Práctica 2:
Preprocesado de documentos con Lucene.

UNIVERSIDAD DE GRANADA
E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Recuperación de Información (2020-2021)

Daniel Bolaños Martínez
Fernando de la Hoz Moreno
Grupo 10 - Martes 11:30h

Índice

1. Introducción.

El primer paso del proceso de indexación consiste en realizar un análisis del contenido textual del documento para romper el texto en pequeños elementos denominados *tokens*. La forma en que el texto de entrada se divide en tokens influye en cómo la gente podrá buscar ese texto, por lo que constituye una parte esencial de todo proceso de indexación.[1]

En esta práctica veremos cómo preprocesar documentos con la herramienta *Lucene* y aprenderemos a utilizar diferentes analizadores y herramientas de filtrado para extraer estos *tokens*.

Se pide realizar 4 ejercicios en los que trabajaremos con diversos analizadores y filtros que incluye *Lucene*:

- **Ejercicio 1**: Realizar un análisis estadístico sobre los diferentes *tokens* obtenidos usando varios analizadores predefinidos.
- **Ejercicio 2**: Probar sobre un texto pequeño diferentes *tokenFilters*.
- **Ejercicio 3**: Diseñar un analizador propio en un dominio de ejemplo.
- **Ejercicio 4**: Implementar un analizador específico que para un *token* dado, se quede únicamente con los últimos 4 caracteres del mismo.

Los documentos que vamos a usar en los diferentes ejercicios los almacenaremos en el directorio *docs*.

2. Ejercicio 1.

Sobre los documentos 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 la realización de este ejercicio, hemos creado un método llamado *StatisticAnalyzer* que se encarga de dividir el texto en *tokens* y realizar un conteo de la frecuencia total y parcial de cada *token* en función de los analizadores utilizados. Los resultados son guardados en el directorio *Estadisticas*.

```
public static void statisticAnalyzer(ArrayList<Analyzer> analyzers,
    ArrayList<String> types, String str, String file){
  try{
  /*Creamos archivos .csv con el recuento de las palabras de cada
      documento en CSV*/
    PrintWriter writer = new PrintWriter("./Estadisticas/" + file + ".txt
        ");
    for(int i=0; i < analyzers.size(); ++i){
    /*Como resultado del analizador estandar obtenemos un objeto
        TokenStream que nos permitira enumerar la secuencia de tokens.*/
      TokenStream stream = analyzers.get(i).tokenStream(null, str);
      CharTermAttribute cAtt = stream.addAttribute(CharTermAttribute.
          class);
      List<String> list = new ArrayList<String>();
      stream.reset();
      //Aniadimos todos los tokens del TokenStream a una lista
      while(stream.incrementToken()){
        list.add(cAtt.toString());
      }
      stream.end();
      //Creamos un Map con los tokens y sus frecuencias.
      Map <String, Integer > wordCounter = list.stream().collect(
          Collectors.toMap(w -> w, w -> 1, Integer::sum));
      //Ordenamos el Map segun los valores.
      wordCounter = sortByValue(wordCounter);
      Iterator it = wordCounter.keySet().iterator();
      /*Creamos archivos .txt con el recuento de los tokens de cada
          documento en Estadisticas*/
      writer.print("----" + types.get(i) + "----\n");
      writer.print("Numero de tokens en el archivo: " + wordCounter.size
          () + "\n");
      while(it.hasNext()){
        String key = (String) it.next();
        writer.print(key + ": " + wordCounter.get(key) + "\n");
      }
    }
    writer.close();
  } catch(IOException e){
    throw new RuntimeException(e);
    }
}
```

Hemos reutilizado el método *sortByValue* de la práctica anterior que es capaz de ordenar los *tokens* que se encuentran en el Map según su frecuencia de mayor a menor.

```
public static Map<String, Integer> sortByValue(final Map<String, Integer>
    wordCounts) {
/*Introduce las entradas del Map en un Stream, lo ordena con los valores
    del Map y crea otro con las entradas ordenadas y los valores del
    antiguo Map.*/
  return wordCounts.entrySet().stream().sorted((Map.Entry.<String,Integer
    >comparingByValue().reversed())).collect(Collectors.toMap(Map.Entry
    ::getKey, Map.Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));
}
```

Para cuatro analizadores diferentes, definiremos su funcionamiento y veremos los resultados estadísticos obtenidos al aplicarlos sobre el documento del *Quijote* (archivo *quijote.txt*)

WhiteSpaceAnalyzer

Divide el texto considerando como separadores de *tokens* los espacios en blanco.

El número de *tokens* diferentes obtenidos al analizar el documento ha sido de 39162. Los *tokens* más frecuentes son los siguientes.

que: 19236 de: 17745 y: 15706 la: 10073 a: 9483

SimpleAnalyzer

Divide el texto considerando como separadores de *tokens* todo aquello que no sean letras y convierte a minúsculas.

El número de *tokens* diferentes obtenidos al analizar el documento ha sido de 23246. Los *tokens* más frecuentes son los siguientes.

que: 20414 y: 17982 de: 17953 la: 10227 a: 9766

StandardAnalyzer

Es el más elaborado, y es capaz de gestionar acrónimos, direcciones de correo, etc. Convierte a minúscula y elimina de una lista de palabras vacías configurables.

El número de *tokens* diferentes obtenidos al analizar el documento ha sido de 23287. Los *tokens* más frecuentes son los siguientes.

que: 19236 de: 17745 y: 15706 la: 10073 a: 9483

SpanishAnalyzer

Basados en el castellano, permite la eliminación de palabras vacías de una lista específica para el idioma.

El número de *tokens* diferentes obtenidos al analizar el documento ha sido de 17062. Los *tokens* más frecuentes son los siguientes.

don: 2638 quijot: 2162 sanch: 2152 si: 1938 dijo: 1804

Podemos observar que en todos los analizadores menos el *SpanishAnalyzer* los *tokens* más repetidos son palabras vacías. *SpanishAnalyzer* al estar adaptado al castellano, elimina las palabras vacías del idioma y los *tokens* más frecuentes son términos más específicos del documento.

3. Ejercicio 2.

Probar sobre un texto relativamente pequeño el efecto que tienen los siguientes tokenFilters: StandardFilter, LowerCaseFilter, StopFilter, SnowballFilter, ShingleFilter, EdgeNGramTokenFilter, NGramTokenFilter, CommonGramsFilter, SynonymFilter.

Para realizar la prueba de los diferentes *tokenFilters*, hemos definido una función *probarTokenFilter* a la que le pasamos un *TokenStream* que contiene una secuencia de *tokens* enumerados extraídos por el analizador o *tokenFilter* y un *String* con el nombre del filtro.

```
//Imprime por pantalla los tokens y el nombre del tokenFilter usado.
public static void probarTokenFilter(TokenStream stream, String filter){
  System.out.println("\n——" + filter + "——\n");
  stream.reset();
  while(stream.incrementToken()){
    System.out.println(stream.getAttribute(CharTermAttribute.class));
  }
  stream.end();
  stream.close();
  System.out.println();
}
```

Utilizaremos como prueba, una de las expresiones más famosas del *Quijote*: *"En un lugar de la Mancha, de cuyo nombre no quiero acordarme."*

Para utilizar el *tokenFilter* debemos tener un objeto *TokenStream* extraído del texto a partir de un analizador, para ello hemos definido dos analizadores: **WhiteSpaceAnalyzer** y **StandardAnalyzer**. Indicaremos la separación de *tokens* con el símbolo '-' para no abusar del salto de línea.

- **WhiteSpaceAnalyzer**: divide el texto considerando como separadores de *tokens* los espacios en blanco. *Resultado*:

  En - un - lugar - de - la - Mancha, - de - cuyo - nombre - no - quiero - acordarme.

- **StandardAnalyzer**: convierte a minúscula, separa por signos de puntuación y puede gestionar acrónimos y direcciones de correo.

  Resultado:

  en - un - lugar - de - la - mancha - de - cuyo - nombre - no - quiero - acordarme

Para probar cada filtro, procederemos de forma análoga para cada *tokenFilter* salvo especificación de parámetros para algunos de ellos.

```
String cadena = "En un lugar de la Mancha, de cuyo nombre no quiero
    acordarme.";
Analyzer an = new WhitespaceAnalyzer();
probarTokenFilter(new LowerCaseFilter(an.tokenStream(null, cadena)), "
    WhitespaceAnalyzer+LowerCaseFilter");
```

A continuación, daremos para cada *tokenFilter* una breve explicación de su funcionamiento y el resultado obtenido para el texto de prueba:

StandardFilter

Este *tokenFilter* no se incluye en la versión *Lucene 8.6.2*. Su funcionamiento era muy similar al de **StandardAnalyzer** ya que, en versiones anteriores, estaba implementado por un *StandardFilter* junto con un *LowerCaseFilter* y un *StopFilter* de palabras en inglés.[3]

LowerCaseFilter

Convierte todos los *tokens* a minúsculas.

- Resultado con **WhiteSpaceAnalyzer**:

  en - un - lugar - de - la - mancha, - de - cuyo - nombre - no - quiero - acordarme.

- Resultado con **StandardAnalyzer**:

  en - un - lugar - de - la - mancha - de - cuyo - nombre - no - quiero - acordarme

  Como *StandardAnalyzer* ya implementa el paso a minúsculas, el filtro no afecta al *TokenStream*.

StopFilter

Elimina los *tokens* de la cadena que se encuentren exactamente en el conjunto de palabras vacías pasado como parámetro. Para obtener el conjunto de palabras vacías en español, cargamos el conjunto de palabras vacías contenido en el *SpanishAnalyzer*.

```
CharArraySet stopSet = SpanishAnalyzer.getDefaultStopSet();
probarTokenFilter(new StopFilter(stdan.tokenStream(null, cadena), stopSet
    ), "StandardAnalyzer+StopFilter");
```

- Resultado con **WhiteSpaceAnalyzer**:

  En - lugar - Mancha, - cuyo - nombre - quiero - acordarme.

- Resultado con **StandardAnalyzer**:

  lugar - mancha - cuyo - nombre - quiero - acordarme

  Podemos ver que como el *WhiteSpaceAnalyzer* no pasa a minúscula ni elimina los signos de puntuación, el comportamiento del *StopFilter* falla al eliminar algunas palabras vacías.

SnowballFilter

Realiza, según el idioma, un *stemming* de bola de nieve reduciendo cada *token* a su raíz. En nuestro caso pasando como parámetro "Spanish" obtenemos como *tokens* los lexemas de las diferentes palabras obtenidos siguiendo las reglas del castellano.

- Resultado con **WhiteSpaceAnalyzer**:

  En - un - lug - de - la - Mancha, - de - cuy - nombr - no - quier - acordarme.

- Resultado con **StandardAnalyzer**:

  en - un - lug - de - la - manch - de - cuy - nombr - no - quier - acord

  Como *StandardAnalyzer* elimina los signos de puntuación, al aplicar *SnowballFilter* se realiza el *stemming* a los *tokens Mancha* y *acordarme*.

ShingleFilter

Añade combinaciones de N *tokens* sucesivos como un único *token*. Usaremos el tamaño por defecto N=2.

- Resultado con **StandardAnalyzer**:

  en - en un - un - un lugar - lugar - lugar de - de - de la - la - la mancha - mancha - mancha de - de - de cuyo - cuyo - cuyo nombre - nombre - nombre no - no - no quiero - quiero - quiero acordarme - acordarme

EdgeNGramTokenFilter

Reduce los *tokens* a la longitud dada, tomando las N primeras letras, si es de menor longitud, lo elimina. Usaremos N=5.

- Resultado con **StandardAnalyzer**:

  lugar - manch - nombr - quier - acord

NGramTokenFilter

Reduce los *tokens* a una longitud dada como el filtro anterior, pero formando los *tokens* con subcadenas de N letras de cada palabra. Usaremos N=4.

- Resultado con **StandardAnalyzer**:

  luga - ugar - manc - anch - ncha - cuyo - nomb - ombr - mbre - quie - uier - iero - acor - cord - orda - rdar - darm - arme

CommonGramsFilter

Añade combinaciones de los *tokens* establecidos como *commonWords* junto con los *tokens* anterior y posterior a este como un único *token*. Para nuestro caso, estableceremos como *commonWords* las palabras: *lugar, mancha* y *nombre*.

- Resultado con **StandardAnalyzer**:

  en - un - un_lugar - lugar - lugar_de - de - la - la_mancha - mancha - mancha_de - de - cuyo - cuyo_nombre - nombre - nombre_no - no - quiero - acordarme

SynonymFilter

Añade *tokens* para los sinónimos de los *tokens* extraídos del texto. Para ello es necesario construir un diccionario de sinónimos.[4]

```
SynonymMap.Builder builder = new SynonymMap.Builder(true);
builder.add(new CharsRef("lugar"), new CharsRef("sitio"), true);
builder.add(new CharsRef("lugar"), new CharsRef("espacio"), true);
builder.add(new CharsRef("nombre"), new CharsRef("apelativo"), true);
builder.add(new CharsRef("mancha"), new CharsRef("Castilla La Mancha"),
    true);
SynonymMap synonymMap = builder.build();
```

- Resultado con **StandardAnalyzer**:

  en - un - lugar - sitio - espacio - de - la - mancha - Castilla La Mancha - de - cuyo - nombre - apelativo - no - quiero - acordarme

4. Ejercicio 3.

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

Nuestra idea es crear un analizador que gestione adecuadamente enlaces web, elimine los *tokens* demasiado cortos, las palabras vacías y los números irrelevantes. Para ello el analizador deberá obtener el idioma del archivo para realizar tanto el stemming como el filtrado de *stopwords* y reconocer los números que no cumplan la condición requerida.

Hemos construido el analizador propio sobreescribiendo el método *createComponents* de la clase **Analyzer** tal y como se nos indicaba en el guión de la práctica.[1]

Para extraer los *tokens* de los documentos utilizaremos el **UAX29URLEmailTokenizer** que es ideal para gestionar las direcciones de correo y enlaces web. Además implementaremos los siguientes *tokenFilters* cuyo funcionamiento ya se ha explicado a lo largo de la memoria.

- **LowerCaseFilter**: para pasar los *tokens* a minúsculas.
- **StopFilter**: para eliminar las palabras vacías.

- **NumerosFilter**: para eliminar números (a continuación explicaremos cómo funciona).
- **SnowballFilter**: para aplicar *stemming* de bola de nieve a los *tokens*.
- **LengthFilter**: para eliminar los *tokens* con longitud menor a la especificada.

```
public static Analyzer buildAnalyzer(final String language, final
    CharArraySet stopwords, final int min){
  return new Analyzer(){
    @Override
    protected TokenStreamComponents createComponents(String fieldname){
      final Tokenizer source = new UAX29URLEmailTokenizer();
      TokenStream result = new LowerCaseFilter(source);
      result = new StopFilter(result, stopwords);
      result = new NumerosFilter(result);
      result = new SnowballFilter(result, language);
      result = new LengthFilter(result, min, 1000);
      return new TokenStreamComponents(source, result);
    }
  };
}
```

El **NumerosFilter** se ha sobreescrito para que elimine los números decimales, esten separados por ',' o '.' y para eliminar los números con longitud diferente a 3 o 4 cifras. De esta forma, eliminamos todos los números que puedan ofrecer ruido y nos quedamos con aquellos que por ejemplo puedan referirse a años o fechas.

```
static class NumerosFilter extends FilteringTokenFilter{
  private final CharTermAttribute termAtt = addAttribute(
      CharTermAttribute.class);
  public NumerosFilter(TokenStream in){
    super(in);
  }
  @Override
  protected boolean accept() throws IOException{
    String token = new String(termAtt.buffer(), 0, termAtt.length());
  if(token.matches("[0-9]{1,2}") || token.matches("[0-9]{5,}") || token.
      matches("[0-9]+[.,][0-9]+"))
    return false;
  return true;
  }
}
```

Con ayuda de *Tika*, extraeremos el contenido de cada archivo del directorio como texto plano, analizaremos el contenido para reconocer el idioma del texto (sólo tendremos en cuenta español e inglés) y aplicaremos el diccionario de palabras vacías para **StopFilter** y el *stemming* para **SnowballFilter** dependiendo del lenguaje del documento.

```
//Construimos un analizador para cada idioma posible
//Los tokens de longitud menor que 4 seran eliminados en ambos casos
Analyzer an_es = buildAnalyzer("Spanish", SpanishAnalyzer.
    getDefaultStopSet(), 4);
Analyzer an_en = buildAnalyzer("English", EnglishAnalyzer.
    getDefaultStopSet(), 4);
```

```
//Identificamos el lenguaje del documento
LanguageIdentifier identifier = new LanguageIdentifier(contenido);
String idioma = identifier.getLanguage();
//Creamos archivos .txt con el resultado de aplicar nuestro Analyzer a
    cada documento
PrintWriter writer = new PrintWriter("./PARSER3/" + file + ".txt");
//Utilizamos un analizador diferente dependiendo del idioma
if(idioma.equals("es"))
  imprimirTokens(an_es.tokenStream(null, contenido), writer);
else
  imprimirTokens(an_en.tokenStream(null, contenido), writer);
```

Finalmente, aplicaremos el analizador construido y guardaremos los *tokens* obtenidos en la carpeta *TOKENS3* haciendo uso de la función *imprimirTokens*.

```
//Imprime los tokens extraidos por el analizador en un archivo haciendo
    uso de la clase PrintWriter
public static void imprimirTokens(TokenStream stream, PrintWriter writer)
     throws IOException{
  //offsetAtt indica la posicion del token
  OffsetAttribute offsetAtt = stream.addAttribute(OffsetAttribute.class);
  stream.reset();
  while(stream.incrementToken()){
    writer.print(stream.getAttribute(CharTermAttribute.class) + " : [" +
        (offsetAtt.startOffset()) + "," + offsetAtt.endOffset() + "]\n");
  }
  stream.end();
  stream.close();
  writer.close();
}
```

A continuación mostraremos los primeros *tokens* obtenidos al pasar el analizador al documento del *Quijote*.

```
primer : [0,7]
part : [8,13]
ingeni : [18,27]
hidalg : [28,35]
quijot : [40,47]
manch : [54,60]
capitul : [69,77]
primer : [78,85]
trat : [91,96]
condicion : [103,112]
ejercici : [115,124]
famos : [129,135]
hidalg : [142,149]
quijot : [154,161]
manch : [168,174]
manch : [194,200]
nombr : [210,216]
quier : [220,226]
acord : [227,236]
tiemp : [250,256]
hidalg : [270,277]
lanz : [288,293]
astiller : [297,306]
adarg : [308,314]
antigu : [315,322]
```

Figura 1: Tokens producidos por el analizador propio para el archivo *quijote.txt*

.

5. Ejercicio 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 en el método incrementToken

Creamos la clase *ultimas4Letras* que hereda de la clase *TokenFilter*. Esta sobreescribe el método *incrementToken* para que devuelva las últimas cuatro letras de los tokens obtenidos por el objeto *TokenStream* que se ha pasado al contructor. El método *accept* nos indica si tiene menos de cuatro letras para que no se tenga en cuenta dicho *token*.

```
// Sobreescribimos la clase TokenFilter para crear nuestro filtro
public class ultimas4Letras extends TokenFilter{
  private final CharTermAttribute cAtt = addAttribute(CharTermAttribute.
     class);
  public ultimas4Letras(TokenStream in){
    super(in);
  }
  /*Metodo accept que se encarga de decir que tokens devuelve
     incrementToken y cuales no.*/
  public boolean accept() throws IOException{
    return cAtt.length() >= 4;
  }
  //Sobreescribimos incrementToken para obtener los tokens deseados
  @Override
  public final boolean incrementToken() throws IOException{
    //Vamos recorriendo los tokens del TokenStream que nos han pasado en
       el constructor.
    while(input.incrementToken()){
      /*Si el token no es de longitud menor que cuatro modificamos el
         token y sus
        atributos para quedarnos como token las ultimas 4 letras del
           mismo.*/
      if(accept()){
        char[] buffer = new char[4];
        char[] s = cAtt.buffer();
        for(int i=0; i < 4; ++i)
          buffer[i] = s[cAtt.length()-(4-i)];
        cAtt.setLength(4);
        cAtt.copyBuffer(buffer, 0, 4);
        return true;
      }
    }
    return false;
  }
}
```

Creamos el método *tokenizeString* el cual, haciendo uso de la clase *ultimas4Letras*, obtiene las últimas cuatro letras de los *tokens* extraídos por el analizador de nombre el pasado como argumento. También imprimiremos para cada *token* su posición de inicio y finalización en el texto.

```
public static void tokenizeString(Analyzer analyzer, String string,
    PrintWriter writer){
     try{
     /*Obtenemos del analizador un TokenStream que nos permite enumerar la
          secuencia de tokens.*/
       TokenStream stream = analyzer.tokenStream(null, string);
       /*Creamos un TokenStream que elimina las palabras de tamanio menor
           que 4 del anterior, y las que no las reduce a sus ultimas 4
           letras.*/
       stream = new ultimas4Letras(stream);
       //cAtt es el token y offsetAtt su posicion
       OffsetAttribute offsetAtt = stream.addAttribute(OffsetAttribute.
           class);
       CharTermAttribute cAtt = stream.addAttribute(CharTermAttribute.
           class);
       //Obtenemos los tokens y los imprimimos junto con sus posiciones.
       stream.reset();
       while(stream.incrementToken()){
         writer.print(cAtt.toString() + " : [" + offsetAtt.endOffset()-4 +
              "," + offsetAtt.endOffset() + "]\n");}
       stream.end();
     } catch(IOException e){
       throw new RuntimeException(e);}}
```

Mostramos los primeros *tokens* obtenidos al pasar el analizador al documento del *Quijote*.

```
mera : [3,7]
arte : [9,13]
ioso : [23,27]
algo : [31,35]
jote : [43,47]
ncha : [56,60]
tulo : [73,77]
mero : [81,85]
rata : [92,96]
ción : [108,112]
icio : [120,124]
moso : [131,135]
algo : [145,149]
jote : [157,161]
ncha : [170,174]
ugar : [183,187]
ncha : [196,200]
cuyo : [205,209]
mbre : [212,216]
iero : [222,226]
arme : [232,236]
ucho : [245,249]
```

Figura 2: Tokens producidos por el analizador específico para el archivo *quijote.txt*

6. Método de compilación.

Ejecutamos el *shell* de la práctica haciendo uso de la orden siguiente:

```
> ./practica2.sh -option
```

Donde -*option* se refiere al número del ejercicio que se quiera ejecutar $[-1, -2, -3, -4]$.

7. Trabajo en Grupo.

El trabajo lo hemos repartido, en primera instancia, de la siguiente manera:

- **Daniel Bolaños Martínez**: Ejercicios 2 y 3.
- **Fernando de la Hoz Moreno**: Ejercicios 1 y 4.

No obstante, hemos mantenido el contacto durante el desarrollo de la práctica y hemos colaborado conjuntamente en la elaboración de la memoria y del proyecto.

Referencias

[1] Guión de la práctica 2 de la asignatura.

[2] `https://lucene.apache.org/core/8_6_2/core/index.html`

[3] `https://lucene.apache.org/core/5_5_1/analyzers-common/org/apache/lucene/analysis/standard/StandardAnalyzer.html`

[4] `https://www.programcreek.com/java-api-examples/?api=org.apache.lucene.analysis.synonym.SynonymFilter`