Práctica 1:
Desarrollo de un Sistema de Recuperación de Información con Lucene

Gestión de información en la web

Curso 2020-2021

Francisco José González García

fran98@correo.ugr.es

Índice

1 Introducción

En esta práctica se ha construido una aplicación (tres si contamos cada módulo por separado) que nos permite indexar una colección documental y recuperarlos a partir de consultas realizadas por los usuarios a través de la interfaz gráfica.

Como decía la aplicación se divide en tres módulos principales:

1. Un indexador, que recibe la colección a indexar y la ruta donde crear el índice y lleva a cabo las tareas de indexación utilizando las librerias de Lucene.
2. Un motor de búsqueda, que recibe como parámetros el texto a buscar (query) y la ruta donde encontrar el índice. Utilizando Lucene recupera aquellos ficheros que coinciden con la búsqueda.
3. Una aplicación de escritorio que hace como interfaz de usuario para el motor de búsqueda. Le presenta al usuario una barra de búsqueda donde poder realizar las consultas y devuelve en forma de enlace agrupados en una lista los elementos encontrados. La aplicación ha sido creada usando React y Electron.

2 Colección documental

La colección documental elegida es una colección extraída de la Wikipedia que recoge algunos documentos de los que podemos encontrar en su web en la versión inglesa. Son en total 6043 documentos en formato html.

3 Indexador

Tanto para el indexador como para el motor de búsqueda se ha utilizado Scala[1] como lenguaje de programación. A su vez, y como se comentaba al principio, se ha utilizado Lucene[2] como API para implementar el indexador y el motor de búsqueda. Lucene es una potente librería ligera y rápida para realizar búsquedas. Construida en un primer momento en Java, tiene ports a multitud de lenguajes (python, C, C++...), aunque como vamos a utilizar scala utilizaremos las mismas librerías de Java (ya que scala se ejecuta sobre una JVM y tiene total compatibilidad con las bibliotecas de Java).

Utilizar Lucene es bastante sencillo y con un par de lineas tienes ya una aplicación funcionando. Ejemplo de ello es el código del indexador que en varias lineas queda montado:

```
object indexer extends App {
val usage = "scala indexer-0.1.jar -index INDEX_PATH -docs
DOCS_PATH"
var indexPath = "index";
var docsPath = "";
args.foreach(arg ⇒ {
if (arg.equals("-index")) indexPath = args(args.indexOf(arg)
+ 1)
if (arg.equals("-docs")) docsPath = args(args.indexOf(arg) +
1)
})
if (docsPath.isEmpty) {
println(s"Usage: $usage")
System.exit(1)
}
val docDir = Paths.get(docsPath)
val dir: Directory = FSDirectory.open(Paths.get(indexPath))
val analyzer: Analyzer = new EnglishAnalyzer()
val iwc: IndexWriterConfig = new IndexWriterConfig(analyzer)
iwc.setOpenMode(OpenMode.CREATE)

val writer: IndexWriter = new IndexWriter(dir, iwc)


indexDocs(writer, docDir)
```

En primer lugar parseamos los argumentos del programa, en este caso el path donde almacenar el índice (-index) y el path donde se encuentra la colección documental a indexar (-docs). Hecho esto, creamos un objeto Directory a partir del path recibido para almacenar el índice. Luego creamos, el que sea quizás, el objeto más importante del código, el *Analyzer.* Este objeto se encarga de todas las labores de procesado del texto, como son: Tokenización, Eliminación de palabras vacías (Stopwords), stemming o selección de terminos. Lucene incorpora multitud de Analyzers ya definidos para multitud de idiomas y en específico para el inglés. Todos ellos los podemos encontrar en el páquete *org.apache.lucene.luecene-analyzers-common.* En este caso se ha utilizado org.apache.lucene.analysis.en.EnglishAnalyzer que es el que incorpora Lucene por defecto para el inglés y que realiza las siguientes funciones:

- Eliminación de los posesivos en inglés: ‘s
- Normalizado de tokens a minúsculas
- StopFilter: eliminado de palabras vacías del inglés
- Marcado de palabras clave para que posteriormente no sean procesadas en el proceso de stemming
- PorterStemFilter: proceso de stemming utilizando el algoritmo de Porter

Luego, creamos un IndexWriter que será el encargado de construir el índice y que recibe como parámetros el Analyzer definido previamente y el directorio donde escribir el índice.

indexDocs es una función auxiliar que recorre el árbol de directorios de la colección documental y recupera todos los documentos almacenados. A su vez, cada uno de estos documentos son procesados por una función *indexDoc* que crea los Documentos correctamente procesados para que sean consumidos por el indexWriter.

```
def indexDoc(writer: IndexWriter, file: Path, lastModified:
Long): Unit = {
val source = Source.fromFile(file.toFile)
val rawDoc = source.getLines().mkString
val regex = "<[^>]*>".r
val cleanText = regex.replaceAllIn(rawDoc, "")
val doc: Document = new Document

val pathField: Field = new StringField("path",
file.toString, Field.Store.YES)
doc.add(pathField)
doc.add(new LongPoint("modified", lastModified))
doc.add(new TextField("contents", cleanText,
Field.Store.NO))

// println("indexing file " + file)
writer.addDocument(doc)
source.close
}
```

Lo que se hace básicamente es preprocesar el texto antes de pasarlo a Lucene quitando todas las etiquetas html del documento utilizando la expresión

regular `"<[^>]*>"`. Luego creamos un par de campos más como la ruta del documento y la fecha de última modificación y lo agregamos al *index.*

4 Motor de búsqueda

El motor de búsqueda es un script muy sencillo que recibe como parámetros la búsqueda a realizar y el índice donde buscar. Posteriormente definimos todos los elementos necesarios para realizar la búsqueda como son la lectura de índices del directorio de índices, el Analyzer a utilizar, un QueryParser() que analiza y procesa nuestra consulta con el Analyzer previamente definido y el propio buscador que recibe la salida ya procesada del QueryParser para realizar la búsqueda. Una factor muy importante a tener en cuenta es que el Analyzer debe ser exactamente el mismo utilizado en el proceso de indexación, ya que de otra forma la búsqueda no sería satisfactoria (o al menos completa al 100%) porque al texto de la búsqueda no se le habría aplicado el mismo procesamiento que al texto indexado.

```
...
val reader =
DirectoryReader.open(FSDirectory.open(Paths.get(indexDir)))
val searcher = new IndexSearcher(reader)
val analyzer = new EnglishAnalyzer()
val parser = new QueryParser(field, analyzer)
val query = parser.parse(queryString)
println(s"Buscando = ${query.toString(field)}")
val search = searcher.search(query, 100)
val results = search.scoreDocs
println(s"results.size = ${results.size}")
results.foreach(hit ⇒ {
val hitDoc = searcher.doc(hit.doc)
println(s"Doc: ${hitDoc.get("path")}")
})
println(s"Total hits: ${search.totalHits.value}")
reader.close()
...
```

Este es el trozo de código que define todos los elementos necesarios para realizar la búsqueda y hace la búsqueda propiamente dicha. En este caso devuelve como máximo un total de 100 elementos aunque se encuentren más (esto es pensando en la aplicación posterior). Además, como vemos se devuelve el path que habíamos almacenado previamente como un campo más.

5 Interfaz gráfica

Para hacer uso del motor de búsqueda que acabamos de crear de una forma amigable para el usuario se ha creado un pequeño programa de escritorio que hace de interfaz gráfica para poder realizar consultas al índice y obtener resultados que podrán ser seleccionados para abrir el/los documentos correspondientes. Esta aplicación se ha creado utilizando Electron[3] y React[4]. Electron es un framework de código abierto que permite crear aplicaciones de escritorio multiplataforma utilizando tecnologías web. Por otro lado React es una biblioteca, creada por Facebook, diseñada para el desarrollo de interfaces de usuario.

Como esta parte no está tan relacionada con el objetivo de la práctica no voy a comentar paso a paso el código ni como se ha construido la aplicación. Para más información se puede encontrar todo el código y documentación en este repositorio público de Github: https://github.com/Neo-Stark/GIW

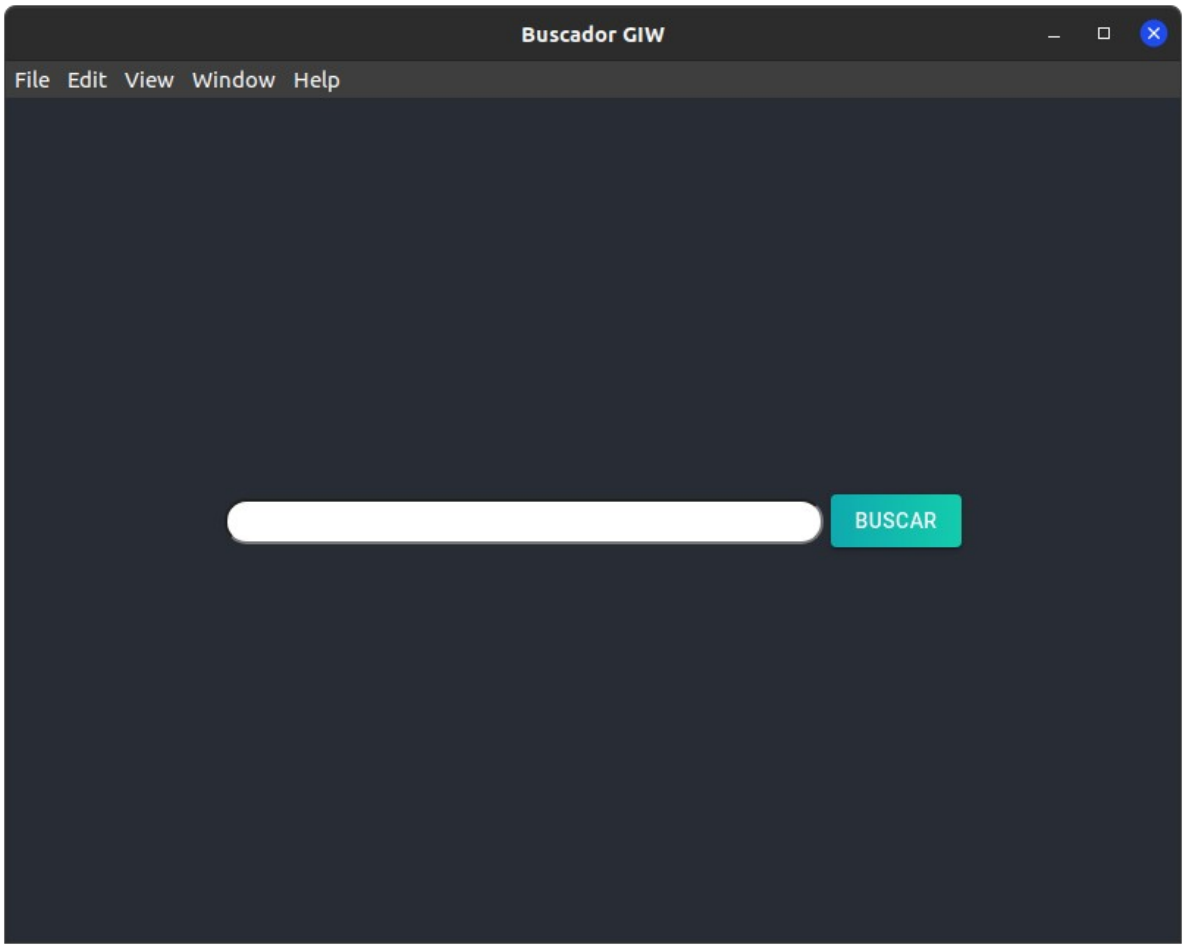

Figura 1: Vista principal del buscador

Cuando el usuario abre la aplicación se le muestra la ventana que se ve en la Figura 1. donde puede introducir su búsqueda y el sistema le devolverá los resultados. En caso de no encontrar ninguna coincidencia se le muestra un mensaje como el de la Figura 2.

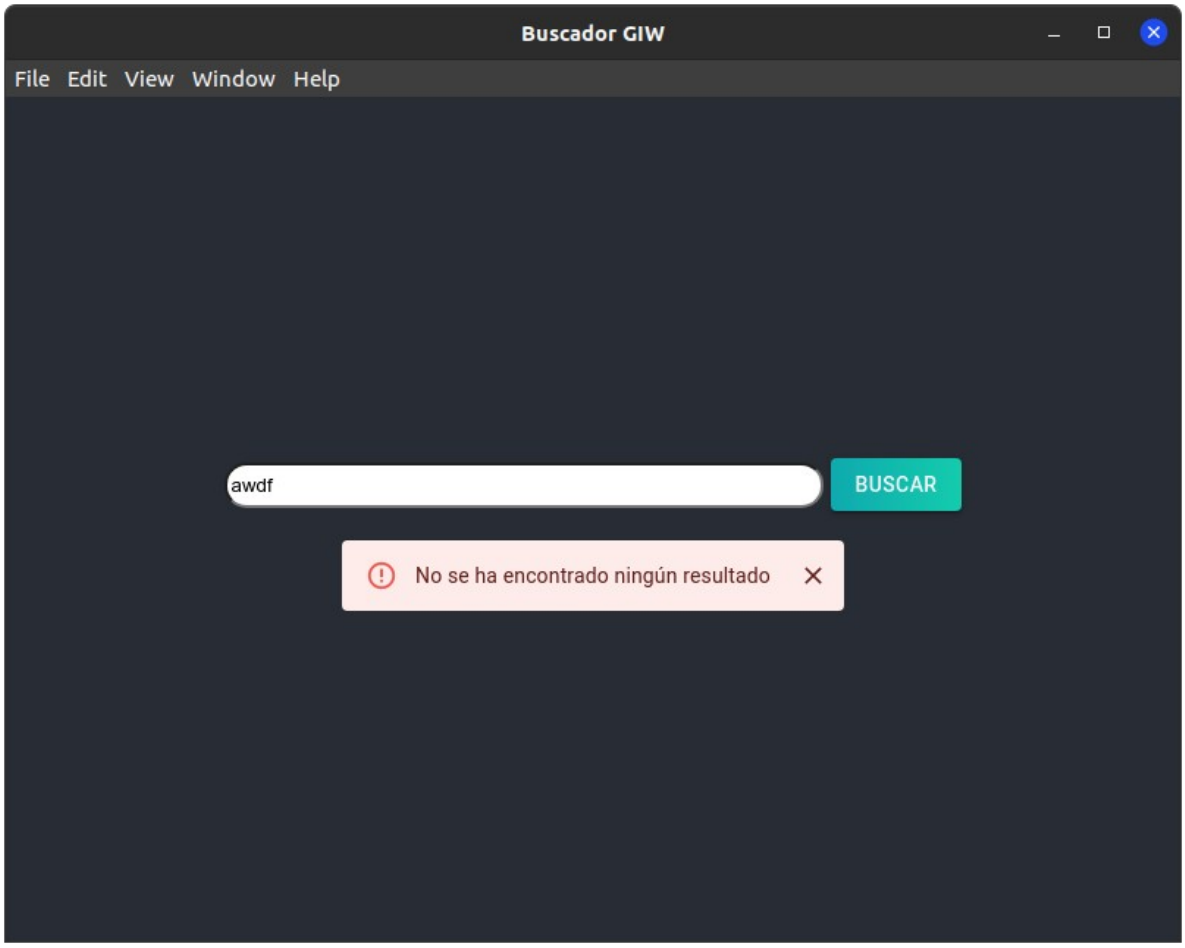

Figura 2: Mensaje de error cuando no se encuentra ningún resultado

Los resultados se muestran en forma de lista ordenados de mayor a menor peso o relevancia, es decir, aquellos documentos más relevantes se mostrarán en primer lugar.

Por ejemplo, si buscamos “Real Madrid” se nos muestran los resultados que vemos en la Figura 3 (Recordar que no están todos los documentos de la Wikipedia, solo una pequeña parte de ellos). En primer lugar aparece Sergio

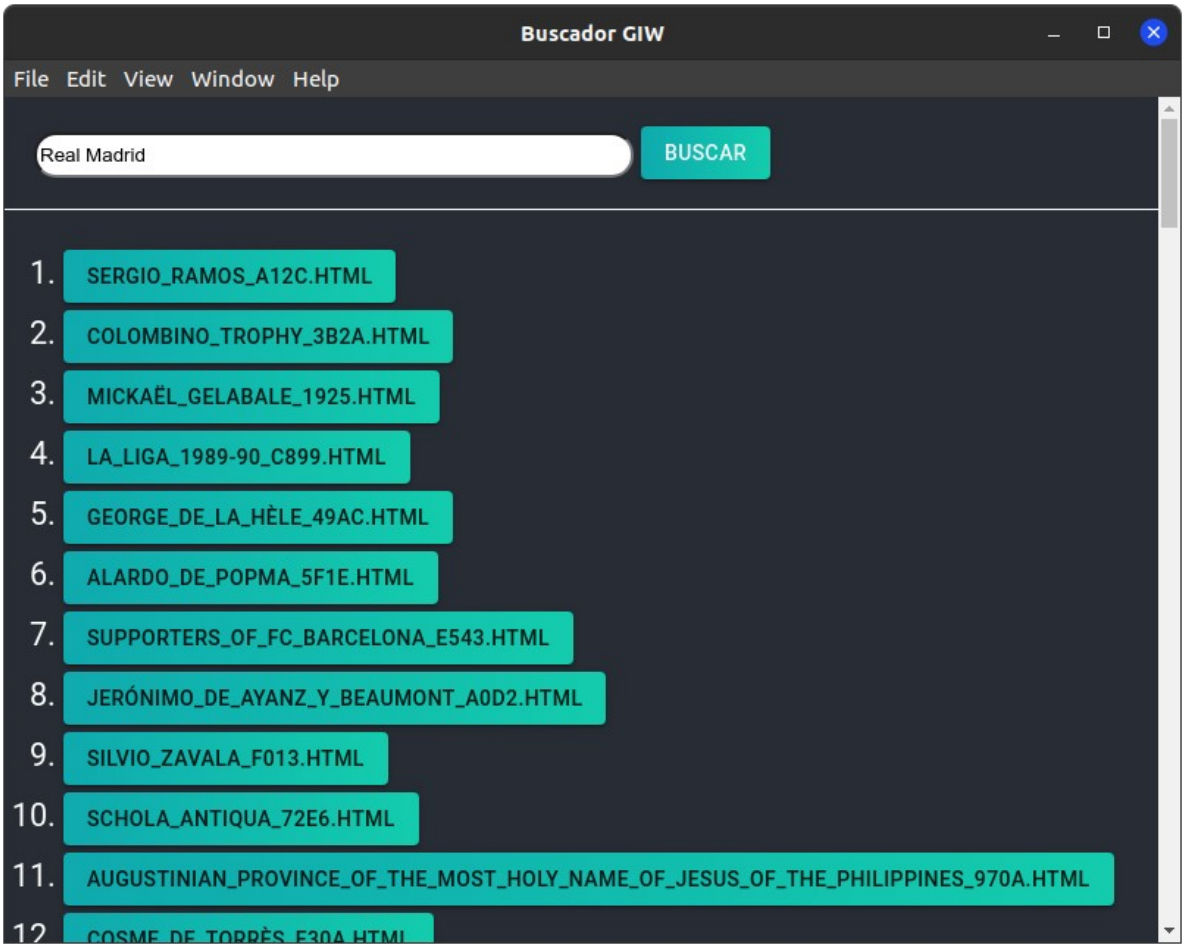

Figura 3: Resultados de la búsqueda "Real Madrid"

Ramos y si analizamos el texto podemos comprobar que el término Real Madrid aparece 23 veces. Luego aparece el Trofeo Colombino que el Real Madrid ha ganado 3 veces. Como tiene menos ocurrencias que en el documento de Sergio Ramos aparece en segunda posición. El resto de documentos siguen la misma ordenación.

Por último, comentar que haciendo click en cada uno de los resultados se nos abrirá el documento en cuestión con la aplicación/visor que tengamos configurado en nuestro sistema. En este caso al ser ficheros html se abrirán con el navegador por defecto que tengamos.

6 Estructura del proyecto

Tabla 1: Estructura del proyecto

```
.
├── articles
├── buscador
├── index
├── indexer-0.1.jar
├── motorBusqueda
├── scala-2.13.5
└── searcher-0.1.jar
```

El proyecto está dividido en los directorios que vemos en la Tabla 1. Cada uno de ellos contiene los siguentes elementos:

- **articles**: colección documental.
- **buscador**: aplicación de escritorio. Moviendonos a este directorio como se explicará en el siguiente apartado se podrá ejecutar la aplicación.
- **motorBusqueda**: contiene tanto el código desarrollado para el indexador como para el motor de busqueda. Es un proyecto de sbt con dos subproyectos: indexer y searcher
- **index**: índice generado por el indexador. (Por defecto se genera en la raíz del proyecto).
- **searcher-0.1.jar** y **indexer-0.1.jar** : paquetes jar que se pueden ejecutar directamente usando el interprete de scala y que corresponden a la versión en línea de comandos del buscador y el indexador.

7 Como ejecutar la aplicación

En primer lugar debemos descargar todo el código necesario desde el repositorio de GitHub: https://github.com/Neo-Stark/GIW

Si tenemos git instalado podemos hacerlo directamente desde la línea de comandos usando:

```
$git clone https://github.com/Neo-Stark/GIW
```

Una vez clonado y situados en la raíz del proyecto nos movemos al directorio *buscador:*

$cd buscador

Instalamos las dependencias del proyecto (es necesario tener instalado npm[4]):

$*npm install*

Ejecutamos la aplicación, usando nuevamente npm:

$*npm run start-app*

Para crear o actualizar el índice nos situamos nuevamente en la raíz del proyecto y ejecutamos:

$scala-2.13.5/bin/scala indexer-0.1.jar -index index -docs articles

Para ejecutar el buscador desde la línea de comandos:

$scala-2.13.5/bin/scala searcher-0.1.jar -index index -search [busqueda]

*Nota para la entrega:

Todo el código y la colección se encuentra alojado en el respositorio:

https://github.com/Neo-Stark/GIW

Bibliografía

[1] https://www.scala-lang.org/

[2] https://lucene.apache.org/core/8_8_1/

[3] https://www.electronjs.org/

[4] https://es.reactjs.org/docs/getting-started.html

[5] https://nodejs.org/api/