **prefijo**. Si $w = b \cdot x$ se llama **sufijo**.
El elemento b tal que $x \cdot b \cdot y = w$ (x prefijo e y sufijo) se conoce como **stem** o raíz.

En las palabras de un lenguaje natural nos interesan aquellos patrones que son comunes a muchas palabras.

Prefijo	Stem	Sufijo
in	toca	ble
in	tacha	ble
	ama	ble

Stemming

Stemming

El proceso de **stemming** o truncamiento consiste en reducir un token léxico a una supuesta base por medio de truncar los afijos correspondientes.

Esto es:

$$w \mapsto b \in M$$

Donde M es el conjunto de bases de una lengua ($w = x \cdot b \cdot y$).

x e y son afijos y muchas veces se conocen como **contexto**.

El algoritmo más común para realizar stemming se conoce como **Algoritmo de Porter**.

Algoritmo de Porter

El **Algoritmo de Porter** es un algoritmo de stemming basado eliminación de **sufijos**.
Tiene varias fases; la primera de ellas es buscar **regiones** que se aproximen a las sílabas en una lengua:

- R1: después de la primera consonante que sigue a una vocal, o la región nula al final de la palabra si no existe dicha vocal.
- R2: después de la primera consonante que sigue a una vocal dentro de R1, o la región nula al final de la palabra si no existe dicha vocal.
- RV: si la segunda letra es una consonante, después de la siguiente vocal subsecuente. Si las dos primeras letras son vocales, después de la consonante subsiguiente. En otro caso, después de la tercera letra.

Zonas de Porter

Considérese la palabra **trabajar**. Las regiones que podemos localizar son las siguientes:

	t	r	a	b	a	j	a	r
R1				b	a	j	a	r
R2						j	a	r
RV				b	a	j	a	r

En el caso de **aúreo**:

	á	u	r	e	o
R1			r	e	o
R2					
RV			r	e	o

Algoritmo de Porter

Algorithm Regla para algoritmo de Porter

```
1: procedure PRONOUNS(w)
2:     pronoun ← [me, se, sela, selo, selas, selos, la, le, lo, las, les, los, nos]
3:     suffix ← [iéndo, ándo, ér, ár, ír, iendo, ando, er, ar, ir, uyendo]
4:     if pronoun · suffix in RV(w) then
5:         delete pronoun
6:     end if
7: end procedure
```

Algoritmo de Porter

```
1: procedure SUFFIX-REMOVAL(w)
2:     suffix ← [anza, anzas, ico, ica, icos, icas, ismo, ismos, able, ables, ible, ibles, ista, istas, oso,...
3:                                                  ..osa, osos, osas, amiento, amientos, imiento, imientos]
4:     if suffix in R2(w) then
5:         delete suffix
6:     end if
7:     suffix ← [adora, ador, ación, adoras, adores, aciones, ante, antes, ancia, ancias]
8:     if ic · suffix in R2(w) then
9:         delete suffix
10:    end if
11:    suffix ← [iva, ivo, ivas, ivos]
12:    if at · suffix in R2(w) then
13:        delete suffix
14:    end if
15: end procedure
```

Algoritmo de Porter

```
1: procedure SUFFIX-REMOVAL-CONTEXT(w)
2:     suffix ← [idad, idades]
3:     prec ← [abil, ic, ib]
4:     if prec · suffix in R2(w) then
5:         DELETE suffix
6:     end if
7:     prec ← [iv, os, ic, ad]
8:     if prec · amente in R2(w) then
9:         DELETE amente
10:    end if
11:    prec ← [ante, able, ible]
12:    if prec · mente in R2(w) then
13:        DELETE mente
14:    end if
15: end procedure
```

Algoritmo de Porter

```
 1: procedure REPLACEMENT(w, suffix, replace)
 2:     if suffix in R2(w) then
 3:         replace sufix
 4:     end if
 5: end procedure
 6: procedure SUFFIX-REPLACEMENT(w)
 7:     REPLACEMENT(w, [logía, logías], log)
 8:     REPLACEMENT(w, [ución, uciones], u)
 9:     REPLACEMENT(w, [encia, encias], ente)
10:     REPLACEMENT(w, [ución, uciones], u)
11: end procedure
```

Algoritmo de Porter

```
1: procedure VERB-SUFFIXES-A(w)
2:     suffix ← [ya, ye, yan, yen, yeron, yendo, yo, yó, yas, yes, yais, yamos]
3:     if u · suffix in RV(w) then
4:         delete suffix
5:     end if
6: end procedure
```

Algoritmo de Porter

1: **procedure** VERB-SUFFIXES-B(w)
2: suffix ← [arían, arías, arán, arás, aríais, aría, aréis, aríamos, aremos, ará, aré, erían, erías, erán, erás, eríais, ería, eréis, eríamos, eremos, erá, eré, irían, irías, irán, irás, iríais, iría, iréis, iríamos, iremos, irá, iré, aba, ada, ida, ía, ara, iera, ad, ed, id, ase, iese, aste, iste, an, aban, ían, aran, ieran, asen, iesen, aron, ieron, ado, ido, ando, iendo, ió, ar, er, ir, as, abas, adas, idas, ías, aras, ieras, ases, ieses, ís, áis, abais, íais, arais, ierais, aseis, ieseis, asteis, isteis, ados, idos, amos, ábamos, íamos, imos, áramos, iéramos, iésemos, ásemos, en, es, éis, emos]
3: **if** *suffix* **in** RV(w) **then**
4: **delete** suffix
5: **end if**
6: suffix ← [en, es, éis, emos]
7: **if** *gu* · *suffix* **in** RV(w) **then**
8: **delete** u · suffix
9: **end if**
10: **end procedure**

Algoritmo de Porter

```
1: procedure RESIDUAL-SUFFIXES(w)
2:     suffix ← [os, a, o, á, í, ó]
3:     if suffix in RV(w) then
4:         delete suffix
5:     end if
6:     suffix ← [e, é]
7:     if gu · suffix in RV(w) then
8:         delete u · suffix
9:     end if
10: end procedure
```

Algoritmo de Porter

Algorithm Algoritmo de Porter

```
1: procedure STEMMING(w)
2:     w ← PRONOUNS(w)
3:     w ← SUFFIX-REMOVAL(w) and SUFFIX-REMOVAL(w) and SUFFIX-REPLACEMENT(w)
4:     if no se qutaron sufijos en el paso anterior then
5:         w ← VERB-SUFFIXES-A(w)
6:     end if
7:     if no se qutaron sufijos en el paso anterior then
8:         w ← VERB-SUFFIXES-B(w)
9:     end if
10:    w ← Residual-sufficex(w)
11:    return w
12: end procedure
```

Ultra-stemming

El algoritmo de Porter suele ser el estándar de *stemming*; sin embargo, como se ve su implementación es compleja sobre todo cuando no se conoce del todo la lengua con la que se trabaja.

Una propuesta más simple, de menor complejidad y más rápida, aunque menos efectiva, es el ultra-stemming (Torres-Moreno, 2012):

Ultra-stemming

Éste es un algoritmo de fuerza bruta que se basa en realizar un corte fijo a las palabras. Dada una cadena $w \in \Sigma^*$ su stem está definido como:

$$stem = w[:n]$$

para algún $n \in \mathbb{N}$.

Su base radica en la intuición lingüística de que existe un tamaño mínimo de palabra. Generalmente, se asume que $4 \leq n \leq 6$.

Lematización

Lematización

El proceso de lematización consiste en llevar un token a su lema o forma de diccionarios; es decir, en mapear:

$$w \mapsto \text{LEMA}(w)$$

Para hacer esto, se utiliza un diccionario que relaciona los tipos con los lemas. Es común que se haga un proceso de stemming antes para reducir el número de entradas del diccionario.

Token	stem	lema
niñas	niñ	niño
niñito	niñ	niño
podremos	pod	poder
puedo	pued	poder

Búsqueda booleana

Búsqueda indexada

Una alternativa a la búsqueda lineal (escanear el documento hasta encontrar los términos) consiste en indexar los documentos de antemano.

Matriz de incidencia

Una matriz de incidencia término-documento indica qué documentos contienen un término dado. Sean $d_j = \{w_1, ..., w_{|d_i|}\}$, $j = 1, 2, ..., N$, una colección de documentos, la matriz de incidencia se define como:

$$M = (m_{i,j}) = \begin{cases} 1 & \text{si } w_i \in d_j \\ 0 & \text{otros casos} \end{cases}$$

Los términos funcionan aquí como las unidades de indexación.
Se tienen representaciones para las palabras (renglones) y los documentos (columnas).

Matriz de incidencia

Supóngase que se tienen los siguientes documentos:

- $d_1 = \{\text{gato, perro, aves}\}$
- $d_2 = \{\text{perro, can}\}$
- $d_3 = \{\text{auto, llantas, gato}\}$

La matriz de incidencia se conformaría como:

	d_1	d_2	d_3
gato	1	0	1
perro	1	1	0
aves	1	0	0
can	0	1	0
auto	0	0	1
llantas	0	0	1

Matriz de incidencia

Construir la matriz de incidencia implica comparar los términos con los documentos, por lo que su complejidad temporal y espacial es de $O(n \cdot N)$.

Algorithm Construcción de matriz de incidencia

```
1: procedure BUILD-INCIDENCE-MATRIX(collection)
2:     M ∈ ℝ^{n×N}, n número de términos, N número de documentos.
3:     for term_i in collection do
4:         for document_j in collection do
5:             if term_i in document_j then M_{i,j} = 1
6:             end if
7:         end for
8:     end for
9:     return M
10: end procedure
```

Representaciones de documentos y palabras

Las **columnas** de la matriz de incidencia pueden representar los **documentos**.
Por ejemplo, para el documento d_2 tenemos que:

$$\rho(d_2)^T = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Por su parte, los **renglones** pueden representar a los **términos**. Así, el término "gato" tiene la representación:

$$\rho(gato)^T = \begin{pmatrix} 1 & 0 & 1 \end{pmatrix}$$

Recuperar las **posiciones de los 1** en esta representación nos indica cuáles documentos **contienen al término**.

Modelo de búsqueda booleana

Modelo de búsqueda booleana

El modelo de búsqueda booleana es un modelo de recuperación de información en el que la query es una expresión booleana; es decir, en que se usan operadores AND, OR y NOT.

Los operadores booleanos trabajan con valores binarios, los cuales pueden traducirse a bits: $True \mapsto 1$, $False \mapsto 0$.

El operador **not** es un operador unitario que opera con un bit de la siguiente forma:

$$\text{NOT}(x) = \begin{cases} True & \text{si } x = False \\ False & \text{si } x = True \end{cases}$$

Operaciones booleanas

Las operaciones de **and** y OR son operaciones binarias que toman con dos bits y devuelven uno. Estas operan de la siguiente forma:

bit 1	bit 2	AND	OR
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

bit 1	bit 2	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Por ejemplo, se puede expresar la operación $\text{AND}(\textit{True}, \textit{False}) = \textit{Flase}$ como:

$$1 \text{ AND } 0 = 0$$

Operaciones entre términos

Podemos pensar que cada término está codificado con **bits**. Es decir que $c(\textit{gato}) = 101$. Aplicar la operación NOT implica aplicarla a cada uno de los bits:

$$\text{NOT } \textit{gato} := \text{NOT}(101) = 010$$

De igual forma aplicar las operaciones de AND, OR en dos cadenas implica aplicarlas a cada bit de las dos cadenas. Asúmase $c(\textit{perro}) = 110$:

$$\textit{perro AND gato} = \text{AND}(101, 110) = 100$$

Las **posiciones de los 1** en el resultado indican los documentos en que están presente los términos: 'NOT gato' está contenido en el documento 2, y 'perro AND gato' en el documento 1.

Operaciones entre términos

El modelo de búsqueda booleana permite hacer operaciones más complejas que dependan de los operadores booleanos básicos.

- Concatenación de operaciones, como: term_1 AND term_2 AND term_3 AND term_4.
- Combinación de operaciones: term_1 OR (term_2 AND NOT term_3).
- Operaciones complejas como OR exclusivo:

$$\text{XOR}(t_1, t_2) = \text{AND}\Big(\text{NOT}\big(\text{AND}(t_1, t_2)\big), \text{OR}(t_1, t_2)\Big)$$

Dificultades de la matriz de incidencia

Las búsquedas booleanas son útiles para realizar recuperación de información de manera rápida. Pero en **colecciones muy grandes** la matriz de incidencia puede ser difícil de manejar.

La matriz de incidencia es una matriz **dispersa** (*sparse*); es decir contiene una gran cantidad de 0s que no aportan información a la búsqueda.

Las matrices dispersas son problemáticas puesto que requieren mayor almacenamiento que la información que guardan efectivamente.

Se propone una alternativa a la matriz de incidencia que es el **índice invertido**.

Índice invertido

Índice invertido

Un índice invertido para el modelo de búsqueda booleana consiste en dos elementos:

- Diccionario de términos: contiene los términos que se encuentran en una colección.
- Lista de *postings*: A cada término se le asocia los índices de los documentos que lo contienen.

El diccionario obtenido asociará los documentos a cada término:

$$w \rightarrow \{j : w \in d_j\}$$

j funciona como un identificador de documento. Esto **elimina los 0s**, pues ahora sólo se cuenta con los índices de los documentos.

Índice invertido

Por ejemplo, considérese la siguiente colección de documentos:

- $d_1 = \{\text{gato, perro, aves}\}$
- $d_2 = \{\text{perro, can}\}$
- $d_3 = \{\text{auto, llantas, gato}\}$

El diccionario que conformariamos sería:

Términos	**Postings**
gato	1, 3
perro	1, 2
aves	1
can	2
auto	3
llantas	3

Construcción de índice invertido

La construcción de un índice invertido conlleva los siguientes pasos:

1. Recolectar los documentos que conformarán la colección/corpus.
2. Tokenizar los documentos para convertirlos en conjuntos de términos: $d = \{w_1, ..., w_n\}$.
3. Preprocesamiento lingüístico, como stemming o lematización.
4. Indexar los documentos, creando un índice invertido.

Nos enfocaremos en el último punto que tiene que ver con el índice invertido.

Construcción de índice invertido

Para poder crear adecuadamente el índice invertido, se debe asociar un identificador a cada documento.

docID

Un identificador de documento, que denotamos **docID**, asigna a cada documento un único número en una serie. Es decir:

$$d \mapsto i \in \mathbb{N}$$

Por cada término en un documento se asocia el término al identificador:

$$(term, docID)$$

Construcción de índice invertido

Seguimos, entonces, los siguientes pasos para crear las lista de postings:

1. Los pares $(term, docID)$, que corresponden a un documento, se **ordenan** alfabéticamente.
2. Los términos de un mismo documento se llevan hacia un sólo **tipo**.
3. Los términos correspondientes a un mismo **tipo en documentos diferentes se asocian**, indicando los docIDs, **lista de postings**, y asociando a cada término el número de documentos que lo contienen, **frecuencia de término**.

El resultado será que a cada término se le asociará la**frecuencia** y la **lista de postings**:

$$term \rightarrow frequency(term), \{docID(d) : term \in d\}$$

Construcción de índice invertido

Lista inicial al revisar los documentos:

término	docID
gato	1
perro	1
aves	1
perro	2
can	2
gato	3
auto	3
llantas	3

Lista al ordenar los términos por orden alfabético:

término	docID
auto	3
aves	1
can	2
gato	1
gato	3
llantas	3
perro	1
perro	2

Índice invertido de términos, con frecuencia y postings:

término	Frecuencia	Postings
auto	1	[3]
aves	1	[1]
can	1	[2]
gato	2	[1, 2]
llantas	1	[3]
perro	2	[1, 2]

Recuperación de información con índices invertidos

Los índices invertidos permiten una fácil recuperación de información a través de los docIDs.

Algorithm Búsqueda simple con índice invertido

```
1: procedure RETRIEVE-QUERY(index, query)
2:     term ← index.DICTIONARY(query)
3:     postings ← term.POSTINGLIST
4:     for docID in postings do
5:         document ← docID.DOCUMENT
6:         yield document
7:     end for
8: end procedure
```

Operadores booleanos en índices invertidos

Si bien la recuperación de un sólo término es sencilla, para realizar **operaciones booleanas**, debemos ver estos operadores como **operaciones entre conjuntos**:

- Operador NOT: Funciona como el complemento del conjunto (de postings):

$$\text{NOT}(term) = documents \setminus postings(term)$$

- Operador OR: Funciona como unión de dos conjuntos (de postings):

$$\text{OR}(term_1, ter_2) = postings(term_1) \cup postings(term_2)$$

- Operador AND: Funciona como intersección de dos conjuntos (de postings):

$$\text{AND}(term_1, ter_2) = postings(term_1) \cap postings(term_2)$$

Operadores booleanos en índices invertidos

El operador OR que denota **unión** de lsitas de posting simplemente agrega los elementos de la primera lista en la segunda lista (o viceversa).

El operador NOT representa el **complemento** con respecto al conjunto de todos los documentos, su algoritmo es simple (complejidad $O(n)$).

Algorithm Algoritmo para operador NOT

```
1: procedure NOT(postingA, collection)
2:     result ← []
3:     for docID in collection.DOCUMENTS do
4:         if docID not in postingA then
5:             result ←+ docID
6:         end if
7:     end for
8:     return result
9: end procedure
```

Intersección para operador AND

```
1: procedure INTERSECT(postingA, postingB)
2:     result ← []
3:     for docID in postingA do
4:         while postingA ≠ ∅ and postingB ≠ ∅ do
5:             i, j ← postingA.FIRSTPOSITION, postingB.FIRSTPOSITION
6:             if postingA.docID[i] = postingA.docID[j] then
7:                 result ←+ docID[i]
8:                 i, j ← next(i), next(j)
9:             end if
10:            elif postingA.docID[i] < postingA.docID[j] do i ← next(i)
11:            else j ← next(j)
12:        end while
13:    end for
14:    return result
15: end procedure
```

Optimización de consulta

El algoritmo de intersección tiene complejidad lineal sobre el total de postings ($O(n+m)$, con n y m postings en cada lista, respectivamente). Esto se puede complicar cuando se **aplican** varias operaciones AND. Podemos **optimizar** la búsqueda.

Optimización de consulta

La optimización de consulta consiste en definir la organización del trabajo para contestar una query, de tal forma que se reduzca la cantidad de trabajo que se realiza en su búsqueda.

Para búsqueda de varios términos t_1 AND t_2 AND ... AND t_k podemos intersectar **primero** las listas de posting **más pequeñas**. Haciendo que las listas no crezcan, tal que n y m sean lo más pequeñas posibles.

Operador AND sobre varios términos

Algorithm Consulta optimizada

```
1: procedure AND(t_1, t_2, ..., t_k)
2:     terms ← SORT-BY-FREQUENCY(t_1, t_2, ..., t_k)
3:     result ← terms[0].POSTING
4:     terms ← terms\result
5:     while terms ≠ ∅ and result ≠ ∅ do
6:         result ← INTERSECT(result, terms[0].POSTING)
7:         terms ← terms\result
8:     end while
9:     return result
10: end procedure
```

Modelo de recuperación booleana: conclusiones

La búsqueda de información con un modelo booleano puede aplicarse a queries planteadas en un lenguaje **muy cercano al natural** (operadores aparecen como palabras de la lengua).

Las consultas booleanas son **precisas**: los documentos coinciden con la búsqueda o no lo hacen.

Algunas **desventajas** de los operadores booleanos son:

- AND produce alta precisión, pero baja exhaustividad.
- OR produce alta exhaustividad, pero baja precisión.

Similitud y proximidad entre cadenas

Extensión de búsquedas booleanas

Una forma de extender los modelos de búsqueda booleana es agregar al modelo **operadores extra** que aporten herramientas de búsqueda. Algunos ejemplos son **operadores de proximidad**:

- Buscar coincidencias dentro de las misma oración (/s denota t_1 y t_2 aparecen en la misma oración).
- Buscar coincidencias en el mismo párrafo (/p).
- Buscar coincidencias que estén a una distancia máxima de $k \in \mathbb{N}$ palabras (/k, t_1 y t_2 no aparecen a más de k palabras de distancia).
- Términos que inicien o terminen con una subcadena (por ejemplo, 'amanec!' denota términos que empiezan con 'amanec').

Términos similares

Cuando se realiza una búsqueda pueden pasar casos que no permitan recuperar un término adecuadamente:

- El término no está en el índice (no fue observado) en la forma en que se busca (términos conjugados, en plural, etc.).
- Hay variación en la escritura del término, lo que no permite que las consultas coincidan.
- Se quiere recuperar términos similares.

Podemos definir un término de **similitud entre cadenas**, que determine qué tanto se parecen dos o más términos.

Espacio de cadenas

En las cadenas puede definirse una estructura algebraica que permite aplicar métodos que nos permitirán determinar similitud entre estas:

Estructura de las cadenas

Dado un alfabeto Σ y la operación de concatenación $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$, se cumple que:

1. Para todo $a, b, c \in \Sigma^*$, se cumple $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
2. Existe $\epsilon \in \Sigma^*$, tal que para toda $a \in \Sigma^*$ se cumple $\epsilon \cdot a = a \cdot \epsilon = a$.

$(\Sigma, \cdot, \epsilon)$ es un monoide.

Espacios métricos

Cuando hablamos de un espacio, podemos proponer propiedades que lo caracterizan y nos permiten operar con ellos. Una de ellas es la propiedad de ser métrico. Para esto, debemos definir el concepto de métrica.

Métrica

Dado un conjunto X, una métrica sobre X es una función $d : X \times X \to X$ que cumple:

1. Para todos $x, y \in X$, $d(x, y) \geq 0$.
2. Para todo $x, y \in X$, $d(x, y) = d(y, x)$ (simetría).
3. Para todo $x, y, z \in X$ se cumple que $d(x, y) \leq d(x, z) + d(z, y)$ (desigualdad triangular).

Espacio métrico

Un espacio métrico es una tupla (X, d), donde X es un conjunto y d una métrica sobre X.

Métrica de cadenas

Nuestro objetivo es definir una métrica sobre el espacio de cadenas de tal forma que podamos determinar la cercanía de los términos.

Métrica de cadenas

Una métrica de cadenas es una métrica que determina la distancia entre dos cadenas. Esta distancia aproxima la coincidencia entre las dos cadenas.

Una métrica de cadenas puede verse como una medida de **similitud** inversa; es decir, dos cadenas se parecen más entre sí en tanto su métrica sea más cercana a 0.

Métrica de coincidencia

Dado un alfabeto Σ, una métrica sobre las cadenas Σ^* puede definirse como:

$$d(t_1, t_2) = \begin{cases} 0 & \text{si } t_1 = t_2 \\ 1 & \text{en otro caso} \end{cases}$$

Esta métrica determina una **coincidencia** entre una cadena y otra.

En un modelo de búsqueda, la utilización de esta métrica equivale a encontrar las coincidencias de un término con respecto a otro. Un término sólo es similar a sí mismo

Podemos extender la similitud entre cadenas a elementos que no sean coincidentes.

Ediciones

Para definir una métrica con la que podamos determinar una similitud de manera más aporpiada, introduciremos el concepto de edición.

Edición

Una edición puede entenderse como un cambio que se da entre las dos cadenas. En general, se consideran tres tipos de ediciones:

- Inserción: En una cadena $a = c_1...c_i...c_T$ (de longitud T) se inserta el símbolo x en la posición $i+1$; se obtiene la cadena $b = c_1...c_i x...c_T$ (de longitud $T+1$).
- Eliminación: Dada la cadena $a = c_1...c_i...c_T$ (de longitud T) se elimina el símbolo c_i obteniendo la cadena $b = c_1...c_{i-1}c_{i+1}...c_T$ (de longitud $T-1$).
- Sustitución: En la cadena $a = c_1...c_i...c_T$ (de longitud T) se sustituye el símbolo c_i por x obteniendo la cadena $b = c_1...c_{i-1}xc_{i+1}...c_T$ (de longitud T).

Transformación de cadenas

Proposición

Dado un alfabeto Σ, toda cadena de este alfabeto $a \in \Sigma^*$ puede transformarse en otra cadena $b \in \Sigma^*$ a partir de aplicar ediciones a la primera cadena hasta obtener la segunda.

Por ejemplo, las cadenas "perro" y "gato" pueden transformarse una en otra:

- Elimina *r* en la posición 3 (o 2) en la cadena *perro*.
- Sustituye *p* por *g* en la cadena *pero*.
- Sustituye *e* por *a* y *r* por *t*.

Estas ediciones no son únicas, pues se puede transformar la primera cadena en la segunda a partir de aplicar diferentes ediciones.

Distancia de Levenshtein

La distancia de Levenshtein es una métrica entre cadenas definida por Vladimir Levenshtein en 1965.

Distancia de Levenshtein

La distancia de Levenshtein es una función $lev : \Sigma^* \times \Sigma^* \to \mathbb{R}^+$ definida como:

$$lev(a, b) = \min\{|edits| : edits(a) = b\}$$

tal que *edits* es el conjunto de ediciones para pasar de la cadena a a la cadena b.

La distancia de Levenshtein **es una métrica** sobre el espacio Σ^*. Por tanto, (Σ^*, lev) es un espacio métrico.

Algoritmo de Wagner-Fischer

Computar la distancia de Levenshtein con fuerza bruta resulta muy costoso pues tenemos que determinar todas las posibles ediciones que puedan llevar de una cadena a otra.

El **algoritmo de Wagner-Fischer** es un algoritmo dinámico que reduce el cálculo para la distancia de levenshtein.

Se busca crear una matriz donde los renglones son los símbolos en la cadena de mayor longitud y las columnas los símbolos de la cadena de menor longitud.

Algoritmo de Wagner-Fischer

Dadas dos cadenas $a = a_1, ..., a_{T_1}$ y $b = b_1, ..., b_{T_2}$ con $T_1 \geq T_2$:

- Se crea tabla M de tamaño $(T_1 + 1) \times (T_2 + 1)$, sus entradas $M[i, j]$ inician vacías.
- Se llena la columna $M[i + 1, 1] = i$ para $i = 0, ..., T_1$; y el renglón $M[1, j + 1] = j$ para $j = 0, ..., T_2$. Estos valores representan las posiciones de los símbolos en a y b.
- En cada posición i de a y j de b, se toman los valores de ediciones observando las entradas alrededor de la posición actual:
  - Inserción: $M[i - 1, j] + 1$.
  - Eliminación: $M[i, j - 1] + 1$.
  - Substitución: $M[i - 1, j - 1] + \delta_{c_i \neq c_j}$.
  - Se toma la edición más pequeña:
    $M[i, j] = \min(M[i - 1, j] + 1, M[i, j - 1] + 1, M[i - 1, j - 1] + \delta_{c_i \neq c_j})$ para $i, j \geq 1$

Algoritmo de Wagner-Fischer

Cada entrada del la matriz M se llena con el valor de la edición más pequeña; esto es (para $i, j \geq 1$):

$$M[i,j] = \min(M[i-1,j]+1, M[i,j-1]+1, M[i-1,j-1]+\delta_{c_i \neq c_j})$$

		b_1	b_2	$\cdots$	b_{T_2}
	0	1	2	$\cdots$	T_2
a_1	1	$\min(1,1)$	$\min(1,2)$	$\cdots$	$\min(1, T_2)$
a_2	2	$\min(2,1)$	$\min(2,2)$	$\cdots$	$\min(2, T_2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
a_{T_1}	T_1	$\min(T_1,1)$	$\min(T_1,2)$	$\cdots$	$\min(T_1, T_2)$

La distancia de Levenshtein será:

$$lev(a,b) = \min(T_1, T_2) = \min(M[T_1-1, T_2]+1, M[T_1, T_2-1]+1, M[T_1-1, T_2-1]+\delta_{c_{T_1} \neq c_{T_2}})$$

Algoritmo de Wagner-Fischer

Algorithm Algoritmo de Wagner-Fischer

```
1: procedure LEVENSHTEIN(a,b)
2:     if |a| < |b| then
3:         return LEVENSHTEIN(b,a)
4:     end if
5:     M[0,i] ← [0,1,2,..., |a|], M[j,0] ← [0,1,2,..., |b|]
6:     for i from 1 to |a| do
7:         for j from 1 to |b| do
8:             M[i, j] ← min{M[i − 1, j] + 1, M[i, j − 1] + 1, M[i − 1, j − 1] + δ_{c_{T_1} ≠ c_{T_2}}}
9:         end for
10:    end for
11:    return M[|a|, |b|]
12: end procedure
```

Algoritmo de Wagner-Fischer: Ejemplo

Sean las cadenas $a = sol$ y $b = sal$, apliquemos el algoritmo de Wagmer-Fischer.

En primer lugar construyamos la tabla inicial que iremos llenando:

		s	o	l
	0	1	2	3
s	1			
a	2			
l	3			

Primer paso: para la inserción tendremos $M[0,1] + 1 = 1 + 1 = 2$; eliminación $M[1,0] + 1 = 1 + 1 = 2$; y la sustitución $M[0,0] + \delta(s \neq s) = 0 + 0$ y por tanto $M[1,1] = 0$.

Algoritmo de Wagner-Fischer: Ejemplo

Después de computar los valores de la tabla, esta resulta en:

		s	o	l
	0	1	2	3
s	1	0	1	2
a	2	1	1	2
l	3	2	2	

En el último caso tenemos que:

$$\begin{aligned} M[3,3] &= \min\{M[2,3]+1, M[3,2]+1, M[2,2]+\delta(l \neq l)\} \\ &= \min\{2+1, 2+1, 1+0\} \\ &= 1 \end{aligned}$$

Textos recomendados

Manning, C, Raghavan, P. y Schütze, H. (2008). "1. Boolean Retieval". *Introduction to Information Retrieval*. Cambridge University Press, pp. 1-17.

Clark, S., Fox, C. y Lappin, S. (2013). "1. Formal Language theory". *The Handbook of Computational Linguistics and Natural Language Processing*. John Wiley and Sons, pp. 11-42.

Jurafski, D. y Martin, J. (2018). "2. Regular Expression, Text Normalization, Edit Distance". *Speech and Language Processing*. Pearson.