Proyecto RI

Recuperación de la Información

Cándido Alonso Barro
Andrés Picazo Mesa

CURSO 2024/25

Índice general

Capítulo 1

Crawler

Este código define un crawler llamado `CorpusCrawler` que utiliza la librería `Jsoup` para descargar y analizar enlaces desde una página HTML alojada en un repositorio remoto. Los archivos encontrados en la página se descargan y organizan en el sistema local manteniendo su estructura relativa.

Constantes principales

- `BASE_URL`: URL base del repositorio remoto donde se encuentran los archivos.
- `START_URL`: URL inicial que actúa como punto de entrada para el análisis.
- `OUTPUT_DIRECTORY`: Directorio local donde se almacenarán los archivos descargados.

Metodo principal

Aquí podemos observar lo importante del crawler, es cuando descarga y analiza la página inicial. Primero se conecta a la URL proporcionada anteriormente y descarga su contenido, extrae todos los elementos <a> que contengan `href` y ya dentro del bucle descarga cada archivo con su `URL` adecuada.

Listing 1.1: Estructuras de datos

```
System.out.println("Descargando␣indice...");
Document document = Jsoup.connect(START_URL).get();
```

```
Elements links = document.select("a[href]");

for (Element link : links) {
    String relativePath = link.attr("href");
    String fileUrl = BASE_URL + relativePath;
    downloadFile(fileUrl, relativePath);
}
```

1.1. Descarga documentos

La función downloadFile descarga un archivo desde una `URL` proporcionada y lo almacena localmente. Para lograrlo, establece una conexión con la *URL* del archivo y obtiene el flujo de entrada desde el cual leer los datos. Antes de guardar el archivo, verifica y crea los subdirectorios que hagan falta según la ruta. Luego, escribe los datos en un archivo local usando un buffer. Si la descarga y el almacenamiento salen bien, muestra un mensaje en la consola diciendo que el archivo se ha descargado correctamente. En caso de que ocurra un error muestra un mensaje de error.

Capítulo 2

Indexación

2.1. Preprocesado del texto

El preprocesamiento del texto se realiza a través de una clase dedicada (Preprocesado.java), diseñada para mejorar la modularidad y legibilidad del código. Esta clase recibe un objeto de tipo `String` y, mediante la función `procesar(String)`, aplica una serie de operaciones para normalizar el texto. Estas operaciones incluyen la conversión a minúsculas, la eliminación de signos de puntuación, números, espacios innecesarios, guiones, barras invertidas y palabras vacías (stopwords).

La lista de palabras vacías se carga desde un archivo externo (`stopwords-en.txt`) utilizando una estructura `HashSet` para garantizar una búsqueda eficiente.

La clase Preprocesado.java se puede ver en: Anexo-6.1

2.2. Creación del índice invertido y cálculo de longitud

Para almacenar el índice invertido en memoria durante el proceso de indexación, se ha utilizado un `Hashmap` en el que las claves corresponden a cada uno de los términos lematizados del corpus. Cada clave está asociada a un valor que consiste en una `Tupla`, implementada en `Tupla.java`. El primer elemento de la tupla representa el IDF del término, mientras que el segundo es otro `Hashmap` que contiene los nombres de los documentos en los que aparece dicho término (como clave) y el peso correspondiente de este término en cada documento (como valor).

Esta estructura de datos permite una mayor eficiencia tanto en las operaciones de

búsqueda como en las de inserción, a cambio de un mayor uso de memoria durante la ejecución.

La longitud de cada documento se almacena en otro `Hashmap`, esta vez más simple, donde la clave son los nombres de los documentos y su valor asociado su longitud.

Listing 2.1: Estructuras de datos

```
1 private static Map<String, Tupla<Double, Map<String, Double
     >>> indice_invertido = new HashMap<String, Tupla<Double,
     Map<String, Double>>>();
2 private static Map<String,Double> longitud = new HashMap<>();
```

2.2.1 Cálculo de TF

Como se indica en el documento de ayuda del proyecto, el cálculo del TF-IDF se realiza en dos pasos.

Primero se calcula el tf de cada término en cada documento aprovechando el bucle que lee todos los ficheros del corpus (6.2). Para obtener la frecuencia de cada término dentro de un documento se emplea la función `dividir_en_terminos()` (6.3) que emplea una estructura `Hashmap<String, Integer>` auxiliar, que se vaciará antes de pasar al siguiente documento.

Por último, con la frecuencia obtenida se calcula el TF (6.4) y se empieza a rellenar la estructura que empleará el índice invertido.

2.2.2 Cálculo de IDF y longitud

En este paso se calcula el IDF de cada término, la longitud de cada documento y el peso de cada término en cada documento. Se recorre cada término almacenado en el `Hashmap` y se obtienen los valores necesarios para el cálculo del IDF, número de documentos del corpus (obtenido al recorrer los documentos al principio) y el número de documentos en los que aparece dicho término (empleo la función size() del `Hashmap` que devuelve el número de claves).

Ahora se calcula el peso de término y se realizan los cálculos parciales para la longitud. Por último, se aplica la raíz cuadrada a cada valor de la longitud. La función encargada de esto se puede ver en: Anexo-6.6.

Capítulo 3

Búsqueda

3.1. La interfaz

El programa comienza con una interfaz basada en texto que permite al usuario ingresar consultas de búsqueda. La interfaz guía al usuario mediante mensajes iniciales y acepta consultas que pueden incluir operadores como `AND` u `OR`. También cuenta con un comando para salir del programa escribiendo "salir". A continuación, se presenta un fragmento del código relevante:

```
System.out.print("> ");
String query = scanner.nextLine().trim();

// Salir si el usuario escribe "salir"
if (query.equalsIgnoreCase("salir")) {
    System.out.println("Saliendo del buscador.");
    break;
}
// Procesar la consulta
query = preprocesado.procesar(query);

// Realizar la b squeda y ranking
Map<String, Double> rankResultado = rankDocumentos(query);

// Mostrar los resultados
monstrarResultados(rankResultado);
}
scanner.close();
```

3.2. Ranking de documentos

El ranking de documentos se realiza a partir de una consulta procesada. Se calculan puntuaciones basadas en la frecuencia del término (TF) y el factor inverso de frecuencia de documentos (IDF). Los documentos se ordenan en función de su relevancia y se devuelven los mejores resultados. El cálculo sería tal que: $puntuacion = tf\,Documento * idf\,Documento$;

```
private static Map<String, Double> rankDocumentos(String
    consulta) {
     Stemmer stemmer = new Stemmer();
     String[] terminos = consulta.split("␣");
     Map<String, Double> puntuacionesDocumentos = new HashMap
        <>();

     for (String termino : terminos) {
          stemmer.add(termino.toCharArray(), termino.length());
          stemmer.stem();
          termino = stemmer.toString();

          List<DocumentoPeso> pesosDocumentos = indiceInvertido
             .getOrDefault(termino, Collections.emptyList());
          for (DocumentoPeso docPeso : pesosDocumentos) {
               double puntuacion = docPeso.tf * docPeso.idf;
               puntuacionesDocumentos.merge(docPeso.
                  nombreDocumento, puntuacion, Double::sum);
          }
     }

     return puntuacionesDocumentos.entrySet().stream()
               .sorted((a, b) -> Double.compare(b.getValue(), a.
                  getValue()))
               .limit(10)
               .collect(Collectors.toMap(
                       Map.Entry::getKey,
                       Map.Entry::getValue,
                       (e1, e2) -> e1,
                       LinkedHashMap::new
               ));
}
```

3.3. Cómo diferencia entre AND, OR o ninguno

La consulta ingresada se analiza para identificar si contiene los operadores AND o OR.En este caso se puede observar que los analizo para cuando están en minúsculas y es porque antes ya han recibido un trato de preprocesamiento. Dependiendo de esto, se procesan los términos de manera distinta:

- AND: Los documentos relevantes deben contener todos los términos especificados.
- OR: Los documentos relevantes pueden contener cualquiera de los términos especificados.
- **Sin operadores**: Se tomara como si fuera un termino unico.

El siguiente fragmento de código ilustra esta lógica:

```
boolean esConsultaAnd = consulta.contains("and");
boolean esConsultaOr = consulta.contains("or");
String[] terminos;

if (esConsultaAnd) {
    terminos = consulta.split("\sand\s");
}else if (esConsultaOr) {
    terminos = consulta.split("\sor\s");
} else {
    terminos = new String[]{consulta.trim()};
}
```

3.4. Cálculo de documentos relevantes AND

Cuando la consulta tiene el operador AND, se realiza una intersección de los documentos asociados a cada término. Así aseguramos que solo los documentos que contienen los términos sean considerados relevantes.

```
if (esConsultaAnd) {
    if (documentosRelevantes.isEmpty()) {
        documentosRelevantes = new HashSet<>(
            documentosParaTermino);
    } else {
        documentosRelevantes.retainAll(documentosParaTermino)
            ;
```

```
    }
}
```

3.5. Cálculo de puntuajes de documentos relevantes

Para cada término de la consulta, se calcula un puntaje que se basa en la fórmula. Los puntajes se suman y se van acumulando para cada documento relevante.

```
for (DocumentoPeso docPeso : pesosDocumentos) {
    if (documentosRelevantes.contains(docPeso.nombreDocumento
       )) {
        double puntuacion = docPeso.tf * docPeso.idf;
           puntuacionesDocumentos.merge(docPeso.
           nombreDocumento, puntuacion, Double::sum);
    }
}
```

3.6. Mostrar resultados

Los resultados se presentan ordenados por su puntuacion en orden descendente. Si no se encuentran documentos relevantes, se muestra.

```
private static void monstrarResultados(Map<String, Double>
   resultadosRankeados) {
    if (resultadosRankeados.isEmpty()) {
        System.out.println("No se encontraron documentos
           relevantes para la consulta.");
    } else {
        System.out.println("Documentos encontrados:");
           resultadosRankeados.forEach((doc, puntuacion) ->
           System.out.printf("- %s (Puntuacion: %.4f)%n", doc,
            puntuacion));
    }
}
```

Capítulo 4

Puntos extra

4.1. Stemming

Se ha aplicado el algoritmo de Porter para el stemming empleando una clase Java encontrada en GitHub. El repositorio se puede acceder mediante el siguiente enlace: `https://gist.github.com/ldclakmal/667d8ecb620a0cce7d3dedae80a2c013`.

El algoritmo de Porter funciona principalmente eliminando sufijos típicos del inglés, como pueden ser '-ing', '-ed', '-ly'. Aunque no siempre se pueda obtener la raíz exactamente correcta empleando el algoritmo, consideramos que es una solución óptima para nuestro sistema RI debido a su simplicidad y velocidad, aparte de ser un algoritmo muy probado y extendido.

4.2. Operadores AND y OR en la búsqueda

Como ya vimos anteriormente en la parte de búsqueda, el *AND* y el *OR* se obtienen viendo si dichas palabras están dentro de la consulta. En el caso de que estén, si es un *AND* vemos qué documentos comparten en común y son esos los que elegimos, y en el caso de que sea un *OR*, escogemos todos. Para el cálculo de la puntuación de los documentos que comparten ambos, se hace mediante la operación:

$$\text{puntuación} = tf_{Documento1} \cdot idf_{Documento1} + tf_{Documento2} \cdot idf_{Documento2}$$

Capítulo 5

Instalación y manual del usuario

5.1. Instalacion

Para ejecutar el proyecto, lo que necesitamos hacer es asegurarnos de tener instalado *Java* en nuestro ordenador. Puedes verificarlo ejecutando el siguiente comando en la terminal:

```
java --version
```

Si no te aparece la versión, tendrás que instalarte *Java*. Una vez que tengas Java instalado, descarga y descomprime la carpeta del proyecto que te hemos dado. Dentro de esta carpeta, encontrarás todos los archivos. Para compilar y ejecutar el proyecto, solo necesitas ejecutar el archivo ejecucion.bat, que está incluido en la carpeta. Este archivo es un script que automatiza todo el proceso de compilación y ejecución del proyecto.

El corpus también debe encontrarse en la carpeta raíz del proyecto. De lo contrario, será necesario modificar la variable `corpus_path` de Indexacion.java y sustituirla por la dirección deseada.

5.2. Manual del usuario

Al ejecutar el archivo ejecucionIndexacion.bat, se ejecutará tanto el crawler como la parte de indexación, y seguidamente tendremos que ejecutar el archivo ejecucionBuscador.bat, que ejecutará el buscador.

Cuando el programa esté en funcionamiento, se te pedirá que ingreses las palabras o términos que deseas buscar. Para hacerlo, solo tendrás que escribir la palabra clave y presionar Enter. El programa realizará la búsqueda en los datos procesados y te devolverá los resultados relevantes.

Si en algún momento deseas finalizar el programa, simplemente escribe la palabra ‘salir‘ y presiona Enter. El programa se cerrará de inmediato.

Capítulo 6

Anexo

A continuación se presentan algunos de los códigos desarrollados:

6.1. Preprocesado.java

Listing 6.1: Clase `Preprocesado.java`

```
public class preprocesado {

    private static Set<String> STOPWORDS = new HashSet<>();

    static {
        try {
            STOPWORDS = new HashSet<>(Files.readAllLines(
               Paths.get("./utility/stopwords-en.txt")));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static String procesar(String cad) {
        cad = minusculas(cad);
        cad = eliminar_signos(cad);
        cad = eliminar_barra_invertida(cad);
        cad = eliminar_numeros(cad);
        cad = eliminar_guiones(cad);
        cad = eliminar_espacios(cad);
        cad = eliminar_stopwords(cad);
```

```
21
22          return cad;
23      }
24
25      private static String eliminar_signos(String cad) {
26          Pattern pat = Pattern.compile("[!\"#$%&'()
                *+,./:;<=>?\\@\\[\\]^_`{|}~]");
27          Matcher mat = pat.matcher(cad);
28          return mat.replaceAll("␣");
29      }
30
31      private static String eliminar_barra_invertida(String cad
            ) {
32          return cad.replace("\\", "␣");
33      }
34
35      private static String eliminar_guiones(String cad) {
36          return cad.replaceAll("\\s-+\\s", "␣");
37      }
38
39      private static String eliminar_numeros(String cad) {
40          return cad.replaceAll("\\b\\d*\\b", "␣");
41      }
42
43      private static String minusculas(String cad) {
44          return cad.toLowerCase();
45      }
46
47      private static String eliminar_espacios(String cad) {
48          return cad.replaceAll("\\s+", "␣");
49      }
50
51      private static String eliminar_stopwords(String cad) {
52          ArrayList<String> aux = Stream.of(cad.split("\\s"))
53                  .collect(Collectors.toCollection(ArrayList<
                        String>::new));
54          aux.removeAll(STOPWORDS);
55          return aux.stream().collect(Collectors.joining("␣"));
56      }
57  }
```

6.2. Indexacion.java

Listing 6.2: Bucle que lee los documentos

```
File dir = new File(corpus_path);
    if (dir.exists() && dir.isDirectory()) {
        File[] documentos = dir.listFiles();
        if (documentos != null) {
            for (File documento : documentos) {
                try {
                    String contenido = new String(Files.
                       readAllBytes(Paths.get(documento.
                       getPath())));
                    contenido = preprocesado.procesar(
                       contenido);
                    dividir_en_terminos(contenido);
                    calcular_tf(documento.getName());
                    terminos_map.clear();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                    // Aumento el numero de documentos
                    N++;
                }
            }
        }
     }
```

Listing 6.3: Division del texto en términos y conteo de la frecuencia de estos

```
private static void dividir_en_terminos(String texto) {
    // Divido el texto en palabras
    String[] terminos = texto.split("\\s+");
    // Sin esto se crea siempre al principio un registro
       vacio
    ArrayList<String> listaTerminos = new ArrayList<>(Arrays.
       asList(terminos));
    listaTerminos.removeIf(String::isEmpty);
    terminos = listaTerminos.toArray(new String[0]);
    // Recorro todos los terminos
    for (String termino : terminos) {
        //Aplico el algoritmo de stemming
        Stemmer stemmer = new Stemmer();
        char[] termArray = termino.toCharArray();
```

```
13          stemmer.add(termArray, termArray.length);
14          stemmer.stem();
15          termino = stemmer.toString();
16          // Si no esta en el map
17          if (terminos_map.get(termino) == null)
18              // Inicializo el valor a 1
19              terminos_map.put(termino, 1);
20          // Si esta
21          else {
22              // Sumo 1 al valor
23              terminos_map.put(termino, terminos_map.get(
                    termino) + 1);
24          }
25      }
26  }
```

Listing 6.4: Cálculo del TF

```
1   private static void calcular_tf(String name) {
2       // Recorro todos los terminos del documento actual
3       for (Map.Entry<String, Integer> entry : terminos_map.
            entrySet()) {
4           // Obtengo el termino
5           String termino = entry.getKey();
6           // Obtengo su frecuencia
7           Integer frecuencia = entry.getValue();
8           // Calculo el tf
9           Double tf = 1 + Math.log(frecuencia) / Math.log(2);
10          // Si no esta el termino en el mapa tf-idf lo
                inicializo vacio
11          if (!indice_invertido.containsKey(termino)) {
12              Map<String, Double> mapaInterno = new HashMap<>()
                    ;
13              Tupla<Double, Map<String, Double>> nuevaTupla =
                    new Tupla<>(0.0, mapaInterno);
14              // Inserta la nueva tupla en el mapa con la clave
                     proporcionada
15              indice_invertido.put(termino, nuevaTupla);
16          }
17          // Recupero la tupla del termino actual
18          Tupla<Double, Map<String, Double>> tupla_actual =
                indice_invertido.get(termino);
19          // Recupero el map donde guardo el documento y su tf
20          Map<String, Double> map_actual = tupla_actual.second;
```

```
21          // Guardo el id y el tf
22          map_actual.put(name, tf);
23      }
24  }
```

Listing 6.5: Cálculo del IDF y longitud

```
1   private static void calcular_idf_y_longitud() {
2       for (Map.Entry<String, Tupla<Double, Map<String, Double
           >>> entry : indice_invertido.entrySet()) {
3           Tupla<Double, Map<String, Double>> tupla_actual =
               entry.getValue();
4           Integer n = tupla_actual.second.size();
5           Double a = (double) N / n;
6           Double idf = Math.log(a) / Math.log(2);
7           tupla_actual.first = idf;
8           for (Map.Entry<String, Double> doc : tupla_actual.
               second.entrySet()) {
9               String docName = doc.getKey();
10              Double peso = doc.getValue() * idf;
11              doc.setValue(peso);
12              if (longitud.get(docName) != null)
13                  longitud.put(docName, longitud.get(docName) +
                       peso * peso);
14              else
15                  longitud.put(docName, peso * peso);
16          }
17      }
18      for (Map.Entry<String, Double> doc : longitud.entrySet())
19          doc.setValue(Math.sqrt(doc.getValue()));
20  }
```

6.3. CorpusCrawler.java

Listing 6.6: Crawler

```
1   public class CorpusCrawler {
2       private static final String BASE_URL = "https://raw.
           githubusercontent.com/PdedP/RECINF-Project/refs/heads/
           main/";
3       private static final String START_URL = BASE_URL + "index
           .html"; // Pagina inicial
```

```
4       private static final String OUTPUT_DIRECTORY = ".";
5
6       public static void main(String[] args) {
7           try {
8               File directory = new File(OUTPUT_DIRECTORY);
9               if (!directory.exists()) {
10                  directory.mkdir();
11              }
12
13              // Descargar y analizar la pagina inicial
14              System.out.println("Descargando indice...");
15              Document document = Jsoup.connect(START_URL).get
                    ();
16              Elements links = document.select("a[href]");
17
18              for (Element link : links) {
19                  String relativePath = link.attr("href");
20                  String fileUrl = BASE_URL + relativePath;
21                  downloadFile(fileUrl, relativePath);
22              }
23
24              System.out.println("Descarga completada. Los
                    archivos estan en el directorio: " +
                    OUTPUT_DIRECTORY);
25          } catch (IOException e) {
26              System.err.println("Error al procesar el indice:
                    " + e.getMessage());
27          }
28      }
29
30      private static void downloadFile(String fileUrl, String
            relativePath) {
31          try (InputStream in = new URL(fileUrl).openStream())
                {
32              // Crea los subdirectorios
33              File outputFile = new File(OUTPUT_DIRECTORY,
                    relativePath);
34              outputFile.getParentFile().mkdirs();
35
36              try (OutputStream out = new FileOutputStream(
                    outputFile)) {
37                  byte[] buffer = new byte[1024];
38                  int bytesRead;
```

```
39                  while ((bytesRead = in.read(buffer)) != -1) {
40                      out.write(buffer, 0, bytesRead);
41                  }
42                  System.out.println("Archivo descargado: " +
                        relativePath);
43              }
44          } catch (IOException e) {
45              System.err.println("Error al descargar el archivo
                    : " + fileUrl + " - " + e.getMessage());
46          }
47      }
48  }
```

6.4. Buscador.java

Listing 6.7: Buscador de documentos

```
1   import utility.Stemmer;
2   import utility.preprocesado;
3
4   import java.io.*;
5   import java.util.*;
6   import java.util.stream.Collectors;
7
8   public class Buscador {
9
10      // Indice invertido construido desde el archivo
11      private static final Map<String, List<DocumentoPeso>>
            indiceInvertido = new HashMap<>();
12
13      public static void main(String[] args) {
14          // Construir el indice desde el archivo en la carpeta
                "utility"
15          try {
16              cargarIndexArchivo("utility/indice_invertido.dat"
                    );
17          } catch (IOException e) {
18              System.err.println("Error al cargar el indice
                    invertido: " + e.getMessage());
19              return; // Termina el programa si no se puede
                    cargar el archivo
```

```
20          }
21
22          Scanner scanner = new Scanner(System.in);
23          System.out.println("Bienvenido␣al␣buscador␣de␣
               documentos.");
24          System.out.println("Introduce␣una␣consulta␣(puedes␣
               usar␣operadores␣AND/OR):");
25
26          while (true) {
27              System.out.print(">␣");
28              String query = scanner.nextLine().trim();
29
30              // Salir si el usuario escribe "salir"
31              if (query.equalsIgnoreCase("salir")) {
32                  System.out.println("Saliendo␣del␣buscador.");
33                  break;
34              }
35
36              // Procesar la consulta
37              query = preprocesado.procesar(query);
38
39              // Realizar la busqueda y ranking
40              Map<String, Double> rankResultado =
                   rankDocumentos(query);
41
42              // Mostrar los resultados
43              monstrarResultados(rankResultado);
44          }
45
46          scanner.close();
47      }
48
49      private static void cargarIndexArchivo(String fileName)
           throws IOException {
50          try (BufferedReader reader = new BufferedReader(new
               FileReader(fileName))) {
51              String line;
52              while ((line = reader.readLine()) != null) {
53                  // Parsear cada linea del archivo
54                  String[] parts = line.split(";");
55                  if (parts.length < 3) continue; // Saltar
                       lineas mal formateadas
56
```

```
57                  String word = parts[0].toLowerCase(); //
                        Primera parte: palabra clave
58                  double idf = Double.parseDouble(parts[1]); //
                        Segunda parte: IDF
59                  List<DocumentoPeso> documents = new ArrayList
                        <>();
60
61                  // Procesar documentos-tf asociados
62                  for (int i = 2; i < parts.length; i++) {
63                      String[] docParts = parts[i].split("-");
64                      if (docParts.length > 1) {
65                          String nombreDocumento = docParts[0];
66                          double tf = Double.parseDouble(
                                docParts[1]);
67                          documents.add(new DocumentoPeso(
                                nombreDocumento, tf, idf));
68                      }
69                  }
70
71                  // Agregar al indice invertido
72                  indiceInvertido.put(word, documents);
73              }
74          }
75      }
76
77      private static Map<String, Double> rankDocumentos(String
            query) {
78          // Crear una instancia del stemmer
79          Stemmer stemmer = new Stemmer();
80
81          // Determinar el tipo de consulta
82          boolean esAndQuery = query.contains("and");
83          boolean esOrQuery = query.contains("or");
84
85          // Si la consulta tiene AND o OR, dividirla en
                terminos
86          String[] terminos;
87          if (esAndQuery) {
88              terminos = query.split("\\s*and\\s*");  // Se
                    maneja AND con espacios opcionales
89          } else if (esOrQuery) {
90              terminos = query.split("\\s*or\\s*");   // Se
                    maneja OR con espacios opcionales
```

```
91          } else {
92              // Si no contiene AND ni OR, tratamos toda la
                   consulta como un unico termino
93              terminos = new String[]{query.trim()};  // Se
                   toma como un unico termino
94          }
95
96          Set<String> documentoRelevante = new HashSet<>();
97          Map<String, Double> puntuacionDocumento = new HashMap
               <>(); // Para acumular los puntajes
98
99          for (String term : terminos) {
100             term = term.trim();  // Limpiar posibles espacios
                    extras
101
102             if (term.isEmpty()) continue;  // Asegurarse de
                   que no estamos procesando terminos vac os
103
104             // Aplicar stemming al termino
105             stemmer.add(term.toCharArray(), term.length());
106             stemmer.stem();
107             term = stemmer.toString();  // Obtener la raiz
                   del termino
108
109             List<DocumentoPeso> pesoDocumento =
                   indiceInvertido.getOrDefault(term, Collections.
                   emptyList());
110             Set<String> documentParaTermino = pesoDocumento.
                   stream()
111                     .map(dw -> dw.nombreDocumento)
112                     .collect(Collectors.toSet());
113
114             if (esAndQuery) {
115                 // Si es una consulta con AND, hacemos la
                       interseccion de documentos
116                 if (documentoRelevante.isEmpty()) {
117                     documentoRelevante = new HashSet<>(
                           documentParaTermino);
118                 } else {
119                     documentoRelevante.retainAll(
                           documentParaTermino);
120                 }
121             } else if (esOrQuery) {
```

```
122                 // Si es una consulta con OR, hacemos la
                        union de documentos
123                 documentoRelevante.addAll(documentParaTermino
                        );
124             } else {
125                 // Implicito OR si no hay operadores
126                 documentoRelevante.addAll(documentParaTermino
                        );
127             }
128         }
129
130         // Si no hay documentos relevantes, retornar un mapa
                vacio
131         if (documentoRelevante.isEmpty()) {
132             return Collections.emptyMap();
133         }
134
135         // Calcular puntajes para los documentos relevantes
136         for (String term : terminos) {
137             term = term.trim();  // Limpiar espacios
138
139             if (term.isEmpty()) continue;  // Saltar si el
                    termino esta vacio
140
141             // Aplicar stemming nuevamente (si es necesario)
                    antes de calcular puntajes
142             stemmer.add(term.toCharArray(), term.length());
143             stemmer.stem();
144             term = stemmer.toString();
145
146             List<DocumentoPeso> pesoDocumento =
                    indiceInvertido.getOrDefault(term, Collections.
                    emptyList());
147             for (DocumentoPeso docWeight : pesoDocumento) {
148                 if (documentoRelevante.contains(docWeight.
                        nombreDocumento)) {
149                     // Calcular el puntaje para cada
                            documento
150                     double score = docWeight.tf * docWeight.
                            idf;
151                     // Sumar puntaje al documento sin
                            duplicar
```

```
152                     puntuacionDocumento.merge(docWeight.
                            nombreDocumento, score, Double::sum);
153                 }
154             }
155         }
156
157         // Ordenar los documentos por puntaje en orden
                descendente
158         return puntuacionDocumento.entrySet()
159                 .stream()
160                 .sorted((a, b) -> Double.compare(b.getValue()
                        , a.getValue()))
161                 .limit(10) // Mostrar solo los 10 mejores
                        resultados
162                 .collect(Collectors.toMap(
163                         Map.Entry::getKey,
164                         Map.Entry::getValue,
165                         (e1, e2) -> e1,
166                         LinkedHashMap::new
167                 ));
168     }
169
170     private static double calculaIDF(String term) {
171         // Obtener los documentos donde aparece el termino
172         List<DocumentoPeso> pesoDocumento = indiceInvertido.
                getOrDefault(term, Collections.emptyList());
173         if (pesoDocumento.isEmpty()) return 0.0;
174
175         // Tomar el IDF del primer documento asociado
176         return pesoDocumento.get(0).idf;
177     }
178
179     private static void monstrarResultados(Map<String, Double
            > rankResultado) {
180         if (rankResultado.isEmpty()) {
181             System.out.println("No se encontraron documentos
                    relevantes para la consulta.");
182         } else {
183             System.out.println("Documentos encontrados:");
184             rankResultado.forEach((doc, score) -> System.out.
                    printf("- %s (Score: %.4f)%n", doc, score));
185         }
186     }
```

```
187
188     // Clase auxiliar para representar un documento y su peso
189     // Clase auxiliar para representar un documento, su TF y
            el IDF del termino
190     private static class DocumentoPeso {
191     String nombreDocumento;
192     double tf; // Frecuencia del termino en el documento
193     double idf; // IDF del termino
194
195         DocumentoPeso(String nombreDocumento, double tf,
                double idf) {
196             this.nombreDocumento = nombreDocumento;
197             this.tf = tf;
198             this.idf = idf;
199         }
200     }
201
202 }
```