

Preprocesado de documentos

Parte II. Análisis del texto

September 28, 2023

1 Realización de la práctica

La práctica se realizará en grupo. Al final de la práctica se debe entregar un informe que necesariamente debe incluir una sección denominada *Trabajo en Grupo* en el que se indicará de forma clara la contribución de cada alumno. Se recomienda seguir la metodología SCRUM para el desarrollo del proyecto.

2 Manipulando el lenguaje humano

La recuperación de información textual representa una batalla entre precisión (devolver el menor número de documentos irrelevantes posibles) y exhaustividad (devolver el mayor número posible de documentos relevantes). Utilizar un modelo que sólo considere el emparejamiento exacto entre términos no es útil en la mayoría de las aplicaciones de búsqueda. De ser así, es posible que perdamos muchos documentos relevantes que contienen términos que aún sin ser idénticos a la consulta están relacionados con la misma.

Situaciones que se pueden considerar son:

- No diferenciar entre singulares o plurales (perro vs perros) o tiempos verbales (comiendo, comido, comerá). Para ello, será útil reducir una palabra a su raíz.
- Presencia/ausencia de términos sin significado. Eliminar palabras como *el, la, los, y, ni* puede incrementar la calidad de la recuperación,

además de reducir el tamaño del índice.

- La inclusión de sinónimos (buscar por `coche` y que nos devuelva documentos que contengan el término `automóvil`).
- Proporcionar la corrección de errores de forma automática.

Pero antes de considerar los términos de forma aislada debemos de ser capaces de dividir el texto en términos, y por tanto debemos de conocer qué es un término. Veremos como podemos utilizar Lucene para este fin.

2.1 Cómo se realiza el análisis

El primer paso cuando trabajamos con un texto, para crear un índice o extraer conocimiento (minería de textos) del mismo, es el identificar cuales son los atributos (tokens) de los que está compuesto. Aunque parezca un proceso sencillo, son varios los pasos que se pueden realizar hasta determinar si una secuencia de caracteres es un token o no. Por ello, la mayoría de las herramientas proporcionan algún mecanismo para automatizar esta tarea basándose en algún algoritmo estándar. Por ejemplo, ante la cadena:

Obtener los TOKENS (términos-de-indexación) permitirá indexar, eficientemente, un documento de la colección.

Podríamos obtener el siguiente conjunto de tokens:

[Obtener] [los] [TOKENS] [términos] [de] [indexación] [permitirá]
[indexar] [eficientemente] ...

De resultado del análisis dependerá en gran medida la calidad de nuestro sistema, por lo que será necesario estudiar con detalle qué tipo de análisis conviene para nuestro objetivo final. Por ejemplo, si estamos implementando un buscador donde los documentos relevantes a una consulta se determinan al considerar el emparejamiento entre consulta y los tokens del documento podemos decir que ante la consulta "TOKENS" nuestra cadena sería relevante, pero no lo sería si consideramos como término de la consulta a "TOKEN" o "tokens". En este caso podríamos preferir como resultado del análisis la secuencia:

[obten] [los] [token] [termin] [de] [index] [permit] [index] [eficiente]

Por tanto, son muchas las situaciones en las que será necesario modificar el criterio utilizado por defecto en la aplicación, o incluso desarrollar métodos propios, para adaptarlo a nuestras necesidades particulares (como por ejemplo sería solo considerar los tokens con un tamaño entre 3 y 12 caracteres).

En esta práctica profundizaremos en el conocimiento del proceso de análisis de texto, en concreto nos centraremos en cómo podemos utilizar la librería Lucene para dicho proceso. Se recomienda consultar el javadoc de la versión de Lucene que se utilice (en la práctica asumimos 7.1.0, aunque a día de hoy está la versión 9.7.0)

```
https://lucene.apache.org/core/9\_7\_0/core/index.html?overview-summary.html
```

también es interesante este enlace

```
https://lucene.apache.org/\_\_root/docs.lucene.apache.org/core/9\_7\_0/demo/
```

Para realizar el análisis Lucene utiliza una secuencia de operaciones sobre el texto de entrada, entre las que se puede incluir separar por blancos, eliminar símbolos de puntuación, eliminar tildes, convertir a minúscula, eliminar palabras comunes (determinantes, preposiciones, etc.), reducir el término a su raíz e incluso cambiar términos. En muchos casos, estas operaciones ya vienen implementadas bajo un analizador por lo que no requiere mucho esfuerzo de programación, en otros sólo será necesario implementar alguna parte del proceso e integrarlo dentro de la librería.

3 Apache Lucene

Apache-Lucene <https://lucene.apache.org/core/> es una biblioteca, escrita completamente en Java, y diseñada para la implementación de un motor de búsqueda de alto rendimiento. Actualmente se encuentra disponible para su

descarga la versión 9.3.0. En el caso en que los documentos que queremos buscar dispongan de una estructura (por ejemplo, un email tiene subject, body, from, to, etc.) podemos aprovecharla para mejorar la calidad de la búsquedas.

Lucene es el núcleo de servidores de búsqueda como Solr y Elasticsearch. También puede incorporarse a aplicaciones Java, Android o backends web.

Lucene ofrece potentes funciones a través de una sencilla API que permite:

- Indexación escalable y de alto rendimiento
 - Más de 150 GB / hora en hardware moderno
 - Requisitos de RAM pequeños - sólo 1 MB de Heap
 - Indexación incremental tan rápido como la indexación por lotes
 - El tamaño del índice es aproximadamente 20-30% del tamaño del texto indexado
- Algoritmos de búsqueda potentes, precisos y eficientes
 - Búsqueda ordenada - los mejores resultados devueltos primero
 - Consulta potentes: consultas de frases, consultas comodín, consultas de proximidad, consultas de rango y más
 - Búsqueda por campos (por ejemplo, título, autor, contenido)
 - Ordenación por cualquier campo
 - Búsqueda en múltiples índices
 - Permite la actualización y búsqueda simultáneas
 - Búsqueda por categorías (facetas), resaltado (highlighting), agrupación de resultados.
 - Incluye distintos modelos de búsqueda.

4 Procesado de textos con Lucene

La entrada de Lucene es un texto sin formato, y el primer paso del proceso de indexación consiste en realizar un análisis del contenido textual del documento

para romper el texto en pequeños elementos de indexación - 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. Como es normal, el análisis es una de las principales causas de la indexación lenta. En pocas palabras, cuanto más se analiza, más lenta es la indexación (en la mayoría de los casos).

Lucene nos proporciona un conjunto de métodos que nos facilitan esta tarea. Este es el objetivo de esta práctica, donde utilizaremos la terminología Lucene, donde la clase encargada de todo el proceso es `Analyzer`

En Lucene la clase `Analyzer` llama a tres procesos que se ejecutan secuencialmente:

- Filtrado de caracteres, que se encarga de procesar el texto eliminando caracteres como podrían ser signos de puntuación, paréntesis, etiquetas, etc.
- Tokenizador, encargado de separar los tokens
- Filtrado de Tokens, donde se realizan tareas como la eliminación de palabras vacías, stemming, normalización del texto, expansión de sinónimos. Estos procesos de filtrado se pueden encadenar para producir el token deseado

4.1 Ejemplos de Analyzer implementados en Lucene

Podemos encontrar distintos `Analyzer` ya implementados. `Analyzers` la podemos encontrar en https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/Analyzer.html. Entre ellos, destacaremos:

- `KeywordAnalyzer`: Considera el texto como un único token
- `WhiteSpaceAnalyzer`: Divide el texto considerando como separadores de tokens los espacios en blanco
- `SimpleAnalyzer`: Divide el texto separando por todo aquello que no sean letras y convierte a minúsculas.

- `StopAnalyzer`: Hace lo mismo que el `SimpleAnalyzer` pero elimina las palabras vacías (sin significado). Viene con una lista de palabras vacías predefinidas, pero como es obvio podemos darle las nuestras.
- `StandardAnalyzer`. Es el más elaborado, y es capaz de gestionar acrónimos, direcciones de correo, etc. Convierte a minúscula y elimina palabras vacías.
- `UAX29URLEmailAnalyzer`. Diseñado específicamente para trabajar con URL y direcciones de email, incluyendo además y filtrado de palabras vacías y conversión a minúsculas.
- Basados en Idiomas, permitiendo la eliminación de palabras vacías:
 - `EnglishAnalyzer`, en inglés.
 - `SpanishAnalyzer`, en castellano.
 - `FrenchAnalyzer`, en francés,
 - ...

También permite tratar los distintos campos de un texto de forma distinta utilizando un `PerFieldAnalyzerWrapper`. Así, por ejemplo, si queremos indexar emails, podríamos utilizar un `StandardAnalyzer` para el cuerpo del texto y un `KeywordAnalyzer` para los campos `From` y `To`, por ejemplo.

5 Estudio de Analyzers

Normalmente, cuando tratamos de indexar un documento o realizamos una consulta, nos limitamos a indicarle a Lucene qué analizador queremos utilizar de entre los que ya vienen contruidos, o por el contrario decidimos crear uno propio. En cualquier caso, podemos considerar que en el uso normal de un analizador se toma como entrada el documento y su salida pasa directamente al proceso de indexación. Nosotros no llegamos a ser conscientes del resultado del análisis.

Si embargo, en esta práctica si queremos ver con más detalla el resultado del mismo. Será de utilidad para saber exactamente qué tokens serán indexados finalmente.

Con esta finalidad, y a modo de ejemplo, podemos utilizar el plugin Analyzer Tool de Luke: Luke nos proporciona una interfaz gráfica para ver un índice Lucene, pero en este caso nos centraremos en considerar el plugin Analyzer para ver que tokens son obtenidos por cada analizador.

Las últimas versiones de Lucene contienen integrado a Luke (lo podemos encontrar en el subdirectorio `luke` de la versión de Lucene que descarguemos) y ejecutar el fichero `.jar`.

Al ejecutar Luke, la primera opción es abrir un índice ya creado, pero nosotros no lo haremos en esta práctica, sino que nos vamos directamente a la pestaña Analysis. Una vez que estamos en esta ventana, ya podemos seleccionar el analizador que nos interesa dentro del desplegable y proporcionar el texto que queremos analizar. Podremos ir viendo cada uno de los tokens generados, como nos muestra la imagen de la Figura 1.

5.1 Ejemplo simple de uso de un analizador

Una vez que hemos experimentado un poco con Luke, pasaremos a ilustrar como llamar a un Analyzer desde nuestros programas. Recordad que este proceso no es el normal en una etapa de indexación, sólo lo haremos para comprender mejor todo el proceso.

En esta sección veremos como podremos crear una pequeña aplicación Java que utilice un analizador. Para ello, nos creamos un nuevo proyecto Java, incluyendo las librerías de Lucene (asumo versión 7.1.0), por tanto, tenemos que incluir en el classpath o en las dependencias del proyecto los siguientes paquetes:

- `lucene-core-7.1.0.jar` que se encuentra en `lucene-7.1.0/core`
- `lucene-analyzers-common-7.1.0.jar`, que se encuentra dentro del directorio

`lucene-7.1.0/analysis/common/lucene-analyzers-common-7.1.0.jar`

El objetivo de la aplicación será listar por consola todos y cada uno de los tokens que se obtienen al aplicar un `WhitespaceAnalyzer`, para lo que utilizaremos en el siguiente código:

Figure 1: Ejemplo de uso del StandardAnalyzer con Luke.

Luke: Lucene Toolbox Project - v9.7.0

FileToolsHelp

OverviewDocumentsSearchAnalysisCommitsLogs

ConfigDir

BrowseBuild AnalyzerLoad external jars

Char Filters

Selected

Show & Edit

+ Add

Tokenizer

Selectedstandard

Show & Edit

⚙ Set

Token Filters

SelectedspanishLightStem

Show & Edit

+ Add

Selected Analyzer: org.apache.lucene.analysis.custom.CustomAnalyzer (Show analysis chain)

Esto es ejemplo de como hablabamos y comiamos

💡 Test Analyzer☒ Step By StepClear

Hint: Double click the cell to show all token attributes.

Name	Text							
standard	Esto	es	ejemplo	de	como	hablabamos	y	comiamos
spanishLightStem	Esto	es	ejempl	de	como	hablabam	y	comiam
lowercase	esto	es	ejempl	de	como	hablabam	y	comiam


```
1 package analizador1;  
2  
3 import java.io.IOException;  
4  
5 import org.apache.lucene.analysis.Analyzer;  
6 import org.apache.lucene.analysis.TokenStream;  
7 import org.apache.lucene.analysis.core.WhitespaceAnalyzer;  
8 import org.apache.lucene.analysis.tokenattributes.  
    CharTermAttribute;  
9  
10 public class Analizador1 {  
11  
12     public static void main(String[] args) throws IOException {  
13         Analyzer an = new WhitespaceAnalyzer();  
14         String cadena = "Ejemplo de analizador + WhiteSpace ,  
            lucene -7.1.0";  
15         TokenStream stream = an.tokenStream(null, cadena);  
16  
17         stream.reset();  
18         while (stream.incrementToken()) {  
19             System.out.println(stream.getAttribute(CharTermAttribute.  
                class));  
20         }  
21         stream.end();  
22         stream.close();  
23     }  
24  
25 }
```

Así, en la línea 13 del código anterior nos creamos una instancia de la clase `WhitespaceAnalyzer` que será el encargado de procesar la cadena que se le pasa como entrada, línea 15 de código donde, como resultado de proceso obtenemos un `TokenStream`¹, que nos permitirá enumerar la secuencia de tokens. Lucene, cuando tokeniza una cadena obtiene por un lado el término, pero también recupera otros atributos asociados al mismo, como puede la posición del término, el tipo de atributo, etc. Estos atributos se pueden recuperar en caso de necesitarlo

¹https://lucene.apache.org/core/7_1_0/core/index.html?org/apache/lucene/analysis/TokenStream.html

a través de la API de `TokenStream` (ver documentación de la API). En nuestro caso, sólo recuperaremos el término en si (`CharTermAttribute`) (línea 19).

Brevemente (lo veremos con más detalle después), tras inicializar el stream (reset), avanzamos por cada uno de los tokens, hasta que `incrementToken` devuelve falso, recuperando los atributos que deseemos. Al final debemos de indicar que hemos terminado (end) y liberar los recursos asociados al stream (close) .

Como resultado, obtenemos

```
1 Ejemplo
2 de
3 analizador
4 +
5 WhiteSpace ,
6 lucene -7.1.0
```

Utilizar otro analizador es fácil, por ejemplo, podemos modificar la línea 13 para utilizar un `SimpleAnalyzer`,

```
1 Analyzer an = new SimpleAnalyzer();
```

y obtendremos como salida

```
1 ejemplo
2 de
3 analizador
4 whitespace
5 lucene
```

6 Clases encargadas del análisis

Hay cuatro clases principales encargadas de realizar el análisis, además de `Analyzer` que es la clase mas general, en el proceso se ejecuta una secuencia de pasos (`CharFilter+Tokenizer+TokenFilter`). A simple vista, la relación entre ellas puede parecer confusa, por lo que es conveniente entender los siguientes elementos que forman parte del proceso del análisis.

- `CharFilter`: Extiende la clase `Reader` para transformar el texto antes de ser tokenizado, pero controlando los offsets (desplazamientos) de los caracteres

corregidos. Pueden modificar el texto de entrada añadiendo, eliminando o transformando caracteres.

- **Tokenizer:** Es un `TokenStream` y es el responsable de romper el texto en tokens. Un token representa un término o palabra del documento, teniendo asociado elementos como su posición, offset de inicio y fin, tipo de token, etc.

En algunos casos, basta con romper el texto de entrada considerando la separación por espacios en blanco, tabuladores o saltos de línea, pero en otros muchos es necesario un análisis más profundo. Por ejemplo, cuando deseamos construir N-gramas o se utilizan expresiones regulares para construir distintos tokens

- **TokenFilter:** Es un `TokenStream` y el responsable de modificar los tokens obtenidos por el `Tokenizer` pudiendo
 - **Stemming** - Sustitución de palabras con por sus raíces. Por ejemplo, en castellano, perro y perra se unen en un único término con raíz 'perr'.
 - **Eliminación de palabras vacías** - las palabras comunes como "el", "y" y "a" raramente agregan cualquier valor a una búsqueda. Su eliminación reduce el tamaño del índice y aumenta el rendimiento. También puede reducir el "ruido" y mejorar la calidad de la búsqueda.
 - **Normalización de texto** - Eliminar acentos y otras marcas de caracteres pueden mejorar la búsqueda.
 - **Expansión de Sinónimos** - Añadir sinónimos en la misma posición simbólica a la palabra actual puede ayudar a una mejor coincidencia cuando los usuarios buscan con palabras en el conjunto de sinónimas.
- **Analyzer:** Es el encargado de construir un `TokenStream` que pueda ser digerido por los procesos de indexación como de búsqueda. No se encarga de procesar el texto, para ello utiliza a elementos del tipo `CharFilter`, `Tokenizer` y `TokenFilter`.

Si queremos utilizar una combinación concreta de `CharFilter`, `Tokenizer` y `TokenFilter` lo mas cómodo es crear una subclase de `Analyzer`. Sin em-

bargo, antes de nada será conveniente considerar aquellas que nos proporciona Apache Lucene ya implementadas, de entre las que podemos destacar el StandardAnalyzer, que es utilizado en muchas aplicaciones.

6.1 Listado de Tokenizers, Char Filters y Token Filters disponibles en Lucene 9.3

En las siguientes tablas mostramos de un listado de Tokenizers, Char Filters y Token Filters disponibles en Lucene 9.3.0 así como la clase que se encuentran dentro de la biblioteca Lucene.

	Tokenizers en Lucene 9.3.0
uax29UrlEmail	org.apache.lucene.analysis.email.UAX29URLEmailTokenizerFactory
pathHierarchy	org.apache.lucene.analysis.path.PathHierarchyTokenizerFactory
wikipedia	org.apache.lucene.analysis.wikipedia.WikipediaTokenizerFactory
nGram	org.apache.lucene.analysis.ngram.NGramTokenizerFactory
edgeNGram	org.apache.lucene.analysis.ngram.EdgeNGramTokenizerFactory
thai	org.apache.lucene.analysis.th.ThaiTokenizerFactory
pattern	org.apache.lucene.analysis.pattern.PatternTokenizerFactory
simplePatternSplit	org.apache.lucene.analysis.pattern.SimplePatternSplitTokenizerFactory
letter	org.apache.lucene.analysis.core.LetterTokenizerFactory
keyword	org.apache.lucene.analysis.core.KeywordTokenizerFactory
standard	org.apache.lucene.analysis.standard.StandardTokenizerFactory
simplePattern	org.apache.lucene.analysis.pattern.SimplePatternTokenizerFactory
classic	org.apache.lucene.analysis.classic.ClassicTokenizerFactory
whitespace	org.apache.lucene.analysis.core.WhitespaceTokenizerFactory
	Char Filters en Lucene 9.3.0
patternReplace	org.apache.lucene.analysis.pattern.PatternReplaceCharFilterFactory
htmlStrip	org.apache.lucene.analysis.charfilter.HTMLStripCharFilterFactory
cjkWidth	org.apache.lucene.analysis.cjk.CJKWidthCharFilterFactory
persian	org.apache.lucene.analysis.fa.PersianCharFilterFactory
mapping	org.apache.lucene.analysis.charfilter.MappingCharFilterFactory

	Token Filters en Lucene 9.3.0 (página 1)
swedishMinimalStem	org.apache.lucene.analysis.sv.SwedishMinimalStemFilterFactory
elision	org.apache.lucene.analysis.util.ElisionFilterFactory
cjkWidth	org.apache.lucene.analysis.cjk.CJKWidthFilterFactory
tokenOffsetPayload	org.apache.lucene.analysis.payloads.TokenOffsetPayloadTokenFilter
stemmerOverride	org.apache.lucene.analysis.miscellaneous.StemmerOverrideFilterFactory
delimitedPayload	org.apache.lucene.analysis.payloads.DelimitedPayloadTokenFilter
hindiStem	org.apache.lucene.analysis.hi.HindiStemFilterFactory
flattenGraph	org.apache.lucene.analysis.core.FlattenGraphFilterFactory
commonGrams	org.apache.lucene.analysis.commongrams.CommonGramsFilterFactory
snowballPorter	org.apache.lucene.analysis.snowball.SnowballPorterFilterFactory
spanishLightStem	org.apache.lucene.analysis.es.SpanishLightStemFilterFactory
scandinavianFolding	org.apache.lucene.analysis.miscellaneous.ScandinavianFoldingFilterFactory
minHash	org.apache.lucene.analysis.minhash.MinHashFilterFactory
germanNormalization	org.apache.lucene.analysis.de.GermanNormalizationFilterFactory
hungarianLightStem	org.apache.lucene.analysis.hu.HungarianLightStemFilterFactory
apostrophe	org.apache.lucene.analysis.tr.ApostropheFilterFactory
lowercase	org.apache.lucene.analysis.core.LowerCaseFilterFactory
brazilianStem	org.apache.lucene.analysis.br.BrazilianStemFilterFactory
trim	org.apache.lucene.analysis.miscellaneous.TrimFilterFactory
length	org.apache.lucene.analysis.miscellaneous.LengthFilterFactory
delimitedTermFrequency	org.apache.lucene.analysis.miscellaneous.DelimitedTermFrequencyFilterFactory
limitTokenPosition	org.apache.lucene.analysis.miscellaneous.LimitTokenPositionFilterFactory
greekLowercase	org.apache.lucene.analysis.el.GreekLowerCaseFilterFactory
hyphenationCompoundWord	org.apache.lucene.analysis.compound.HyphenationCompoundWordFilterFactory
czechStem	org.apache.lucene.analysis.cz.CzechStemFilterFactory
keywordRepeat	org.apache.lucene.analysis.miscellaneous.KeywordRepeatFilterFactory
bengaliStem	org.apache.lucene.analysis.bn.BengaliStemFilterFactory
uppercase	org.apache.lucene.analysis.core.UpperCaseFilterFactory
portugueseMinimalStem	org.apache.lucene.analysis.pt.PortugueseMinimalStemFilterFactory
hunspellStem	org.apache.lucene.analysis.hunspell.HunspellStemFilterFactory
fixedShingle	org.apache.lucene.analysis.shingle.FixedShingleFilterFactory
dateRecognizer	org.apache.lucene.analysis.miscellaneous.DateRecognizerFilterFactory
hindiNormalization	org.apache.lucene.analysis.hi.HindiNormalizationFilterFactory
keepWord	org.apache.lucene.analysis.miscellaneous.KeepWordFilterFactory
delimitedBoost	org.apache.lucene.analysis.boost.DelimitedBoostTokenFilterFactory
frenchLightStem	org.apache.lucene.analysis.fr.FrenchLightStemFilterFactory
indicNormalization	org.apache.lucene.analysis.in.IndicNormalizationFilterFactory

	Token Filters en Lucene 9.3.0 (página 2)
commonGramsQuery	org.apache.lucene.analysis.commongrams.CommonGramsQueryFilterFactory
greekStem	org.apache.lucene.analysis.el.GreekStemFilterFactory
portugueseStem	org.apache.lucene.analysis.pt.PortugueseStemFilterFactory
nGram	org.apache.lucene.analysis.ngram.NGramFilterFactory
galicianStem	org.apache.lucene.analysis.gl.GalicianStemFilterFactory
limitTokenCount	org.apache.lucene.analysis.miscellaneous.LimitTokenCountFilterFactory
germanStem	org.apache.lucene.analysis.de.GermanStemFilterFactory
codepointCount	org.apache.lucene.analysis.miscellaneous.CodepointCountFilterFactory
spanishMinimalStem	org.apache.lucene.analysis.es.SpanishMinimalStemFilterFactory
germanLightStem	org.apache.lucene.analysis.de.GermanLightStemFilterFactory
limitTokenOffset	org.apache.lucene.analysis.miscellaneous.LimitTokenOffsetFilterFactory
swedishLightStem	org.apache.lucene.analysis.sv.SwedishLightStemFilterFactory
germanMinimalStem	org.apache.lucene.analysis.de.GermanMinimalStemFilterFactory
classic	org.apache.lucene.analysis.classic.ClassicFilterFactory
galicianMinimalStem	org.apache.lucene.analysis.gl.GalicianMinimalStemFilterFactory
keywordMarker	org.apache.lucene.analysis.miscellaneous.KeywordMarkerFilterFactory
englishMinimalStem	org.apache.lucene.analysis.en.EnglishMinimalStemFilterFactory
patternTyping	org.apache.lucene.analysis.pattern.PatternTypingFilterFactory
teluguStem	org.apache.lucene.analysis.te.TeluguStemFilterFactory
norwegianMinimalStem	org.apache.lucene.analysis.no.NorwegianMinimalStemFilterFactory
persianNormalization	org.apache.lucene.analysis.fa.PersianNormalizationFilterFactory
hyphenatedWords	org.apache.lucene.analysis.miscellaneous.HyphenatedWordsFilterFactory
spanishPluralStem	org.apache.lucene.analysis.es.SpanishPluralStemFilterFactory
teluguNormalization	org.apache.lucene.analysis.te.TeluguNormalizationFilterFactory
wordDelimiter	org.apache.lucene.analysis.miscellaneous.WordDelimiterFilterFactory
typeAsPayload	org.apache.lucene.analysis.payloads.TypeAsPayloadTokenFilterFactory
shingle	org.apache.lucene.analysis.shingle.ShingleFilterFactory
persianStem	org.apache.lucene.analysis.fa.PersianStemFilterFactory
italianLightStem	org.apache.lucene.analysis.it.ItalianLightStemFilterFactory
kStem	org.apache.lucene.analysis.en.KStemFilterFactory
wordDelimiterGraph	org.apache.lucene.analysis.miscellaneous.WordDelimiterGraphFilterFactory
decimalDigit	org.apache.lucene.analysis.core.DecimalDigitFilterFactory
soraniNormalization	org.apache.lucene.analysis.ckb.SoraniNormalizationFilterFactory
protectedTerm	org.apache.lucene.analysis.miscellaneous.ProtectedTermFilterFactory
type	org.apache.lucene.analysis.core.TypeTokenFilterFactory
synonym	org.apache.lucene.analysis.synonym.SynonymFilterFactory
porterStem	org.apache.lucene.analysis.en.PorterStemFilterFactory

	Token Filters en Lucene 9.3.0 (página 3)
patternReplace	org.apache.lucene.analysis.pattern.PatternReplaceFilterFactory
norwegianLightStem	org.apache.lucene.analysis.no.NorwegianLightStemFilterFactory
patternCaptureGroup	org.apache.lucene.analysis.pattern.PatternCaptureGroupFilterFactory
arabicStem	org.apache.lucene.analysis.ar.ArabicStemFilterFactory
russianLightStem	org.apache.lucene.analysis.ru.RussianLightStemFilterFactory
englishPossessive	org.apache.lucene.analysis.en.EnglishPossessiveFilterFactory
cjkBigram	org.apache.lucene.analysis.cjk.CJKBigramFilterFactory
numericPayload	org.apache.lucene.analysis.payloads.NumericPayloadTokenFilterFactory
fixBrokenOffsets	org.apache.lucene.analysis.miscellaneous.FixBrokenOffsetsFilterFactory
serbianNormalization	org.apache.lucene.analysis.sr.SerbianNormalizationFilterFactory
edgeNGram	org.apache.lucene.analysis.ngram.EdgeNGramFilterFactory
bulgarianStem	org.apache.lucene.analysis.bg.BulgarianStemFilterFactory
norwegianNormalization	org.apache.lucene.analysis.no.NorwegianNormalizationFilterFactory
truncate	org.apache.lucene.analysis.miscellaneous.TruncateTokenFilterFactory
asciiFolding	org.apache.lucene.analysis.miscellaneous.ASCIIFoldingFilterFactory
synonymGraph	org.apache.lucene.analysis.synonym.SynonymGraphFilterFactory
dropIfFlagged	org.apache.lucene.analysis.miscellaneous.DropIfFlaggedFilterFactory
portugueseLightStem	org.apache.lucene.analysis.pt.PortugueseLightStemFilterFactory
removeDuplicates	org.apache.lucene.analysis.miscellaneous.RemoveDuplicatesTokenFilterFactory
latvianStem	org.apache.lucene.analysis.lv.LatvianStemFilterFactory
scandinavianNormalization	org.apache.lucene.analysis.miscellaneous.ScandinavianNormalizationFilterFactory
soraniStem	org.apache.lucene.analysis.ckb.SoraniStemFilterFactory
reverseString	org.apache.lucene.analysis.reverse.ReverseStringFilterFactory
arabicNormalization	org.apache.lucene.analysis.ar.ArabicNormalizationFilterFactory
dictionaryCompoundWord	org.apache.lucene.analysis.compound.DictionaryCompoundWordTokenFilterFactory
bengaliNormalization	org.apache.lucene.analysis.bn.BengaliNormalizationFilterFactory
finnishLightStem	org.apache.lucene.analysis.fi.FinnishLightStemFilterFactory
typeAsSynonym	org.apache.lucene.analysis.miscellaneous.TypeAsSynonymFilterFactory
fingerprint	org.apache.lucene.analysis.miscellaneous.FingerprintFilterFactory
concatenateGraph	org.apache.lucene.analysis.miscellaneous.ConcatenateGraphFilterFactory
stop	org.apache.lucene.analysis.core.StopFilterFactory
irishLowercase	org.apache.lucene.analysis.ga.IrishLowerCaseFilterFactory
indonesianStem	org.apache.lucene.analysis.id.IndonesianStemFilterFactory
frenchMinimalStem	org.apache.lucene.analysis.fr.FrenchMinimalStemFilterFactory
capitalization	org.apache.lucene.analysis.miscellaneous.CapitalizationFilterFactory
turkishLowercase	org.apache.lucene.analysis.tr.TurkishLowerCaseFilterFactory

7 Visitando un Analyzer

Pasamos a ver con más detalle el proceso de análisis y como poder recoger más información sobre los tokens. Para ello, diseñamos una función que recibe como parámetro de entrada el tipo de analizador a utilizar así como la cadena a procesar y nos imprime la lista de tokens, con información sobre las posiciones (offset) de los mismos en la cadena.

En la línea 4 se declara `stream` como una variable del tipo `TokenStream`, capaz de enumerar la secuencia de tokens que devuelve el analizador. El método `addAttribute` de `TokenStream` chequea si la instancia de la clase se encuentra en el `TokenStream` y la devuelve. En caso contrario, se crea una nueva instancia y la añade al token stream. Dicha instancia será posteriormente utilizada por los consultores para conocer información sobre los distintos tokens.

Las líneas 8 a 12 nos muestran la secuencia de pasos que tenemos que realizar para explorar los resultados del analizador:

- `stream.reset()` Inicializa el stream, es necesario hacerlo antes de llamar a `incrementToken`. Se almacenan referencias locales a todos los atributos que se pueden acceder.
- `stream.incrementToken()`: Se posiciona en un atributo en cada llamada, devuelve falso cuando no hay mas atributos en el stream
- `stream.end()` Es necesario llamar a este método cuando se alcanza el final (`incrementToken` devuelve falso).

```
1 public static void tokenizeString(Analyzer analyzer, String
   string) {
2
3     try {
4         TokenStream stream = analyzer.tokenStream(null, new
           StreamReader(string));
5         OffsetAttribute offsetAtt = stream.addAttribute(
           OffsetAttribute.class);
6         CharTermAttribute cAtt= stream.addAttribute(
           CharTermAttribute.class);
```



```

7
8     stream.reset();
9     while (stream.incrementToken()) {
10         System.out.println( cAtt.toString()+"_:"+offsetAtt.
            startOffset()+" "+ offsetAtt.endOffset()+"");
11     }
12     stream.end();
13 } catch (IOException e) {
14     throw new RuntimeException(e);
15 }
16 }

```

Para ver el efecto un un analizador sobre el texto tendremos que utilizar el siguiente código:

```

1 Analyzer an = new WhitespaceAnalyzer( );
2 String text="Ejemplo de analizador + WhiteSpace , lucene
3 -7.1.0";
4 tokenizeString(an, text);

```

```

1 [Ejemplo : (0,7)]
2 [de : (11,13)]
3 [analizador : (14,24)]
4 [+ : (25,26)]
5 [WhiteSpace , : (27,38)]
6 [lucene -7.1.0 : (39,51)]

```

o si utilizamos un SimpleAnalyzer

```

1 [ejemplo : (0,7)]
2 [de : (11,13)]
3 [analizador : (14,24)]
4 [whitespace : (27,37)]
5 [lucene : (39,45)]

```

8 Creando nuestro propio analizador

Aunque Lucene dispone de múltiples Analyzers, a veces es necesario crear un analizador específico para una determinado tipo de texto de entrada. En general,

este analizador puede requerir la llamada a distintos `Tokenizer` y `TokenFilter`.

Para definir el comportamiento del analizador, la subclases deben definir sus `TokenStreamComponents` como salida del método `createComponents(String)`. Las distintas componentes serán utilizadas en las distintas llamadas del `tokenStream(String, Reader)`.

Veamos el siguiente ejemplo simple que viene con la documentación de Lucene:

```
1 Analyzer analyzer = new Analyzer() {  
2     @Override  
3     protected TokenStreamComponents createComponents(String  
4         fieldName) {  
5         Tokenizer source = new FooTokenizer(reader);  
6         TokenStream filter = new FooFilter(source);  
7         filter = new BarFilter(filter);  
8         return new TokenStreamComponents(source, filter);  
9     }  
10    @Override  
11    protected TokenStream normalize(TokenStream in) {  
12        // Assuming FooFilter is about normalization and BarFilter  
13        // is about  
14        // stemming, only FooFilter should be applied  
15        return new FooFilter(in);  
16    }  
17 };
```

Veamos un ejemplo concreto donde crearemos un `Analyzer` con la siguiente secuencia de pasos:

- Tokenizar utilizando un `StandardTokenizer`, sigue las reglas de segmentación para Unicode <http://unicode.org/reports/tr29/>
- Convertir a minúscula, `LowerCaseFilter`
- Eliminar números, pero no cuando estén en un string como H20, veremos como lo podemos implementar después.
- Eliminar palabras vacías, `StopFilter`
- Utilizar el stemmer de snowball, `SnowballFilter` para un determinado lenguaje

```

1
2 public Analyzer buildAnalyzer(final String language, final
   CharArraySet stopwords) {
3     return new Analyzer() {
4         @Override
5         protected Analyzer.TokenStreamComponents createComponents
           (String string) {
6             final Tokenizer source = new StandardTokenizer( );
7
8             TokenStream result = new StandardFilter( source);
9             result = new LowerCaseFilter( result);
10
11             result = new NumerosFilter(result); //Filtro para
               quitar numeros 2.35 3,235
12
13             result = new StopFilter( result, stopwords);
14             result = new SnowballFilter(result, language);
15             return new TokenStreamComponents(source, result);
16         }
17     }
18 }

```

donde el filtrado de número se hace reimplementando el método accept de la clase FilteringTokenFilter

```

1
2 static class NumerosFilter extends FilteringTokenFilter {
3
4     private final CharTermAttribute termAtt = addAttribute(
       CharTermAttribute.class);
5
6     public NumerosFilter(TokenStream in) {
7         super(in);
8     }
9
10    @Override
11    protected boolean accept() throws IOException {
12        String token = new String(termAtt.buffer(), 0, termAtt.
           length());

```

```
13     if (token.matches("[0-9,.]+")) {  
14         return false;  
15     }  
16     return true;  
17 }  
18 }
```

9 Stemmers

En este ejemplo de Analyzer hemos llamado a la clase que implementa el stemmer de Snowball, que aplica un conjunto de reglas para reducir un término a su raíz. Por ejemplo, si una palabra termina en *s* se entiende que es un plural y se elimina dicho carácter. En este caso, no se tiene en cuenta la palabra en sí, su significado, etc. Sólo se aplica la regla.

9.1 Snowball Stemmer

Snowball <http://snowballstem.org/> es un lenguaje de procesamiento de cadenas diseñado para crear un conjunto de reglas que aplicados sobre un término, lo reducen a su hipotética raíz. Obviamente, estas reglas dependen del lenguaje, y Snowball se puede considerar como un meta-algoritmo que, tomando como entrada el conjunto de reglas dado para un determinado lenguaje, al ejecutarse nos genera el módulo ANSI C o Java con el algoritmo que las ejecuta. Snowball ya incluye un conjunto de reglas para distintos lenguajes, entre ellos el español y dos para el inglés, entre ellos el algoritmo de Porter original.

Veamos algunas reglas para el castellano (obtenidas de <http://snowballstem.org/algorithms/spanish/stemmer.html>)

- Always do steps 0 and 1.

1. Step 0: Attached pronoun

Search for the longest among the following suffixes

me se sela selo selas selos la le lo las les los
nos

and delete it, if comes after one of

(a) iéndo ándo ár ér ír (b) ando iendo ar er ir

2. Step 1: Standard suffix removal

Search for the longest among the following suffixes, and perform the action indicated.

anza anzas ico ica icos icas ismo ismos able ables
ible ibles ista istas oso osa osos osas amiento
amientos imiento imientos

delete if in R2

....

Una demo del funcionamiento de los estemizadores de Snowball la podemos encontrar en <http://snowballstem.org/demo.html> donde podemos ver como las salidas obtenidas para entradas en distintos lenguajes. En la siguiente tabla mostramos un caso para el lenguaje español.

Origen	Estas enseñanza panza permitiéndoselos bonicos populismos
Salida	estas enseñ panz permit bonic popul
Origen	estudios estudiar estudiantes estudiamos estudias estudiarás
Salida	estudi estudi estudi estudi estudi estudi
Origen	detuximos astudiar peturiantes contabanza
Salida	detux astudi peturi contab

Como se puede ver en las últimas líneas de la tabla, las reglas se aplican a cualquier secuencia de caracteres, aunque no pertenezcan al vocabulario en español. Este tipo de situaciones los podemos en parte solventar si consideramos otro tipo de stemmer, como veremos en la siguiente sección.

9.2 Stemmer que utilizan un diccionario

Existe otro tipo de stemmers que en lugar de aplicar un conjunto de reglas lo que realizan es buscar una palabra en el diccionario, por lo que en teoría pueden producir mejores resultados que los algoritmos que se basan en la eliminación de sufijos. Una de las ventajas de estos algoritmos es que siempre tienen como

salida una palabra del vocabulario, reconociendo verbos irregular, identificar términos que son similares pero tienen significado distinto e incluso permiten proponer distintas alternativas cuando un término no está bien escrito como muestra la siguiente tabla:

Origen	Estas enseñanza panza permitiéndoselos bonicos
Salida	esto enseñanza panza permitir bonito
Origen	pecar pesqué pescaste
Salida	pescar pescar pescar
Origen	estudios estudiar estudiantes estudiamos
Salida	estudio estudiar estudiar estudiar
Origen	detuximos
Salida	NO reconocida, alternativas: detuvimos, desentumimos

Lucene incluye uno de estos stemmer, en concreto uno basado en el algoritmo Hunspell <http://hunspell.github.io/> utilizado por LibreOffice, Mozilla, Chrome, etc para el chequeo de palabras. Lo podemos encontrar en la clase `HunspellStemFilter` https://lucene.apache.org/core/7_1_0/analyzers-common/org/apache/lucene/analysis/hunspell/HunspellStemFilter.html.

Este stemmer hace uso de un ficheros, uno de ellos que contiene las palabras de un idioma junto con un código que contiene todas sus posibles afijos (prefijos y sufijos (.dic) y el otro que indica las formas en las que se de deben tratar dichos los afijos en el idioma (.aff). Estos ficheros los podemos obtener bien de LibreOffice o al instalarnos Hunspell en nuestro ordenador.

Un ejemplo del uso del mismo en Lucene es el siguiente.

```

1 ...
2 protected Analyzer.TokenStreamComponents createComponents(String
   string) {
3     InputStream affixStream= new FileInputStream("es_ANY.aff");
4     InputStream dictStream= new FileInputStream("es_ANY.dic");
5     Directory directorioTemp = FSDirectory.open(Paths.get("/temp")
   );
6     Dictionary dic= new Dictionary(directorioTemp, "temporalFile",
   affixStream, dictStream);

```

```
7
8     ....
9     TokenStream result = new StandardFilter( source);
10    ...
11    result = new StopFilter( result , stopwords);
12    result = new HunspellStemFilter( result , dic , true , true );
13    return new TokenStreamComponents( source , result );
14 };
```

10 Se pide

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.
2. Probar sobre un texto relativamente pequeño el efecto que tienen los siguientes tokenFilters: StandardFilter, LowerCaseFilter, StopFilter, SnowballFilter, ShingleFilter, EdgeNGramCommonFilter, NGramTokenFilter, CommonGramsFilter, SynonymFilter,
3. Diseñar un analizador propio. Se os da libertad para poder escoger el dominio de ejemplo que consideréis mas adecuado, justificar el comportamiento. Para ello, la forma mas inmediata puede ser utilizar el CustomAnalyzer (https://lucene.apache.org/core/7_1_0/analyzers-common/org/apache/lucene/analysis/custom/CustomAnalyzer.html). También, podemos crear nuestra propia clase Analyzer, como se indica en la documentación de Lucene, https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/package-summary.html

10.1 Fecha de entrega

- 11 de Octubre.