Primer Informe Proyecto SRI
Querch

Lázaro Daniel González Martínez C311

1 Introducción

El presente informe contiene una breve explicación del estado actual del proyecto final orientado en la asignatura "Recuperación de Información". De forma general consiste en la implementación de un sistema de recuperación donde el usuario pueda interactuar mediante consultas y tal vez retroalimentación. Para esta primera entrega se implementó solamente un modelo: el modelo booleano. A continuación se explicará la implementación técnica del mismo, se indicarán los archivos donde se encuentren las principales implementaciones y se describirá la arquitectura concebida para el sistema completo.

2 Modelo Seleccionado

Para comenzar decidimos implementar un Modelo Booleano. Es sabido que este modelo considera los documentos como vectores binarios por cada término en la consulta; y la consulta es una expresión booleana con los términos como sus átomos. La idea es simple: dada una consulta evaluar los vectores documentos con la expresión booleana. La similitud entre documento y consulta es 1 si existe algún componente de la Forma Normal Disyuntiva (FND) que sea igual al vector del documento. En caso contrario 0. Note que la similitud expresada de esta forma trabaja con la FND de la consulta, lo cual presupone un proceso extra; sin embargo más adelante se propondrá una solución para obviar esto.

3 Arquitectura

Para implementar el Sistema de Recuperación de Información se consideró dividir la recuperación en 2 procesos: Preprocesado de documentos, y Modelación del sistema. Cada fase funciona de manera independiente, aunque en la práctica se haga un Preprocesado primero y luego el Modelado. Sin embargo, se consideran independientes porque ninguno depende técnicamente del otro.

La fase de Preproceso se encarga de extraer datos de los conjuntos de documentos y crear estructuras con estos, que se almacenarán en la memoria física del sistema. Este proceso suele ejecutarse una sola vez teniendo el corpus entero, o se ejecuta sobre cada subconjunto de documentos que se agregue al corpus. Entonces no solo se considera el espacio del corpus, sino también el espacio de datos extraídos propio de un modelo particular, en nuestro caso del Modelo Booleano. Esto posibilita, que no haya necesidad de estar extrayendo los datos[1] que necesita el Modelo, durante la petición de la consulta.

En el Modelado, el sistema de búsqueda carga los datos extraídos del Preprocesado y pone a funcionar el modelo, que tendrá los datos cargados en memoria RAM. Notemos que son los datos extraídos y almacenados en la memoria física; por esto es que se consideran procesos independientes, ya que teniendo los datos extraídos, podemos ejecutar el Modelo sin tener que hacerle un preprocesamiento primero a los documentos. Durante esta fase, se espera constantemente recibir una consulta, para luego pasarla al modelo, y obtener los resultados de la recuperación de información.

4 Preprocesado del corpus

Para una mayor comodidad se trabajó con el conjunto de datos que proporciona el módulo `ir_datasets` de `Python`. Son conjuntos enfocados para la creación y evaluación de SRI de documentos. De la base de datos que contiene el módulo se utilizó:

- `beir/arguana: 8674,`
- `beir/cqadupstack/android: 22998,`
- `beir/cqadupstack/english: 40221,`
- `beir/cqadupstack/gaming: 45301,`
- `beir/cqadupstack/gis: 37637,`
- `beir/cqadupstack/mathematica: 16705,`

[1] Solo los datos que son independientes de la consulta. Por ejemplo, la ocurrencia de términos en documentos, la frecuencia de términos en documentos, ...

- `beir/cqadupstack/physics: 38316,`
- `beir/cqadupstack/programmers: 32176,`
- `beir/cqadupstack/stats: 42269,`
- `beir/cqadupstack/tex: 68184,`
- `beir/cqadupstack/unix: 47382,`
- `beir/cqadupstack/webmasters: 17405,`
- `beir/cqadupstack/wordpress: 48605,`
- `beir/dbpedia-entity`[2]`: 4635922,`
- `beir/fiqa: 57638,`

Se consideró preprocesar conjuntos de datos fuera de este módulo, tales como los proporcionados por los profesores: `reuters` y `20news`. Sin embargo al ser conjuntos con estructuras distintas, requieren un cargado de documentos particular, y se quería probar varios conjuntos para la pre-entrega. No obstante para la entrega final, se incluirán estos.

4.1 Preprocesado de un documento

Teniendo cargado un documento, analizamos su cuerpo para extraer los datos que necesitamos. Como estamos trabajando con el Modelo Booleano, solo necesitaremos extraer la ocurrencia de términos. Para ello, se almacenará en un `set` dichos términos. Sin embargo el cuerpo del documento pasa por un proceso de normalización, donde llevaremos las mayúsculas a minúsculas, y con ayuda del módulo `nltk`[3] normalizaremos algunos de los términos, por ahora de forma sencilla (por ejemplo, convirtiendo plurales en singulares,..., pero no haciendo correcciones a los términos mal escritos). Normalmente se suele eliminar las *palabras vacías* (o *stop-words*), sin embargo se consideró no eliminarlas por ahora del Preprocesamiento. Sin embargo, durante el Modelado, podemos obviarlas o no, a nuestra conveniencia. De ahora en adelante cada vez que se diga término, se considerará a las palabras extraídas de los documentos que pasaron por el anterior proceso de normalización.

Una vez tengamos los términos de cada documento se hace un diccionario invertido de la ocurrencia de términos; es decir, un diccionario donde las llaves son los términos y sus valores correspondientes serían el conjunto de índices de los documentos en los que aparece. Esta consideración es recomendada por [1] ya que más adelante en el evaluado de la consulta, se utilizarán.

4.2 Extracción por bloques

En [1] se recomienda para indexar conjuntos de datos grandes, con memoria RAM limitada: indexar los documentos en bloques. El algoritmo es `BSBIndexConstruction` y se describe en la sección 4. Para extraer los datos se hace una adaptación de este algoritmo y se divide en 2 algoritmos. Uno destinado a la extracción de los datos separados en bloques, y otro para la unión de estos datos.

La extracción por bloques implementada lo que hace es analizar los documentos de un corpus específico, en bloques de un tamaño dado. Por cada bloque se hace el preprocesado de documentos, descrito anteriormente, y se conforma el diccionario invertido. Luego se guarda en la memoria física dicho diccionario. La cantidad de diccionarios creados y almacenados será de:

$$\left\lceil \frac{N}{N_B} \right\rceil = B$$

siendo N la cantidad de documentos total y N_B la cantidad de documentos por bloques.

En el archivo `automatization_extract.py` se tiene la implementación de la extracción en bloques. Por otro lado el archivo `automatization_data.py` se pude ejecutar como un script, con la instrucción:

```
extract [-sb <sizeblock>] ocurrence <namedataset> [<path_result>]
        [-sizeblock <sizeblock>]
```

para extraer la ocurrencia del conjunto de datos con nombre `<namedataset>` en bloques que contengan `<sizeblock>` (1000 por defecto) documentos y ser guardados en un json con nombre `ocurrence.part<i>.json`[4] en la dirección `<path_result>` (`current_path/<namedataset>` por defecto). Ejemplo de esto sería:

```
extract 2000 ocurrence beir/arguana
```

que se traduce a extraer bloques de 2000 documentos del conjunto de datos `beir/arguana`, y guardarlo en la dirección actual.

[2] Este conjunto al ser tan grande se dividió en 99 subconjuntos de 47000 documentos cada uno.
[3] Módulo enfocado en el procesamiento de lenguaje natural
[4] Siendo i el número del bloque al que pertenece.

4.3 Unión de bloques

Para la unión se seleccionan los bloques que se desean unir y se cargan las estructura de diccionario en memoria RAM una a una, mientras se hace una mezcla con un diccionario vacío inicial. Este diccionario que se va conformando es un diccionario unificado, que es resultado de unir los conjuntos de índices de documentos que le corresponde a cada término. Luego este es guardado en memoria física nuevamente.

Entonces para conformar la colección (intermedia) de datos lo que se hará es ejecutar varias instrucciones de extracciones y uniones a nuestra conveniencia. En nuestro caso se trató de tener subconjuntos de no más de 100 000 documentos.

En el archivo `automatization_join.py` se tiene la implementación de la unión de bloques. Se puede ejecutar el archivo `automatization_data.py` como script, con la instrucción:

```
join [-m <mode>] [-b <countblock>] ocurrence <firstname> [<path_files>] [<path_result>]
     [-m <mode>] [-blocks <countblock>]
```

para unificar los datos de ocurrencia en nuevos bloques de tamaño `<countblock>` de bloques con datos extraídos. Se tomarán los bloques guardados en el directorio actual o el especificado por `<path_files>` y los nuevos bloques se guardarán en l carpeta actual o la especificada por `<path_result>` con un nombre con sufijo `...join.part<j>.json`. Es importante señalar que se unificarán los json con nombre `<firstname>.part<i>.json`

5 Modelado del sistema

Al comienzo se explicó de forma breve el funcionamiento del modelo booleano. Una vez preprocesado los documentos, el sistema cargará los datos de ocurrencia de una colección específica, y los tendrá en memoria RAM. Note que leer estos datos la primera vez tiene un costo temporal menor que calcular las ocurrencias de términos en documentos. Estando esta información en memoria se realizarán las operaciones necesarias para poner a funcionar el modelo. Sin embargo, la forma de ejecutar las operaciones no será tal cual indican las fórmulas del modelo, sino mediante consideraciones equivales a este, recomendadas por [1].

5.1 Evaluación de la consulta

Evaluar la consulta según la ecuación de similitud dada al inicio, necesita expresar la consulta en su forma normal disyuntiva y ver si coincide alguna componente disyuntiva con la consulta. Esto implica 2 cosas. Lo primero es que por cada documento se debe realizar la evaluación. Lo segundo, es que se debe expresar la consulta en su forma normal. La misma ecuación de similitud se puede expresar de la forma:

$$sim(q, d_i) = q(d_i)$$

donde q es la expresión booleana considerada como una función de $B^m \to B$ con $B = \{0, 1\}$, y $d_i \in B^m$, sería el vector binario de términos para el documento i. Es bastante sencillo comprobar que ambas son equivalentes, por lo tanto nos ahorraremos para esta entrega una demostración formal. Con esta consideración no tenemos que trabajar con la forma normal disyuntiva de la consulta.

Ahora para eliminar el cálculo de esta similitud por cada documento, daremos un enfoque distinto. Si consideramos los términos como los conjuntos de documentos en los que aparece se puede establecer las equivalencias siguiente:

$$D[t_1 \wedge t_2] \equiv D[t_1] \cap D[t_2]$$
$$D[t_1 \vee t_2] \equiv D[t_1] \cup D[t_2]$$
$$D[\neg t_1] \equiv D[t_1]^c = U - D[t_1]$$

donde $D[t]$ es el conjunto de documentos en el que el término t aparece. La demostración es intuitiva y se aborda en [1]. Por lo tanto, la recuperación dada una consulta se puede realizar de la siguiente forma:

$$D[q(\vec{b})]$$

donde $\vec{b}$ es el vector de términos (genérico) en la consulta. Por ejemplo, dada la consulta:

$$q(animal, beast, rose) = \neg animal \cup (beast \cap rose)$$

el conjunto de documentos que satisface la consulta sería:

$$D[q(animal, beast, rose)] = D[animal]^c \vee (D[beast] \wedge D[rose])$$

La complejidad promedio del cómputo de esto es menor que la de evaluar para cada vector de documentos, su similitud con la consulta. La complejidad se verá influenciada principalmente por las operaciones entre conjuntos, pero por lo general para conjuntos de documentos variados las palabras no vacías suelen aparecer en una cantidad mucho menor que la cantidad de documentos. Un caso sencillo de esto, es el hecho de buscar una sola palabra: en dicho caso la consulta solo tardaría en responder, el tiempo que demore la indexación del documento en el diccionario. Esta evaluación se hace a la par del parseo de la consulta dentro de la clase `ParserBoolean` en el archivo `parser_boolean.py`.

5.2 Parseador de la consulta

Para parsear la consulta se consideró la siguiente gramática

$$\begin{aligned} E \rightarrow &\quad TX \\ X \rightarrow &\quad \wedge E \quad | \quad \vee E \quad | \quad \epsilon \\ T \rightarrow &\quad t \quad | \quad (E) \quad | \quad \neg T \end{aligned}$$

donde los terminales t serían términos. Esta gramática genera un lenguaje para nuestras consultas booleanas, donde cabe destacar que la asociación es a la derecha sin importar si son operaciones `and` u `or`. La implementación del parser tiene la característica de evaluar la consulta durante la ejecución del mismo. Además los caracteres que se usan para las operaciones `and`, `or` y `not` son "&", "|" y " $\sim$ " respectivamente. El parseador trabaja con un diccionario de términos con conjuntos de documentos, y tiene un conjunto universo de documentos. En ambos casos los conjuntos de documentos son conjuntos de sus índices.

5.3 Modelo

La implementación del Modelo se encuentra en el archivo `boolean_model.py` mediante la clase `BooleanModel`. Esta se encarga de cargar los datos intermedios extraídos, en el parseador. A su vez hace de intermediario entre la petición de la consulta y la recuperación de documentos.

Adicionalmente este, permite cambiar de subconjunto de datos. Para ello primero pedirá que se elimine de la memoria RAM los datos y el universo de documentos que el parser tenía. Luego cargará el archivo `.json` pertinente y las estructuras generadas se pasarán al parseador. Esta clase también se encargará de extraer los documentos que se deseen acceder. Por lo general, serán los documentos correspondientes al conjunto de índices que se obtienen como resultado al evaluar la consulta.

6 Interfaz

La implementación del sistema requiere para el usuario final una interfaz gráfica. En esta etapa de desarrollo se implementó una interfaz web mediante el uso de `Django` como framework de desarrollo. La interfaz de inicio solo contiene una barra para escribir las consultas, un botón de búsqueda y un botón para cambiar de conjunto de documentos. El servidor siempre tendrá una instancia del modelo explicado anteriormente, que estará activa en todo momento para que se puedan efectuar consultas sin necesidad de estar leyendo constantemente (de más) de la memoria física. Las 2 páginas actuales solo son esqueletos HTML, lo cual sufrirá modificaciones en las etapas finales del desarrollo.

Por otro lado también se implementó una interfaz por consola, para comprobar de manera más dinámica para los desarrolladores, el funcionamiento del sistema de recuperación. Esta implementación se encuentra en el archivo `boolean_console.py`.

6.1 Requisitos

Los requisitos para ejecutar el software se especifican en el sitio web del repositorio. El repositorio del software se encuentra en `https://github.com/LazaroDGM/Qearch-IR.git`.

7 Conclusiones

Para una futura entrega se tienen pensada varias ideas para implementar independientes de los modelos que se seleccionen:

- Guardar los datos extraídos en archivos binarios diferentes que puedan ocupar menos espacio de memoria física. Actualmente se usan archivos `.json`.
- Almacenar los datos cargados en memoria RAM con estructuras diferentes para valorar el empleo de estas según la eficiencia espacial y temporal. Por ahora se desea experimentar con el módulo `pandas`.
- Extraer de los documentos, URLs (en aquellos donde aparezca) y hacer un rastreo web pequeño para incluir en el corpus nuevos documentos.
- Incluir retroalimentación de los resultados
- Mostrar no tan solo el título de un documento sino los secciones del documento que puedan ser de interés.
- Emplear ontologías para mejorar la recuperación de información, tal vez mediante el empleo de tesauros y revisión de sinónimos en un primer momento.
- Emplear parseadores más potentes que el parseador predictivo descendente.
- Evaluar la calidad de los modelos implementados utilizando conjuntos de consultas etiquetadas, para comparar los resultados, así como analizar las diferencias entre los modelos
- Utilizar varias técnicas de Stemming y Lemmatization para normalizar los términos.

8 Bibliografía

[1] Schütze, H., Manning, C. D., & Raghavan, P. (2008). Introduction to information retrieval (Vol. 39, pp. 234-265). Cambridge: Cambridge University Press.