Práctica 2 Análisis de Texto

Para realizar la práctica hemos partido de nuestra clase `DocumentAnalyzer` para incorporar los diferentes análisis que se nos pide, de tal forma que quede la práctica más limpias y cohesionada.

Apartado 1

En DocumentAnalyzer, hemos sobrecargado el método contador que teniamos en la práctica anterior, la cual tokenizaba las palabras de nuestro texto y las contaba, ahora nuestro método sobrecargado recibe también el tipo de analizador que queremos utilizar.

En nuestro caso hemos dado como opción estos analizadores:

- whiteAnalyzer
- simpleAnalyzer
- stopAnalyzer
- spanishAnalyzer
- standardAnalyzer (por defecto)

Además, también va incluido el analizador que hemos utilizado en la práctica 3.

Una vez seleccionado el analizador, obtenemos el contenido del archivo usando lo que ya teniamos en la clase de la práctica 1 (usamos Tika para obtenerlo).

Abrimos el TokenStream y la recorremos obteniendo todos los CharTermAttribute, por cada uno realizamos lo mismo que con el otro contador, los vamos guardando en un hashmap, y si el elemento ya ha sido introducido, se incrementa el valor asociado.

Finalmente, lo convertimos a array y lo ordenamos de igual forma que hicimos en la práctica anterior, por ese motivo, creamos el método privado `hashMapToSortedArray()`.

Finalmente en el main, nos recorremos todos los archivos del directorio test, y por cada uno lo procesamos con los diferentes analizadores, y los guardamos en carpetas separadas por el analizador utilizado.

Análisis

Para realizar el análisis tomaremos los csv de ideal como referencia, para cada uno de los analizadores

- `WhitespaceAnalyzer`: Nos encontramos con una lista con todas las palabras que nos podemos encontrar en el documento, ya que se ha tokenizado usando el espacio como separador, sin usarse ningún filtro
- `StopAnalyzer`: En este caso se filtran las palabras vacías que tenemos indicadas en el documento `dictionaries/stopwords.txt`, por tanto desaparecen las preposiciones, artículos, y otros términos que se consideran que no aportan semántica al texto. Podemos ver como hay palabras que no es capaz de filtrar, terminos extraidos que no existen en el castellano y por tanto tampoco encontraremos en nuestra lista de stopwords
- `StandardAnalyzer`: En este caso, vemos que siguen apareciendo las stopwords a pesar de que este analizador debería ser capaz de filtrarlos, esto ocurre porque no le estamos indicando que use

nuestra lista de stopwords, por tanto está utilizando su lista por defecto, la cual es una lista de stopwords para la lengua inglesa.

- `SimpleAnalyzer`: Funciona de forma similar a WhitespaceAnalyzer, pero podemos ver que SimpleAnalyzer obtiene un mayor número de tokens que WhitespaceAnalyzer, esto se debe a que SimpleAnalyzer también tokeniza aquellas palabras que van separadas por signos de puntuación y no solamente por espacios.
- `SpanishAnalyzer`: En este caso, nos elimina las palabras vacías correctamente (al usar por defecto las de castellano), y además simplifica las palabras a su raíz, aunque en este caso vemos que ciertos términos sin sentido se mantienen de igual forma que ocurría con el StopAnalyzer.

Apartado 2

En DocumentAnalyzer, hemos implementado un metodo al que hemos nombrado como `applyDifferentFilter(int i)` que recibe como parametro un entero que usamos en la sentencia `Switch` que se encuentra dentro del método.

En dicho `Switch` dependiendo del entero (que se establece en el main de la práctica) se aplica un filtro diferente. La lógica del metodo es la siguiente.

Para cada `case` se construye un analizador customizado en el que se va variando para cada caso el filtro utilizado. Tras haber establecido el analizador, se recupera la salida como `List<String>`.

Mas concretamente, para los filtros donde ha sido necesario, se ha hecho uso de diccionarios creados por nuestra cuenta o conseguidos en la web. Es el caso de los filtros `StopFilter` y `SynonymFilter`.

Los archivos utilizados se pueden encontrar en el directorio dictionaries.

En otros, ha sido necesario definir varios parámetros. Es el caso de `CommonGramsFilter` y `NGramTokenFilter` donde se ha establecido la variable `gramSize = 2` para la creacion de los bigramas.

Por ultimo, el filtro `SnowballFilter` hemos utilizado el stemmer "Spanish".

En `P2.main()` se recogen las diferentes salidas de los filtros adjuntando su propio nombre. Tras esto, la variable que recoge los textos filtrados es pasada como parametro al metodo `txtWriterFromStringList` de la clase `OutputHelper` para su escritura en una archivo de resultado alojado en results/differentFiltersResults/differentFiltersResults.txt.

Apartado 3

Para realizar el analizador, damos la opción en el método contador de usar nuestro propio analizador, de tal forma que nos queda este fragmento de código:

```
analyzer = new Analyzer() {
  @Override
  protected TokenStreamComponents createComponents(String fieldName) {
    try {

      // Importamos los diccionarios
      InputStream affixStream = new
FileInputStream("P2/dictionaries/es.aff");
```

```
        InputStream dictStream = new
FileInputStream("P2/dictionaries/es.dic");

        // Carpeta temporal para el diccionario
        Directory directorioTemp = FSDirectory.open(Paths.get("P2/temp"));
        Dictionary dic = new Dictionary(directorioTemp, "temporalFile",
affixStream, dictStream);

        // Tokenización
        Tokenizer source = new UAX29URLEmailTokenizer();

        // Filtramos las palabras vacías
        File stopWordsFile = new File("P2/dictionaries/stopwords.txt");
        DocumentAnalyzer stopWordsDocumentAnalyzer = new
DocumentAnalyzer(stopWordsFile);
        Collection<String> stopWordsCollection = Arrays
            .asList(stopWordsDocumentAnalyzer.getContenido().split("\\r?
\\n"));
        CharArraySet stopWords = new CharArraySet(stopWordsCollection,
false);

        TokenStream result = new StopFilter(source, stopWords);

        result = new HunspellStemFilter(result, dic, true, true);

        return new TokenStreamComponents(source, result);
      } catch (Exception e) {
        e.printStackTrace();
      }
      return null;
    }
};
```

Para ello hemos utilizado:

Tokenizer

- `UAX29URLEmailTokenizer`: Tokenizará nuestro texto incluyendo emails y URLS.

Filtros

- `StopFilter`: Filtrará las palabras vacías que le indiquemos.
    - Hemos usado el archivo stopwords.txt, donde indicamos que palabras vacías queremos que se filtren
- `HunspellStemFilter`: A diferencia de SpanishAnalyzer, en vez de obtener la raíz. tranforma las derivaciones de los tokens en la palabra principal, por ejemplo en verbos nos lo transformará a infinitivo Hemos usado los siguientes archivos adicionales para realizarlo:
    - Affix dictionary (es.aff): Lista con los diferentes prefijos y sufijos
    - Diccionario castellano (es.dic): Lista de palabras en castellano

Apartado 4

Para realizar este apartado se ha creado una nueva clase `CustomFilter` para la implementación de un filtro customizado. Esta clase extiende la superclase `FilteringTokenFilter`:

```
public class CustomFilter extends FilteringTokenFilter {

    private final CharTermAttribute termAtt =
addAttribute(CharTermAttribute.class);

   //Constructor
    public CustomFilter(TokenStream in) {
      super(in);
    }

    @Override
    protected boolean accept() throws IOException{
        String token = new String (termAtt.buffer(),0, termAtt.length());
        if(token.length()<=4){
            return false;
        }else{
            //Reescribimos el token con la cadena compuesta por los
ultimos 4 caracteres.
            termAtt.copyBuffer(token.toCharArray(), token.length()-4, 4);
            return true;
        }
    }
}
```

En este filtro customizado se ha implementado el método abstracto `accept()` para modificar la manera en que se aceptan los tokens de entrada. De esta forma, aquellos tokens que tienen una longitud mayor a cuatro son eliminados o mas correctamente dicho, no aceptados y aquellos con longitud mayor se modifican reeescribiendose los ultimos cuatro caracteres de los mismos.

El resultado es pasado como parámetro en una llamada al método `txtWriterFromStringList` de la clase `OutputHelper` para volcar la salida al archivo resultado en la ruta P2/results/text-4/text-4.txt.

¿Qué ha hecho cada miembro?

Adrián

- Apartado 2
- Apartado 4

Andrés

- Apartado 1
- Apartado 3