Memoría Recuperación de Información

¿Qué ha hecho cada miembro?

Adrián

- Apartado (-d)
- Clase DocumentAnalyzer.java
- Método csvWriterMetadata de OutputHelper

Andrés

- Clase DocumentAnalyser.java
- Métodos csvWriter y print de OutputHelper
- Apartado (-l)
- Apartado (-t)

Cómo lo hemos hecho

Recorrer directorio

Cogemos el segundo parámetro de la terminal (El nombre del archivo), creamos un File con ese nombre, con `listFiles()` obtenemos los archivos que cuelgan del directorio (ignorando subdirectorios), de esta forma tenemos el listado de archivos que analizaremos.

Flags de los ejercicios

El parámetro de terminal 0 será donde ponemos el flag (-d, -l, -t), se llama a la función determinada dependiendo del flag que se haya puesto.

Dentro de cada función, se recorrerá la lista de archivos que teníamos, y por cada archivo utilizaremos la clase auxiliar DocumentAnalyzer, el cual nos permitirá a los datos necesarios encapsulando el analisis del documento del main, finalmente usaremos la clase auxiliar OutputHelper para mostrar en pantalla o guardar en csv los datos que nos aporta DocumentAnalyzer.

DocumentAnalyzer

Es la clase encargada de analizar el documento y obtener su nombre, metadatos, contenido, enlaces y la lista de palabras.

Constructor

En el constructor generamos todos los datos que podemos extraer directamente con Tika

- Nombre
- Contenido
- Enlaces

- Lenguaje
- Metadatos

Para ello usamos directamente el `AutoDetectParser`

Métodos

Los métodos se encargan de extraer información más específica y en un formato concreto:

- contador()

  Este método devuelve una lista de palabras junto al número de ocurrencias que tiene en el documento

  Para ello toma el contenido del documento y con split, se para las palabras por espacios, ahora por cada palabra, comprobamos que no contiene carácteres extraños con la expresión regular `[a-zA-Z\\u00C0-\\u024F\\u1E00-\\u1EFF]+`, que basicamente comprueba que este formado únicamente por carácteres y letras con tilde.

  Para poder ir contando si aparece o no, vamos metiendo las palabras en un **HashMap**, de tal forma de que podemos comprobar rápidamente si la palabra ya ha aparecido, y cuantas veces lleva, y poder ir introduciendo palabras e incrementar el contador de las que ya estaban.

  Después lo pasamos a un `ArrayList` para poder ordenarlo, para ello hemos usado una función de comparación para luego usar en el método de ordenación `Collections.sort()`, que comparará por el número de ocurriencias de cada palabra.

  Para pasar de `HashMap` a `ArrayList`, necesitamos de forma intermedia un Set.

OutputHelper

Esta clase tiene funciones para asistir a la hora de mostrar en pantalla o exportar los datos en .csv

- `csvWriter()`

Toma la lista con palabra-valor, y lo va introduciendo linea a linea con el formato adecuado de .csv (separado en este caso por ';')

- `csvWriterMetadata()`

Representa la tabla en csv a partir de las diferentes columnas dadas (Nombre, Tipo, Codificación, Idioma)

- `print()`

En este caso imprime la lista de enlaces dada en pantalla

Archivos de prueba

En la carpeta ./test tenemos los diferentes documentos que se analizarán, hemos cogido un sample de .xml, un .doc, un .odt, y por último un html extraido del Ideal y el pdf con la memoria de prácticas.

Resultados

En la carpeta ./results se encuentran los resultados que hemos extraido con nuestro programa (.csv), la nube de palabras, y la gráfica de la Ley de Zipf.

Ambas la hemos realizado con el artículo del Ideal al tener un contenido más completo que el resto.