CS2702 - Base de Datos II

Proyecto 2

Recuperación de Documentos de Texto

Córdova Amaya, Efraín
Soto Aguirre, David
Rios Vasquez, Paul

Índice general

1. Introducción

Se pide crear un motor de búsqueda en lenguaje natural para una base de datos de tweets relacionados a un contexto particular. El puntaje de cada tweet será obtenido con la similitud de coseno aplicada a un índice invertido en memoria secundaria.

2. Backend

2.1. API de recuperación de tweets

Para la recuperación de los tweets, hemos encontrado una herramienta que nos permite recuperar fácilmente 20k tweets o más en un tiempo de alrededor 5 min (anteriormente era un poco menor, pero hemos decidido obtener la posición de cada tweet en otro archivo para poder acceder de una manera más eficiente). Esta es snscrape, que funciona simulando la barra de búsqueda de twitter. Esto nos permite obtener distintos parámetros de búsqueda y usarlos en la data, de forma que obtenemos más información de un tweet, además de que podemos modificar aspectos como el rango entre fechas, obtener aquellos que no tienen link, o filtrar solo los que son tweets (excluyendo a los retweets). Incluso, el usuario puede cambiar el tema de la data que se almacena si así lo desea.
Puede encontrar información un poco más técnica sobre la API y como la estamos utilizando en el README del github del proyecto

2.2. Preprocesamiento

El proceso de preprocesamiento viene dado por la clase **TweetProccesor**, que carga la stoplist y un stemmer de SnowballStemmer (un stemmer, actualmente en desarrollo para soportar lenguaje natural en español) y proporciona una función (*tokenize*) para tokenizar un texto. El proceso de tokenizado de un tweet empieza con el filtrado de las stopwords, para luego reducir las palabras restantes a su lexema en español. El resultado de esta operación es un arreglo con las palabras reducidas del texto ingresado como input.

```
#Para cada palabra en el tweet: si no es una stopword, la reduce a su raíz y la añade a un contenedor
#Al final, se retorna ese vector
#Complejidad: O(n*K) donde K es la cantidad de palabras en la stoplist.
def tokenize(self, tweet):
    #Remover los emojis.
    tweet = tweet.encode('ascii', 'ignore').decode('ascii')
    return [self.stemmer.stem(t) for t in nltk.word_tokenize(tweet.lower()) if t not in self.stoplist]
```

Figura 1: Función TweetProcessor.tokenize(tweet)

2.3. Construcción del Índice Invertido

Para no sobrecargar la memoria principal, debemos dosificar la cantidad de tweets sobre los que se operan. Usaremos un tamaño de bloque o bucket, de forma que

construiremos el índice uniendo índices más pequeños que puedan ser llevados en estos bloques. Hemos tomado un tamaño de bloques de 500 tweets para una base de datos de, inicialmente, 20000 tweets. Para cada uno de estos bloques, construimos un índice temporal siguiendo los siguientes pasos:

1. Tokenizar cada tweet que ingrese al bucket y guardar su longitud
2. Leer cada palabra de la tokenización. Guardar en un contenedor las tuplas (frecuencia de término, id de tweet). Actualizar la frecuencia de término cuando se vuelva a leer la misma palabra en el mismo tweet.
3. Unimos todos los tweets leídos en el bloque en un contenedor similar al usado en el paso 2. Podemos aprovechar para amortiguar los pesos con la fórmula de logaritmo.
4. Leídos todos los tweets que caben en el bloque, proceder a guardar las frecuencias recuperadas: document frecuency (DF) y term frecuency (TF). Note que DF es justamente la longitud del arreglo TF y que aún no calculamos IDF, debido a que ese parámetro se calcula en toda la colección y un bloque solo es una pequeña parte de esta.

Nuestra definición de inverted index es un poco distinta a la vista en clase, debido a que no calculamos el IDF directamente en el índice, sino en la propia función scoring. De todas formas, la frecuencia DF, que es la que guardamos en su lugar, permite sacar el IDF siempre que tengamos acceso al tamaño de la colección, que hemos definido en la primera pasada como 20000. La implementación de los pasos mencionados se realizó de la siguiente forma:

```
term_freq = {}
for i in range(block):
    tweet_data = file.readline()
    tweet_data = json.loads(tweet_data)
    tokens = tp.tokenize(tweet_data["content"])
    lengths[tw_index] = len(tokens)
    terms = {}
    for token in tokens:
        if terms.get(token) == None or terms.get(token)[1] != tw_index:
            terms[token] = (1, tw_index)
        else:
            terms[token] = (terms[token][0] + 1, tw_index)
    #Podemos amortiguar el TF con el logaritmo. No podemos hacer esto con DF, ya que no
    #tenemos la información del resto de indices en los otros bloques.
    for term in terms:
        if term_freq.get(term) == None:
            term_freq[term] = [(1 + math.log10(terms[term][0]), terms[term][1])]
        else:
            term_freq[term].append((1 + math.log10(terms[term][0]), terms[term][1]))
    tw_index += 1
inverted_index = {}
for word in term_freq:
    #Guardamos las frecuencias de cada termino
    inverted_index[word] = {
        "DF": len(term_freq[word]),
        "TF": term_freq[word]
    }
```

Figura 2: Construcción de índices temporales

2.3.1. Manejo de memoria secundaria

Usamos la técnica del *Block sort-based index* (BSBI) para crear el índice en memoria secundaria.
Hemos visto en el apartado anterior como podemos usar un bloque para crear índices locales. Ahora, debemos leer esos índices, unirlos, removerlos de la memoria y escribir en limpio el resultado: el índice invertido de toda la colección. La sintesís de los índices temporales es realizada por la función merge de la clase **InvertedIndex**

```
def __merge(self):
    # arr = os.listdir(path)
    inverted_index = {}

    #Leemos los índices intermedios
    for f_name in glob('./indexs/*.json'):
        with open(f_name, "r") as index:
            i_dic = index.readline()
            i_dic = json.loads(i_dic)
        #Colocamos las frecuencias. Recuerde que el IDF se calcula en la misma función de similitud
        for k in i_dic.keys():
            if k not in inverted_index.keys():
                inverted_index[k] = i_dic[k]
            else:
                inverted_index[k]['TF']+=(i_dic[k]['TF'])
                inverted_index[k]['DF']+=(i_dic[k]['DF'])
        os.remove(f_name)

    #Escribimos el índice completo
    json_file = open('resources/i_index.json', 'a', newline='\n', encoding='utf8')
    json_file.truncate(0)
    json_file.write(json.dumps(inverted_index, ensure_ascii=False, default=str))
```

Figura 3: Construcción del índice global

Podemos abstraer el algoritmo de construcción de indíces temporales en una función *build_inverted_index* que reciba como parametros el tamaño de la colección y el tamaño de bloques. Luego, usamos la función *merge* para sintetizarlo en único archivo. Note que estos dos procesos son todo lo que haría el BSBI, el cual es creado siempre que el contexto de los datos cambie (en el entregable, se le da tweets relacionados a la keyword 'covid'. Puede ir al apartado *Frontend* para una explicación detallada del cambio de temática en el proyecto).
Nuestro algoritmo de índice en bloques quedaría así:

```
def BSB_index_construction(self):
    block = 500
    self.__build_inverted_index(size_tweets, block)
    self.__merge()
```

Figura 4: Construcción del índice global

2.4. Consultas

2.4.1. Lógica y scoring

La fórmula de similitud de coseno para el score de un tweet t_k) que contiene alguna palabra de la query es:

$$Score(t_k) = \sum_{q \,\epsilon\, Q, t_k} \frac{1 + log(freq(q) * \log(\frac{n}{index[q]['DF']}))}{len(Q)} * \frac{index[q]['TF'] * \frac{n}{index[q]['DF']})}{len(t_k)}$$

Considere que, al usar el término Q, nos referimos al arreglo de tokens que nos devuelve tokenize(Query). También, considere *freq(q)* como la cantidad de veces que se repite q en Q, que, por cierto, tambien esta en t_k. Eso, en código, se ve asi:

```
#Obtenemos la frecuencia de las palabras en la query
query_words = {}
for q in tokens:
    if query_words.get(q) == None:
        query_words[q] = 1
    else:
        query_words[q] += 1
```

Figura 5: Frequencia de una palabra en la query

El cálculo como tal se hace para cada documento que contiene alguna palabra en la query, como se puede ver en esta imagen:

```
#Calculamos la distancia de coseno:
for q in query_words:
    if i_dic.get(q) != None:
        i = i_dic[q]
        #Normalizamos localmente el vector de palabras en el query
        qq = round((1 + math.log10(query_words[q])) * (math.log10(n /i['DF'])) / norm, 4)
        for tweet in i['TF']:
            #Sumamos a un documento el puntaje que va consiguiendo (con dist. de coseno)
            cosine = round(tweet[0] * (math.log10( n /i['DF'])) / self.lengths[tweet[1]] * qq, 4)
            if tweets.get(tweet[1]) == None:
                tweets[tweet[1]] = cosine
            else:
                tweets[tweet[1]] += cosine
heap = []
```

Figura 6: Scoring

Le entregaremos al usuario un número k de tweets que son relevantes a su consulta según el scoring propuesto. Dado que ese k usualmente es considerablemente menor a n, nos conviene utilizar una fila de prioridades en lugar de efectuar un ordenamiento a través del scoring:

```
heap = []

#Debemos hacer una segunda pasada porque los cosenos podrían haberse modificado...
for tweet in tweets:
    heappush(heap, (-1 * tweets[tweet], tweet))

#No queremos mostrar los 20K tweets al usuario, asi que nos quedamos solo con los K más relevantes
retrieved = {}
for i in range(min(k,len(tweets))):
    retrieved[heap[0][1]] = -1 * heap[0][0]
    heappop(heap)
    heapify(heap)

return retrieved
```

Figura 7: Obtención de los k tweets más relevantes

Esta recuperación tiene un coste de $O(2n + k * lgn)$ (note que es 2n por el segundo for), frente al coste de realizar un ordenamiento, el cual es $O(nlg(n))$. Para notar la mejora, considere k = 64 en el siguiente calculo (y n = 20000 en este y los demás):

SORTING: 20000 * lg(20000) = 285754.2476
K-HEAP: 2 * 20000 + 64*lg(20000) = 40914.4136
FACTOR DE MEJORA = SORTING/K-HEAP = **6.9842**

Como puede ver, el uso de fila de prioridades es casi 7 veces más rápido que el sorting para valores no muy grandes de k (menores o iguales a 64).
Seamos honestos con nuestra 'optimización'. Es claro que para valores cercanos a n = 20000 nuestra fila de prioridades tiene un peformance súboptimo, en concreto, una cantidad a lo sumo 2n de operaciones adicionales. Seamos aún más precisos, considere k = 17201 en el siguiente cálculo:

SORTING: 20000 * lg(20000) = 285754.2476
K-HEAP: 2 * 20000 + 17201*lg(20000) = 285762,9406
FACTOR DE MEJORA = SORTING/K-HEAP = **0.9999**

Díficilmente usted o cualquier otro usuario podría atinarle a un número tan aparentemente arbitrario como 17201. Lo que sí es más probable es que se quiera un ordenamiento total de los archivos en función a la query. Considere un k = n en:

SORTING: 20000 * lg(20000) = 285754.2476
K-HEAP: 2 * 20000 + 20000*lg(20000) = 325754.2476
FACTOR DE MEJORA = SORTING/K-HEAP = **0.8779**

Las filas de prioridades ahorran muchísimo tiempo para k muy pequeños, pero ese valor se degrada conforme avanza k. En la práctica, y más en algoritmos que se deben particionar como el bsbi, quizás haya que hacer alguna operación para construír el heap. Eso, a la larga, puede ser nocivo para el tiempo de respuesta de la aplicación. Esta optimización solo sirve para n = 20000, pero podemos generalizar el estado de optimalidad del K-heap (aquellos valores de k tal que para cualquier valor menor a k el coste de construír el K-heap es menor a un sorting del diccionario de tweets) con la siguiente operación booleana:

$$Optimalidad\_de\_k = \frac{(n-k) * lg(n)}{n} > 2$$

Si la optimialidad es falsa, quiere decir que nos conviene usar sorting. La formula toma la constante de crecimiento de la diferencia entre $2n + k * lg(n)$ y $n * lg(n)$ y devuelve verdadero si la segunda es mayor a la primera en un k específico y falso en caso contrario. Si aún no esta convencido, puede probar la fórmula con n = 20000 y k = 17201 para ver que la fórmula queda $(1.999 > 2)$.
Coloquemos esta validación en el código:

```
#k critico para el cual a partir de k+1 la fila de prioridades es peor que el sorting
#Ver informe para un analisis preciso de esta situación.
if ((n - k) * math.log2(n) / n) > 2:
    #Debemos hacer una segunda pasada porque los cosenos podrían haberse modificado...
    for tweet in tweets:
        heappush(heap, (-1 * tweets[tweet], tweet))

    #No queremos mostrar los 20K tweets al usuario, asi que nos quedamos solo con los K más relevantes
    retrieved = {}
    for i in range(min(k,len(tweets))):
        retrieved[heap[0][1]] = -1 * heap[0][0]
        heappop(heap)
        heapify(heap)
    return retrieved

else:
    return dict(sorted(tweets.items())[:k])
```

Figura 8: Optimización de obtención de los k tweets más relevantes

2.4.2. Recepción de resultados

Las querys las efecuta la función *do_query*, que le pide al usuario insertar su query y un número de tweets a recuperar:

```
#Procesar consulta
def do_query():
    q = input("Ingrese la query que quiere obtener: ")
    qns = int(input("Ingrese cuantos tweets quiere recuperar, como máximo: "))
    rpta = process_query(q,qns,size_tweets)

    #Lectura de indice y de tweets
    indexs = open("../data/index.txt", "r")
    tweets = open("../data/data.json", "r")
    jsonrpta = {}
    cont = 0

    for n in rpta:
        indexs.seek(0,0)
        tweets.seek(0,0)
        indexs.read((n-1)*10)
        line_tweet1 = int(indexs.read(10))
        line_tweet2 = int(indexs.read(10))
        tweets.read(line_tweet1)
        json_line = tweets.read(line_tweet2 - line_tweet1-1)
        json_line = json.loads(json_line)
        json_line['score'] = rpta[n]
        jsonrpta[cont] = json_line
        cont +=1

    #Escritura de los tweets que van al frontend en un .json
    json_file = open('resources/rpta.json', 'a', newline='\n', encoding='utf8')
    json_file.truncate(0)
    json_file.write(json.dumps(jsonrpta, indent = 6 , ensure_ascii=False))
    return
```

Figura 9: Manejo de consultas

En el siguiente apartado, presentaremos una interfaz de usuario que represente los resultados de forma amigable. Por el momento, hagamos consultas por terminal y obserbemos los archivos generados.
Probemos, por ejemplo, la siguiente consulta en una base de datos recuperada con la keyword **covid**.

```
jztrk@HP-Cipher3  ~/Desktop/UTEC/Ciclo V/BD2/BD2 P2/Backend  main ±  python3 inverted_index.py
Ingrese la query que quiere obtener: cifras de vacunacion peru junio 2021
Ingrese cuantos tweets quiere recuperar, como máximo: 10
```

Figura 10: Ejemplo de query en terminal

Los tweets con mayor scoring fueron escritos en **rpta.json**:


```
{
    "0": {
        "id": 1407312983861125134,
        "username": "iveth_6_76",
        "date": "2021-06-22 12:22:16+00:00",
        "content": "Hoy 22 de Junio del 2021  Día de vacunación 💉 contra el Covid 19 🙌",
        "url": "https://twitter.com/iveth_6_76/status/1407312983861125134",
        "score": 0.2198
    },
    "1": {
        "id": 1408250472880824321,
        "username": "galli0226",
        "date": "2021-06-25 02:27:31+00:00",
        "content": "Qué tal la cifra de hoy casos COVID19",
        "url": "https://twitter.com/galli0226/status/1408250472880824321",
        "score": 0.1991
    },
    "2": {
        "id": 1409685559719579652,
        "username": "hjaimejavier",
        "date": "2021-06-29 01:30:02+00:00",
        "content": "10:53 del 28 de junio del 2021 me vacunaron COVID 19",
        "url": "https://twitter.com/hjaimejavier/status/1409685559719579652",
        "score": 0.174
    },
    "3": {
        "id": 1408090411718135808,
        "username": "Edsito32",
        "date": "2021-06-24 15:51:29+00:00",
        "content": "Se tiene que erradicar dos cosas en el Perú para poder vivir en paz El Covid19 y el Fujimorismo",
        "url": "https://twitter.com/Edsito32/status/1408090411718135808",
```


Figura 11: Algunos tweets recuperados de la consulta anterior

En el video adjunto mostraremos el proceso de construcción del índice, así como la representación en frontend de los datos en rpta.json

3. Frontend

Esta es la vista principal de la página del proyecto, donde puede elaborar sus consultas y ver los tweets recuperados:

Figura 12: Frontend - Consultas

Esta es la vista secundaria, donde puede cambiar la temática de la base de datos y cargar n tweets relacionados a esa temática.

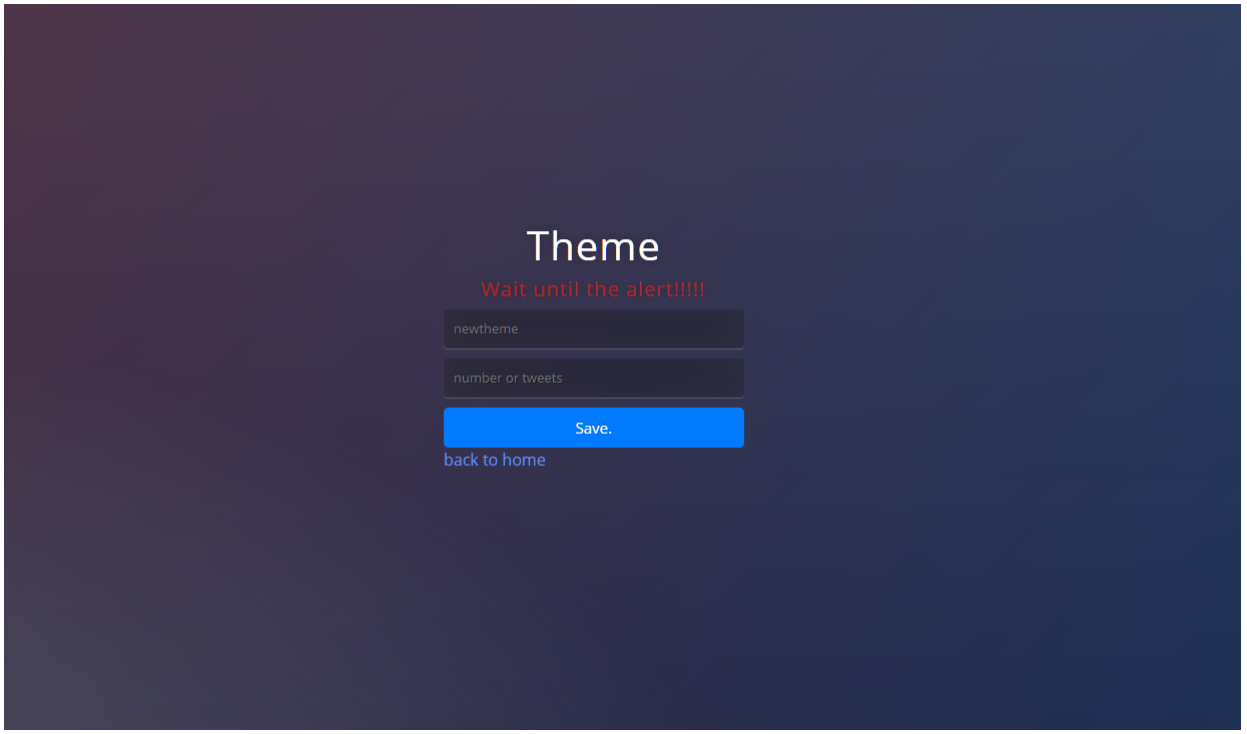

Figura 13: Frontend - Cambio de temática

Las funcionalidades relacionadas al objetivo del proyecto que ofrecemos son:

- El usuario puede realizar consultas de lenguaje natural en español en una base de datos de 20k tweets relacionadas a una temática particular.
- El usuario puede seleccionar el número de tweets a recuperar. La recuperación de tweets esta optimizada para siempre ser O(n*lgn), aunque para k pequeños no es mayor a O(n + k*lg(n))
- El usuario puede cambiar la temática de los 20k tweets a disposición en la base de datos. Esto implica crear un nuevo índice y conectarse con la API. Este último paso no es recomendable hacerlo en intervalos cortos de tiempo; por favor, sea discreto con el uso de esta funcionalidad.

Se realizo un deploy en Heroku para que pueda acceder desde un servidor. Puede acceder a este con **este enlace**

Anexos

1. Github del proyecto
2. Artículo de uso de la API de recuperación
3. Stoplist utilizada en preprocesamiento
4. Stemmer utilizado en preprocesamiento