

CS2702 - Base de Datos II.

Laboratorio 6

Efraín Córdova y Paul Ríos

Lunes 7 de junio 2021



Índice

1	Preprocesamiento	2
2	Indexación	5
3	Consultas	8
3.1	AND	8
3.2	OR	9
3.3	NOT AND	10
3.4	Pruebas	11
4	Anexos	11

Puede encontrar el enlace al repositorio de Github en el Anexo A

1 Preprocesamiento

El algoritmo principal para el preprocesado de un libro de la colección es el siguiente:

```
def readBook(bookNumber):
    stopwords = defineStopwords()
    stemmer = SnowballStemmer(language='spanish')
    with open("Ej1_Preprocesamiento/books/"+str(bookNumber)+".txt") as book,
    open("Ej1_Preprocesamiento/preprocesados/"+str(bookNumber)+".txt", 'w') as result:
        for term in book.read().split():
            term = term.lower()
            term = removeUnnecessarySymbols(term)
            if term not in stopwords:
                term = stemmer.stem(term)
                result.write(term + "\n")
```

Fig1.1 Algoritmo de preprocesamiento

Iteramos por cada palabra del archivo. La *stoplist* que se proporcionó es *case sensitive*, por lo que la primera conversión a la palabra es reducirla a minúsculas. Luego, se remueven los caracteres innecesarios y se filtra que esa palabra, con ambas modificaciones, no se encuentre en la *stoplist*. Esta subrutina viene dada por las funciones:

```
def defineStopwords():
    with open('Ej1_Preprocesamiento/stoplist.txt') as stoplist:
        return nltk.word_tokenize(stoplist.read().lower())

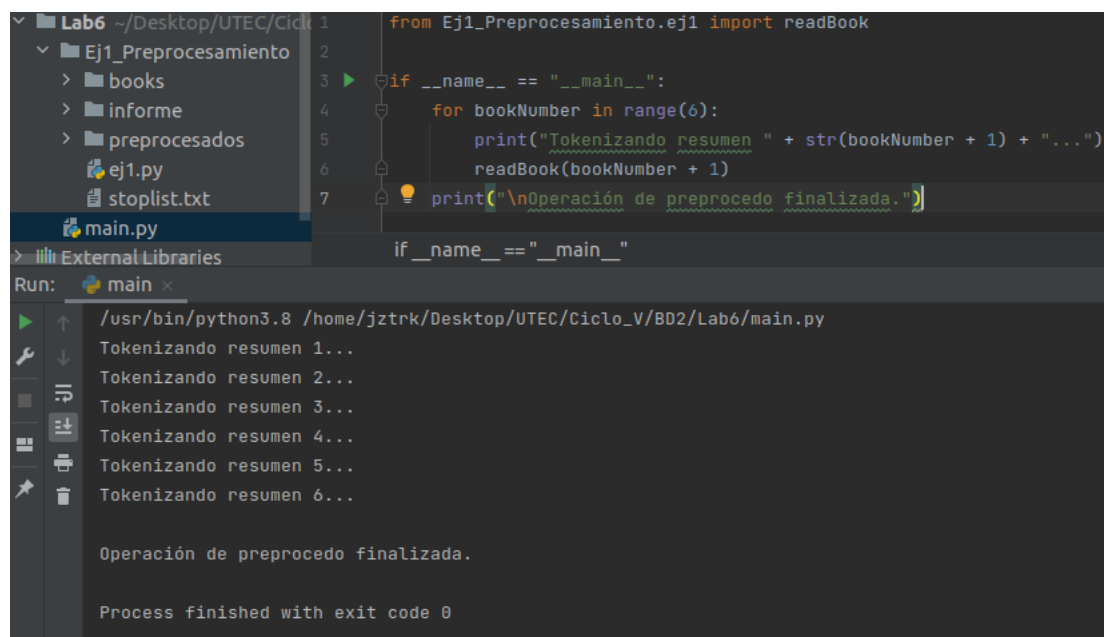
def removeUnnecessarySymbols(word):
    unnecessarySymbols = {"<", ">", ",", ":", ";", "«", "»", ".", "!", "?", "(", ")"}
    filter = ""
    for statement in word:
        if(statement in unnecessarySymbols):
            statement = statement.replace(statement, "")
        filter += statement
    return filter
```

Fig1.2 Subrutinas de normalización y filtrado de palabras

Finalmente, solo nos queda utilizar alguno de los *stemmers* de NLTK para reducir a su raíz las palabras que pasaron las validaciones. *Porter* es

un *stemmer* desarrollado en 1980 que, a pesar de su popularidad, hoy en día se encuentra desfasado. Por otro lado, *Snowball* es una propuesta actual que aún sigue en constante desarrollo (fue actualizada por última vez en octubre de 2020). Quizás los beneficios que *Snowball* sostiene contra *Porter* no se vean reflejados aún en nuestro idioma, pero creemos que trabajar con la tecnología más desarrollada es más interesante. Decidimos, entonces, realizar el índice bajo las reducciones del algoritmo de *Snowball*.

Aclaremos en párrafos anteriores que el algoritmo de preprocesamiento trabaja sobre un único resumen. Basta construir el siguiente main para efectuar la operación con toda la colección:



```
1 from Ej1_Preprocesamiento.ej1 import readBook
2
3 if __name__ == "__main__":
4     for bookNumber in range(6):
5         print("Tokenizando resumen " + str(bookNumber + 1) + "...")
6         readBook(bookNumber + 1)
7     print("\nOperación de preprocedo finalizada.")
8
9 if __name__ == "__main__":
```

Run: main

```
/usr/bin/python3.8 /home/jztrk/Desktop/UTEC/Ciclo_V/BD2/Lab6/main.py
Tokenizando resumen 1...
Tokenizando resumen 2...
Tokenizando resumen 3...
Tokenizando resumen 4...
Tokenizando resumen 5...
Tokenizando resumen 6...

Operación de preprocedo finalizada.

Process finished with exit code 0
```

Fig1.3 Main para el ejercicio de preprocesamiento

Solo para esta parte del laboratorio, escribiremos sobre los archivos preprocesados las palabras removidas de forma que sea evidente el funcionamiento de las funciones de validación. A continuación, mostraremos unas pocas palabras del primer archivo preprocesado, pero puede encontrar los 6 archivos completos en el Anexo B. Recuerde que para los siguientes algoritmos usaremos archivos sin las palabras removidas.

```
[Removed: la]
obra
comienz
[Removed: con]
[Removed: la]
notici
[Removed: de]
[Removed: la]
celebr
[Removed: del]
111
cumpleañ
[Removed: de]
bilb
bolson
[Removed: en]
[Removed: la]
comarc
[Removed: sin]
embarg
[Removed: para]
bilb
[Removed: esta]
[Removed: gran]
fiest
[Removed: tenía]
[Removed: como]
motiv
principal
[Removed: su]
part
[Removed: hacia]
[Removed: su]
[Removed: último]
viaj
product
[Removed: del]
```

Fig1.4 Fragmento del 1er libro preprocesado y las palabras removidas

El algoritmo es $O(n)$, aunque no lo parezca a simple vista. Consideremos que el *stemmer* hace su cometido en no más de n operaciones y que la *stoplist*, al ser un recurso estático, puede usarse iterativamente con un coste constante.

2 Indexación

Explicaremos paso a paso el algoritmo propuesta para la construcción del índice

```
1 def buildIndex():
2     load = {}
3     for bookNumber in range(6):
4         with open("Ej1_Preprocesamiento/preprocesados/" + str(bookNumber + 1) + ".txt", 'r') as book:
5             temp = {}
6             for term in book.read().split('\n'):
7                 if term not in temp:
8                     if term not in load:
9                         load[term] = ((bookNumber + 1),)
10                    else:
11                        newTuple = list(load[term])
12                        newTuple.append(bookNumber + 1)
13                        load[term] = tuple(newTuple)
14                temp[term] = True
15     keys = sorted(load.items(), key=lambda x: len(x[1]), reverse=True)
16     keys = keys[:501]
17     keys.sort(key=lambda x: x[0])
18     with open("Ej2_Indexación/inverted_index.txt", 'w') as index:
19         for value in keys[1:]:
20             index.write(value[0] + ':')
21             books = ''
22             for book in value[1]:
23                 books += str(book) + ','
24             index.write(books[:-1] + '\n')
```

Fig2.1 Algoritmo de indexación por libros

Las líneas 3-14 manejan la indexación propiamente dicha. Tenemos un diccionario para todas las colecciones (*load*) y un temporal (*temp*) para no añadir más de una instancia de una palabra para un mismo libro (e.g Si la palabra "Frodo" aparece un total de 3 veces en el archivo 1, nos interesa guardar una única vez que "Frodo" apareció en el archivo 1). Lo que hacemos es insertar en la *key* de la palabra procesada una actualización de la tupla que se halla en su valor de forma que ahora se almacene el número del libro en la tupla principal en *load*. Si no existe una *key*, entonces la inicializamos con una tupla de un único elemento correspondiente al número de libro que se esta procesando. Lo demás son cuestiones técnicas del lenguaje, en particular se quería trabajar con un diccionario de listas, puesto que el coste, si bien constante, de expandir una lista es menor que el de volver a crear una tupla, la cual es inmutable.

La siguiente parte del código efectúa los requisitos que se le piden al índice. La línea 15 ordena las palabras según la cantidad de libros en las que aparecen, la línea 16 actualiza la lista de tuplas para que solo se almacenen los 500 primeros y la línea 17 hace el ordenamiento lexicográfico. Las últimas líneas se encargan de escribir apropiadamente en el índice según el formato que se pidió.

Mostraremos un fragmento del resultado. Puede ver el archivo completo en el Anexo B.

```
111:1
abism:2,3
acab:1,4,5,6
accidental:1
accion:3
acebed:2
acerc:2
acompañ:1,2,5,6
aconsej:3
acontec:2,3
actos:3
acud:1,2,5,6
adentr:1
agu:2
alcanz:2
alde:1
alert:1,4,5
alli:1,2,5,6
amenaz:4
amig:1,2,3
amo:2,4,6
amon:1,2
anduín:1,2,5
anill:1,2,4,6
aniquil:2
antigu:2
aparec:2,4
aparent:3,4
apart:2
apen:2
apres:3
apresur:3,5
aproveh:3,6
aragorn:1,2,3,5,6
ardid:3
argument:4
armas:1
arnor:1
arrastr:1,2
arreat:1
arroj:2,3,6
asamble:3
ascend:4,6
asent:2
asfaloth:1
atac:1,2,4,5,6
ataqu:3,5
atrap:1,3,4
atraves:2,4
ayud:1,3,5,6
```

Fig2.2 Fragmento del índice

Los algoritmos de ordenamiento aplicados en las líneas 15 y 17 son $O(n \lg(n))$. El resto de líneas entre 17-24 tienen un coste lineal y la línea 16 es constante. Lo que quizás pueda parecer preocupante son las líneas 6,7 y 8, que parecieran ocasionar que el algoritmo propuesto sea tan ineficiente como $O(n^3)$. No podemos argumentar que la línea 6 no provoca que todo lo anterior a la línea

15 no sea al menos lineal, pero sabemos que el coste de búsqueda en un diccionario esta dado por una función *hash* que, en el caso de Python, es $O(1)$ en su mejor caso. ¿La implementación propuesta mantiene al diccionario tal que su función de mapeo sea siempre constante?. Lo que puede causar que el algoritmo se salga de este caso es que existan muchas colisiones, pero puede notar que el algoritmo nunca poseerá ninguna. Es justamente la existencia de una *key* con ese mismo valor lo que impedirá que las líneas 8-13 se ejecuten luego de la primera llamada. El condicional de la línea 8 también consultará en tiempo constante la existencia de la *key*. Aquí podríamos tener colisiones, pero no más de 6, ya que la única forma de encontrar una palabra es que la única vez que se insertó en la tupla temporal fue en un archivo pasado. Podemos tomar ese escenario como constante y notar, irónicamente, que el algoritmo es $O(n \lg(n))$, con la salvedad del comportamiento de los diccionarios en Python.

3 Consultas

En la parte de las consultas mediante métodos booleanos, como nos han especificado, necesitamos que nuestras funciones de comparación sean lineal, o sea, $O(n)$. Aplicando las siguientes funciones: AND, NOT AND y OR, hemos aprovechado de que en todas usamos listas ordenadas.

3.1 AND

```
def AND(l1 : list , l2: list ):
    size_1 = len(l1)
    size_2 = len(l2)

    res = []
    i, j = 0, 0

    while i < size_1 and j < size_2:
        if l1[i] < l2[j]:
            i += 1
        elif l1[i] > l2[j]:
            j += 1
        else:
            res.append(l2[j])
            i += 1
            j += 1

    return res
```

Como recibimos 2 listas para poder comparar, en este caso, empezamos desde el índice 0 en ambas listas e iteramos hasta que alguno de ellos termine de pasar por alguna de las listas, porque debido a esto, ya no habría que poder comparar. Entonces, lo que nosotros queremos comparar, son elementos iguales, por lo tanto, si es que alguno de nuestras comparación es menor, entonces vamos al siguiente elemento. En caso de que ambos sean iguales, entonces, lo añadimos a una lista respuesta("res") y seguimos con los siguientes términos en ambas listas.

3.2 OR

```
def OR(l1 : list , l2: list ):
    size_1 = len(l1)
    size_2 = len(l2)

    res = []
    i, j = 0, 0

    while i < size_1:
        if j == size_2:
            return res + l1[i:]
        if l1[i] < l2[j]:
            res.append(l1[i])
            i += 1
        elif l1[i] > l2[j]:
            res.append(l2[j])
            j += 1
        else:
            res.append(l1[i])
            i += 1
            j += 1
        # print(i)
        # print(j)

    return res + l2[j:]
```

De la misma manera, por la parte del OR, iteraremos primero por la toda la lista 1, si en algún momento j (índice en la lista 2), recorre todos, entonces solo faltaría añadir los restantes en la lista 1, si comparamos $l1[i]$ con $l2[j]$, añadimos el menor y vamos al siguiente del índice que hemos añadido. Si son iguales, añadimos cualquiera y vamos al siguiente en ambos. En caso hallamos terminado de recorrer toda la lista 1, retornamos la respuesta y lo que sobra de la lista 2.

3.3 NOT AND

```
def AND_NOT(l1 : list , l2: list ):
    size_1 = len(l1)
    size_2 = len(l2)

    res = []
    i, j = 0, 0

    # e1
    while i < size_1:
        if j == size_2:
            return res + l1[i:]
        if l1[i] < l2[j]:
            res.append(l1[i])
            i += 1
        elif l1[i] > l2[j]:
            j += 1
        else:
            i += 1
            j += 1
        # print(i)
        # print(j)

    return res
```

En este caso, trabajamos sobre la lista 1, de la manera de que, queremos solo que hay en la lista 1 que no este en la lista 2. En este caso, recorremos toda lista 1. Si llegamos a un punto en que, por otro lado, el índice 2 ya ha recorrido toda la lista 2. Entonces, retornamos lo que ya habíamos tenido previamente, añadiendo lo que sobre en la lista 1. En otra parte, si nuestro elemento i en la lista 1, es menor al elemento j en la lista 2, añadimos el elemento i de la lista, y vamos al siguiente. Si en caso tenemos que nuestro elemento i de la lista 1, es mayor que la del elemento j de la lista 2, iteramos al siguiente de la lista 2. Si en caso son iguales, iteramos al siguiente en ambos índices. Terminando de recorrer toda la lista 1, retornamos la respuesta.

3.4 Pruebas

```
adduser@LAPTOP-NQIGB8NG:/mnt/d/atom/bd2/bd2_inverted_index/Ej3_queries$ python3 ej3.py
lleg: ['1', '2', '3', '4', '5', '6']
logr: ['1', '2', '5', '6']
irse: ['1']
tirith: ['2', '3', '4', '5', '6']
parth: ['2', '3', '5']

(lleg AND logr) AND NOT irse
['2', '5', '6']
(irse OR tirith) AND logr
['1', '2', '5', '6']
(parth OR irse) AND (irse OR logr)
['1', '2', '5']
```

4 Anexos

Anexo A: Repositorio de Github

Anexo B: Preprocesamiento con palabras removidas e índice invertido