Recuperación de Información (2019-2020)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Preprocesado de documentos

Simón López Vico
Miguel Cantarero López
Alberto Jesús Durán López

28 de octubre de 2019

1. **Sobre los documentos (libros del proyecto Gutenberg) utilizados en la práctica anterior, hacer un estudio estadístico sobre los distintos tokens que se obtienen al realizar distintos tipos de análisis ya predefinidos. Por tanto, será necesario contar el número de términos de indexación así como frecuencias de los mismos en cada documento. Realizar un análisis comparativo entre los distintos resultados obtenidos.**

   Para este ejercicio hemos desarrollado una función la cual, pasado un *analyzer* como parámetro, divide el texto en sus diferentes *tokens* y guarda la frecuencia de cada uno de ellos en un `map`. Finalmente, ordena las palabras obtenidas por el número de ocurrencias en el texto (de mayor a menor) y muestra las 20 primeras junto al número total de *tokens*.

```
public static void Analyzer(Analyzer analyzer, String name, String
    contenido) throws IOException{
  System.out.println(name);
  TokenStream stream = analyzer.tokenStream(null, contenido);
  CharTermAttribute attr=stream.addAttribute(CharTermAttribute.class);
  stream.reset();

  Map<String,Integer> palabra_ocurrencias = new HashMap<String,
      Integer>();

  while(stream.incrementToken()){
    String palabra=attr.toString();
    //System.out.println(palabra);
    Integer value=palabra_ocurrencias.get(palabra);
    if(value==null) palabra_ocurrencias.put(palabra,1);
    else            palabra_ocurrencias.put(palabra,value+1);
  }

  palabra_ocurrencias=sortByValue(palabra_ocurrencias);
  PrintPalabras();

  System.out.println("\nNumero de tokens en el fichero: " +
      palabra_ocurrencias.size() + "\n");
  stream.end();
  stream.close();
}
```

   Con esto, podemos llamar a cualquier analizador incluido en *Lucene* mediante el código `Analyzer(new WhitespaceAnalyzer(), "WhitespaceAnalyzer", contenido);`, donde `contenido` será el texto plano extraído con Tika del archivo parseado.

   Hablemos ahora sobre los analizadores utilizados y sus respectivas salidas[1]:

   - **WhitespaceAnalyzer**: un analizador muy simple, solo usa el *WhitespaceTokenizer*, el cuál separa el texto por los espacios en blanco. Como resultado obtenemos:

[1] Para hacer la comparación entre los distintos analizadores utilizados, nos centraremos en el fichero `odisea-es-txt.txt`.

```
*************** WhitespaceAnalyzer ***************
de - 8120 á - 6957 y - 6817 que - 5440 la - 5039 [...]
Número de tokens en el fichero: 25769
```

- **StandardAnalyzer**: utiliza el *StandardTokenizer*, además de pasar todos los tokens generados a minúscula y elimina las palabras que se encuentran en `ENGLISH_STOP_WORDS_SET`. Este analizador puede reconocer URLs. El resultado obtenido es el siguiente:

  ```
  *************** StandardAnalyzer ***************
  de - 8257 y - 7547 á - 7045 que - 5640 la - 5150 [...]
  Número de tokens en el fichero: 16194
  ```

  Como vemos, con este analizador encuentra muchos menos tokens que con el WhitespaceAnalyzer, lo que es normal, pues para WhitespaceAnalyzer "de" y "De" serían dos tokens diferentes, mientras que para StandardAnalyzer es el mismo token.

- **SimpleAnalyzer**: consiste en la aplicación de *LetterTokenizer* y *LowerCaseFilter*, es decir, define los tokens como cadenas máximas de letras adyacentes y luego pasa estos a minúscula. A diferencia de StandardAnalyzer, este analizador no puede reconocer URLs. Obtenemos lo siguiente:

  ```
  *************** SimpleAnalyzer ***************
  de - 8259 y - 7548 á - 7045 que - 5640 la - 5151 [...]
  Número de tokens en el fichero: 15419
  ```

- **StopAnalyzer**: hace lo mismo que SimpleAnalyzer, pero además elimina el conjunto de palabras que le pasemos (*StopWords*) mediante el filtro *StopFilter*. Hemos ejecutado usando un conjunto de palabras vacío y usando el conjunto de palabras inútiles en español, obteniendo los siguientes resultados:

  ```
  *************** StopAnalyzer EMPTY_WORDS_SET ***************
  de - 8259 y - 7548 á - 7045 que - 5640 la - 5151 [...]
  Número de tokens en el fichero: 15419
  *************** StopAnalyzer SPANISH_STOP_WORDS_SET ***************
  ulises - 1706 telémaco - 745 así - 547 pues - 525 si - 512 [...]
  Número de tokens en el fichero: 15171
  ```

  Para el caso del conjunto vacío obtenemos el mismo resultado que SimpleAnalyzer, lo que era de esperar; por otra parte, eliminando las palabras inútiles en español comenzamos a tener una información más relevante sobre el tema del que trata nuestro texto.

- **SpanishAnalyzer**: este analizador es el que más operaciones realiza sobre el texto y los tokens obtenidos de él. Tras pasar los filtros estándar, pasa el

SpanishPossessiveFilter, que eliminará los "restos" posesivos de las distintas palabras. Después pasará todo a minúscula y eliminará las *stopwords*, para finalmente utilizar el filtro *PorterStemFilter*. Obtenemos el siguiente resultado:

```
********** SpanishAnalyzer **********
á - 7045 ulis - 1705 telemac - 744 dios - 608 así - 547 [...]
Número de tokens en el fichero: 12747
```

Con estos resultados, podemos concluir que una buena opción para obtener el tema del que trata un texto sería usar el StopAnalyzer aplicándole el `SPANISH_STOP_WORDS_SET`, aunque también es una buena decisión usar SpanishAnalyzer (o algún tipo de LanguageAnalyzer dependiendo del idioma del texto), pues obtenemos muchos menos tokens que con el resto de analizadores, los cuáles son fácilmente interpretables.

2. **Probar sobre un texto relativamente pequeño el efecto que tienen los siguientes tokenFilters:**
   - StandardFilter: Lleva obsoleto desde la versión 7.5.0 de Lucene, y en la versión actual (8.2.0) ni si quiera aparece, por lo que no hemos podido ejecutarlo.
   - LowerCaseFilter: Pasa todo los tokens a minúscula.

     `ayer me compré un camión negro y por la tarde me comí un filete de ternera`
   - StopFilter: Elimina los tokens que se encuentra en el `array_set` que le pasemos.

     `Ayer compré camión negro tarde comí filete ternera`
   - SnowballFilter: Filtra las palabras según el idioma que le pasemos como parámetro mediante el método de *bola de nieve*.

     `Ayer me compr un camion negr y por la tard me com un filet de terner`
   - ShingleFilter: Este filtro crea combinaciones de tokens como un único token con una longitud dada.

     `Ayer - Ayer me - me - me compré - compré - compré un - un - un camión - camión - camión negro - negro - negro y - y - y por - ...`
   - EdgeNGramTokenFilter: Acorta los tokens a una longitud dada.

     `Aye com cam neg por tar com fil ter`
   - NGramTokenFilter: Acorta los tokens como en el anterior filtro, pero además crea nuevos tokens acortando los originales empezando en distintas posiciones de la palabra.

     `Aye yer com omp mpr pré cam ami mió ión neg egr gro por tar ard rde com omí fil ile let ete ter ern rne ner era`

- CommonGramsFilter: Combina tokens que aparecen con frecuencia, los cuales serán pasados como argumento. Aplicándolo a nuestro texto definiendo como términos de alta frecuencia las palabras "camión" y "filete" obtenemos:

  `Ayer me compré un un_camión camión camión_negro negro y por la tarde me comí un un_filete filete filete_de de ternera`

- SynonymFilter: Agrupa los distintos sinónimos en un único token. Para aplicar este filtro se necesita un mapa a de sinónimos, el cuál no nos proporciona Lucene y es complicado de implementar, por lo que no se ha ejecutado el filtro para comprobar su funcionamiento.

3. **Diseñar un analizador propio. Se os da libertad para poder escoger el dominio de ejemplo que consideréis mas adecuado, justificar el comportamiento.**

   Para este ejercicio, como se nos daba libertad para escoger nuestro dominio de ejemplo, hemos realizado 2 analizadores propios:

   - El primero se trata de un analizador de nombres llamado *NameAnalyzer*, donde hemos creado nuestra propia clase con el mismo nombre. Dado un archivo, se encargará de reconocer las palabras escritas en mayúsculas referentes a nombres propios, ciudades, etc, y borrar las palabras que vayan precedidas de un punto y no aporten información (artículos, determinantes, preposiciones...).

     Para ello, se ha realizado una función que *parsea* un archivo usando Tika. Además, dependiendo del idioma de nuestro fichero, usará el archivo `es.txt` (español) o `en.txt` (inglés) que contendrá las palabras que no aportan información para eliminarlas.

   - Para el segundo analizador propio, llamado *MyAnalyzer*, hemos usado la clase *CustomAnalyzer* la cuál facilitará la creación de un analizador a medida. Con ella, hemos establecido como *tokenizer* el *StandardTokenizer* y hemos añadido distintos filtros para poner todos los tokens en minúscula, darles la vuelta y poner su primera letra en mayúscula.

```
Analyzer myanalyzer = CustomAnalyzer.builder()
  .withTokenizer("standard")
  .addTokenFilter("lowercase")
  .addTokenFilter("reversestring")
  .addTokenFilter("capitalization")
  .build();
}
```

4. **Implementar un analizador específico que para un token dado se quede únicamente con los últimos 4 caracteres del mismo (si el token tiene menos de 4 caracteres es eliminado). Para ello, debemos de crear un**

TokenFilter y diseñar el comportamiento deseado y el método incrementToken.

Para este ejercicio hemos implementado un TokenFilter (last4CharFilter) que realice la comprobación de la longitud del token que le llega del analizador y posteriormente, si es válido, se quede con los últimos 4 caracteres.

La clase *last4CharFilter* hereda de la clase abstracta *TokenFilter* y utiliza el constructor de ésta. Hemos implementado un nuevo método `accept()` que indica si el token tiene una longitud mayor o igual a 4 para saber si hay que eliminarlo.

Además hemos sobrescrito el método `incrementToken()` tal y como se pide en el enunciado.En este método se comprueba si el token es válido, en cuyo caso se procede a el volcado en un nuevo buffer (`newBuffer`) de los 4 últimos caracteres, y se sustituye por el anterior buffer. Si no es válido se saltan las posiciones que ocupe el token y pasa al siguiente.

```
@Override
public final boolean incrementToken() throws IOException {
  int skippedPositions=0;
  while(input.incrementToken()){
    if (accept()) {
      //si nos hemos saltado algun token no valido anteriormente
          avanzamos posiciones con el atributo posIncrAtt
      if (skippedPositions != 0) {
        posIncrAtt.setPositionIncrement(posIncrAtt.
            getPositionIncrement() + skippedPositions);
      }
      //realizamos la copia de los 4 ultimos caracteres del token a un
          nuevo buffer y lo copiamos en termAtt
      int length = termAtt.length();
      char[] buffer = termAtt.buffer();
      char[] newBuffer = new char[4];

      for (int i = 0; i <4 ; i++) {
        newBuffer[3-i] = buffer[length-1-i];
      }

      termAtt.setEmpty();
      termAtt.copyBuffer(newBuffer, 0, newBuffer.length);
      return true;
    }
    //guardamos las posiciones avanzadas
    skippedPositions += posIncrAtt.getPositionIncrement();
  }
  // reached EOS -- return false
  return false;
}
```

Trabajo en Grupo

Para la realización de esta práctica cada integrante del grupo ha contribuido en cada uno de los ejercicios, aportando entre todos la información necesaria para el correcto funcionamiento de las actividades que se piden. Además, Simón López Vico ha profundizado en el ejercicio 1 y 2, Alberto Jesús Durán López ha profundizado en el ejercicio 3 y Miguel Cantarero López ha profundizado en el ejercicio 4.
Este informe se ha realizado entre los tres integrantes del grupo.