Universidad Autónoma de Madrid, Escuela Politécnica Superior

Grado en Ingeniería Informática

Búsqueda y Minería de Información 2013-2014

Práctica 2: Implementación de un motor de búsqueda con Apache Lucene

Fechas

- Comienzo: 7 de febrero
- Entrega: 21 de febrero (hora límite 12:00)

Objetivos

Los objetivos de esta práctica son:

- La creación de índices de búsqueda a través de Apache Lucene
- La búsqueda de documentos a través de Apache Lucene

Los documentos que se indexarán y sobre los que se realizarán consultas de búsqueda serán páginas HTML, que deberán ser tratados para extraer y procesar el texto contenido en ellas. Este tema se abordará en profundidad en prácticas posteriores, p.e. eliminando *stopwords* y aplicando de técnicas de *stemming*. En esta práctica bastará con un *parsing* de las etiquetas HTML, a realizar con cualquier librería Java existente.

La implementación de índice y buscador se llevará a cabo mediante una serie de clases y métodos Java que encapsulen funcionalidades ofrecidas por Lucene. Algunas de estas clases y métodos se usarán en prácticas posteriores en las que habrá que implementar índices y buscadores sin Lucene.

Colección de documentos

El índice y buscador de esta práctica se van a construir y evaluar sobre tres colecciones de documentos HTML disponibles en los ficheros **clueweb-1K.tgz** (10 MB), **clueweb-10K.tgz** (82 MB) y **clueweb-100K.tgz** (865 MB), proporcionados junto a este enunciado.

Respectivamente, las colecciones están compuestas de 1.000, 10.000 y 100.000 documentos, y 5, 10 y 100 consultas de búsqueda (*queries*).

Forman parte del conjunto de datos ClueWeb09 (http://lemurproject.org/clueweb09), usado en varios *tracks* de la conferencia TREC (http://trec.nist.gov). Este conjunto de datos se construyó mediante procesos de *crawling* de la Web. Aquí se omiten detalles al respecto por estar fuera del alcance y objetivos de la práctica. Sin embargo, se destaca que el *crawling* es una tarea fundamental en el desarrollo de un motor de búsqueda.

Cada colección está formada por:

- Un fichero comprimido **docs.zip**, con los documentos HTML.
- Un fichero **queries.txt** con las consultas de búsqueda, en el formato `consulta_id:consulta`.
- Un fichero **relevance.txt** con aquellos documentos relevantes a cada consulta. Por cada consulta, se da la lista de documentos siguiendo el formato `consulta_id \t documento_1 \t documento_2 \t ... documento_N`

A la hora de leer los documentos HTML de una colección para su indexado, no será necesario descomprimir docs.zip. En su lugar, desde Java se accederá al contenido de ese fichero comprimido directamente. Para ello se podrán usar clases o librerías externas, como `java.util.zip.ZipInputStream`. Puede consultarse documentación al respecto en:

http://www.thecoderscorner.com/team-blog/java-and-jvm/12-reading-a-zip-file-from-java-using-zipinputstream

http://docs.oracle.com/javase/7/docs/api/java/util/zip/ZipInputStream.html

Ejercicios

Ejercicio 1: Interfaces y clases principales para índices y buscadores [1,5 puntos]

En varias prácticas de la asignatura se van implementar en Java varios indexadores y modelos de búsqueda. En concreto, en esta práctica se desarrollarán un índice y buscador que hagan uso de la API Apache Lucene.

Los diseños de clases y arquitectura son libres. Sin embargo, para poder evaluarlos de forma común, se exige que soporten las siguientes clases e interfaces:

- **Clase `es.uam.eps.bmi.search.TextDocument`** [0,5 puntos], que representará la estructura básica de un documento de texto. Como mínimo tendrá los siguientes métodos:
  - **String getId()**, que devolverá un identificador único de documento.
  - **String getName()**, que devolverá el nombre (*path* completo) del documento.
  - **boolean equals(Object object)**, que sobre-escribirá el método equals de Object comparando identificadores de documentos.
  - **int hashCode()**, que sobre-escribirá el método hashCode de Object devolviendo el código hash del identificador del documento.
- **Clase `es.uam.eps.bmi.search.ScoredTextDocument` (*implements* `java.lang.Comparable`)** [0,5 puntos], que guardará el *score* (generalmente, resultado de una búsqueda) de un documento de texto dado. La clase deberá tener al menos los siguientes métodos:
  - **String getDocumentId()**, que devolverá el identificador del documento asociado al resultado.
  - **double getScore()**, que devolverá el *score* asociado al resultado.
- **Clase `es.uam.eps.bmi.search.indexing.Posting`** [0,5 puntos], que representará la estructura básica de *posting* usado por un índice. Estará asociado a un término y contendrá información sobre la frecuencia y posición de ese término en un documento indexado. Esta clase deberá tener al menos los siguientes métodos:
  - **String getDocumentId()**, que devolverá el identificador de un documento donde aparece el término asociado al *posting*.
  - **int getTermFrequency()**, que devolverá el número de ocurrencias del término en el documento asociado al *posting*.
  - **List<Long> getTermPositions()**, que devolverá las posiciones del término en el documento del *posting*.
- **Interfaz `es.uam.eps.bmi.search.indexing.Index`**, que deberá cumplirse por los diferentes índices a implementar. Esta interfaz deberá tener al menos los siguientes métodos:
  - **void build(String inputCollectionPath, String outputIndexPath, TextParser textParser)**, que construirá un índice a partir de una colección de documentos de texto plano. Recibirá como argumentos de entrada las ruta de la carpeta en la que se encuentran los documentos a indexar, y la ruta de la carpeta en la que almacenar el índice creado, así como un *parser* de texto que procesará el texto de los documentos para su indexación.
  - **void load(String indexPath)**, que cargará en RAM (parcial o completamente) un índice creado previamente, y que se encuentra almacenado en la carpeta cuya ruta se pasa como argumento de entrada.
  - **List<String> getDocumentIds()**, que devuelve los identificadores de los documentos indexados.
  - **TextDocument getDocument(String documentId)**, que devuelve el documento del identificador dado.
  - **List<String> getTerms()**, que devuelve la lista de términos extraídos de los documentos indexados.
  - **List<Posting> getTermPostings (String term)**, que devuelve los *postings* de un término dado.
  - **List<Posting> getDocumentPostings (String documentId)**, que devolverá la lista de posting indexados de un documento dado.
- **Interfaz `es.uam.eps.bmi.search.searching.Searcher`**, que deberá cumplirse por los diferentes buscadores a implementar. Para esta práctica se asume que un buscador accede a un único índice. Si se quisiera acceder a varios índices de forma simultánea se podría por ejemplo implementar un meta-buscador que combinase los resultados de búsqueda de varios buscadores simples. La interfaz deberá tener al menos los siguientes métodos:
  - **void build(Index index)**, que crea el buscador a partir del índice pasado como argumento de entrada.
  - **List<ScoredTextDocument> search(String query)**, que devolverá un *ranking* (ordenado por score decreciente) de documentos, resultantes de ejecutar una consultada dada sobre el índice del buscador.

A la hora de indexar un documento de texto, su contenido debe ser procesado. Por ejemplo, de un documento HTML hay que identificar aquellas partes que tienen contenido textual, filtrando otras como scripts de código, imágenes, vídeos, e información de formato y estilo; y del texto identificado hay que eliminar las etiquetas HTML que aparezcan en él, como por ejemplo <p>,
 y <ul>. Texto plano a su vez puede ser procesado para su posterior indexación, con el fin de eliminar signos de puntuación y *stopwords*, hacer stemming de los términos, etc.

En las prácticas de la asignatura habrá que implementar y usar diferentes *parsers* de texto. El diseño y complejidad de los mismos será libre. Sin embargo, para poder evaluarlos de forma común, se exige que soporten la siguiente interfaz:

- **Interfaz `es.uam.eps.bmi.search.parsing.TextParser`**, que deberá cumplirse por toda clase que procese un texto. Deberá tener al menos el siguiente método:
    - **String parse(String text)**, que devolverá procesado un texto de entrada.

Ejercicio 2: Motor de búsqueda con Apache Lucene [6,5 puntos]

Lucene es una API de código abierto para recuperación de información, originalmente desarrollada en Java, que es ampliamente utilizada en la implementación de motores de búsqueda.

Proporciona funcionalidades de indexación (incluyendo funciones de procesado de texto como *stemming* y filtrado de *stopwords*) y búsqueda de documentos de texto.

Aunque actualmente la versión estable de Lucene más alta es la 4.1, en esta práctica se va a usar la 3.6.2, con el fin de poder emplear el *toolbox* Luke (ver abajo).

La librería completa de Lucene se puede descargar de **http://lucene.apache.org/core/downloads.html**. En esta práctica será suficiente con usar los siguientes .jar, proporcionados junto con este enunciado:

- **lucene-core-3.6.2.jar**
- **lucene-analyzers-3.6.2.jar**
- **lucene-queryparser-3.6.2.jar**

En la Web existe amplia documentación sobre Lucene. La documentación oficial sobre la versión 3.6.2 está disponible en **http://lucene.apache.org/core/3_6_2**.

En este ejercicio se pide implementar en Java un indexador y un buscador sobre Lucene. En concreto, se pide implementar dos clases **`LuceneIndexer`** y **`LuceneSearcher`**. Estas clases serán *wrappers* de Lucene (es decir usarán clases y métodos Lucene internamente), y deberán satisfacer respectivamente las interfaces **`es.uam.eps.bmi.search.index.Index`** y **`es.uam.eps.bmi.search.searcher.Searcher`**.

Para facilitar la comprensión y uso de Lucene se proporcionan dos clases, **`IndexFiles`** y **`SearchFiles`**, accesibles públicamente en la Web, que usan Lucene para crear un índice y realizar búsquedas sobre un índice creado previamente.

Con el mismo fin se sugiere usar el *toolbox* Luke, **http://code.google.com/p/luke**, para visualizar y hacer búsquedas sobre un índice creado con Lucene. Luke es una herramienta que mediante una interfaz gráfica muy intuitiva permite visualizar, analizar y hacer búsquedas sobre índices Lucene. En las búsquedas, entre otros aspectos, se puede seleccionar el analizador morfológico y la similitud de consultas y documentos.

La solución a este ejercicio constará de:

- **Clase `es.uam.eps.bmi.search.index.LuceneIndex` (*implements* `Index`)** [4 puntos]. Además de los métodos definidos por la interfaz **`Index`** y aquellos se hayan decido incluir, la clase deberá tener un método *main* que reciba dos argumentos de entrada: la ruta de la carpeta que contiene la colección de documentos con los que crear el índice, y la ruta de la carpeta en la que almacenar el índice creado.
- **Clase `es.uam.eps.bmi.search.searcher.LuceneSearcher` (*implements* `Searcher`)** [2,5 puntos]. Además de los métodos definidos por la interfaz **`Searcher`** y aquellos se hayan decido incluir, la clase deberá tener un método *main* que reciba como argumento de entrada la ruta de la carpeta que contenga un índice Lucene, y que de forma iterativa pida al usuario consultas a ejecutar por el buscador sobre el índice y muestre por pantalla los top 5 documentos devueltos por el buscador para cada consulta.

La configuración del indexador y buscador se deja a libre elección: hacer o no filtrado de *stopwords* y *stemming*, modelo de recuperación de información a ejecutar, etc.

Sin embargo, como se indicará en el ejercicio 3, deberán probarse con un *parser* de documentos HTML, que puede utilizar librerías Java externas, como jsoup (http://jsoup.org).

Ejercicio 3: Estadísticas de indexación [1 punto]

A fin de inspeccionar el índice creado, en este ejercicio se pide implementar una clase Java **`TestIndex`** que use `LuceneIndex` para construir el índice de una colección de documentos dada y, una vez creado, acceda a él para escribir en fichero estadísticas de frecuencias globales de términos en la colección.

En concreto, para cada término `TextIndex` deberá sacar 1) el número de veces que es usado en la colección, y 2) el número de documentos en los que aparece. Estas estadísticas deberán visualizarse en diagramas en los que los términos figuren en orden decreciente de frecuencia.

El método de construcción del `LuceneIndex` recibirá un *parser* de documentos HTML (`HTMLSimpleParser` *implements* `TextParser`) que habrá que implementar, que en esta práctica no se evaluará, y que puede ser tan sencillo como encapsular la llamada al método estático `parse` de la clase `Jsoup` (http://jsoup.org).

La solución al ejercicio será:

- Clase **`es.uam.eps.bmi.search.TestIndex`**.
- Clase **`es.uam.eps.bmi.search.parsing.HTMLSimpleParser`**.
- Documento **bmi1314_XXXX_p2_YY_frecuencias.pdf** con gráficas que representen frecuencias (eje Y) contra términos (eje X), para las frecuencias de los seis ficheros anteriores, y con los términos ordenados en orden decreciente de frecuencia.

Ejercicio 4: Evaluación de resultados de búsqueda [1 punto]

En este ejercicio se pide implementar una clase Java **`TestSearcher`** que use `LuceneSearcher` para realizar consultas y evaluar resultados de búsqueda sobre las colecciones de documentos clueweb-1K, clueweb-10K y clueweb-100K.

En concreto, habrá que calcular y reportar los valores de P@5 y P@10 obtenidos para todas las consultas de cada colección.

La solución al ejercicio será:

- Clase **`es.uam.eps.bmi.search.TestSearcher`**.
- Clase **bmi1314_XXXX_p2_YY_precisiones.pdf** con tablas que reporten los valores de P@5 y P@10 para las consultas de cada colección.

Calificación

Esta práctica se calificará con una puntuación de 0 a 10 atendiendo a las puntuaciones individuales de ejercicios y apartados dadas en el enunciado. El peso de la calificación de la práctica en la calificación final de prácticas es del **20%**.

Entrega

La entrega de esta práctica consistirá en un fichero ZIP con el nombre **bmi1314_XXXX_p2_YY.zip**, donde XXXX debe sustituirse por el grupo 2461 ó 2462 según corresponda, e YY debe sustituirse por el número de pareja (01, 02, …, 10, …). Este fichero contendrá:

- Una carpeta **src** con todos los paquetes y ficheros fuente .java desarrollados.
- Una carpeta **doc** con los documentos HTML generados mediante la herramienta javadoc. Nótese que las clases y métodos implementados deberán ir debidamente documentados con sus correspondientes cabeceras.
- Los documentos .txt y .pdf con los resultados de los ejercicios 3 y 4.

Dicho fichero se enviará por el enlace habilitado al efecto en el curso **Moodle** de la asignatura.