**PRACTICA 2 :**

**RECUPERACIÓN DE INFORMACIÓN**

Analizadores Lucene.



4º Grado en Ingeniería Informática

**Realizado por Miguel Acedo y Pablo Mariano Moreno**

Indice

[**Estudio estadístico**](#_f6ef94oprmc4) **3**

[**Uso de Token Filters**](#_e8f96b5upwsi) **4**

[StandardFilter](#_yypnpzv0c4d5) 4

[LowerCaseFilter](#_ko10fr368qcf) 4

[StopFilter](#_hgzla46dn757) 5

[SnowballFilter](#_ac9s804jjof2) 5

[ShingleFilter](#_i34v17lf191k) 6

[EdgeNGramCommonFilter](#_exflamirnb3d) 6

[NGramTokenFilter](#_hou5isxgjupv) 7

[CommonGramsFilter](#_aaslb747b1dj) 8

[SynonymFilter](#_gnpjne7mk088) 8

[**Analizador propio**](#_qgffnl7qbf4n) **9**

[**Analizador específico**](#_c2p1ee41gffs) **10**

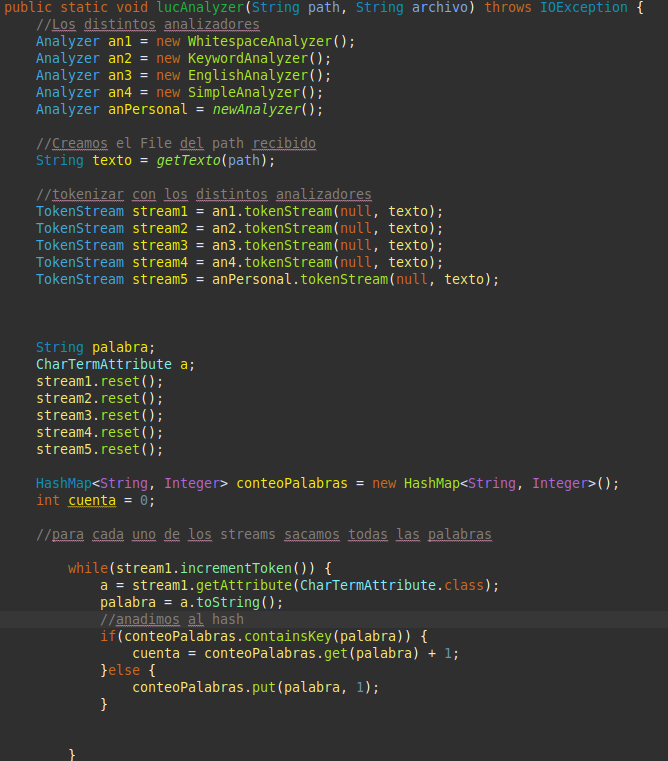
# Durante esta memoria expondremos los distintos usos de procesamiento y análisis de textos a los que nos da acceso Lucene, tanto a aquellos que ya forman parte de la librería como formas de poder implementarlos nosotros mismos de manera que cumplan nuestras necesidades.

# Estudio estadístico

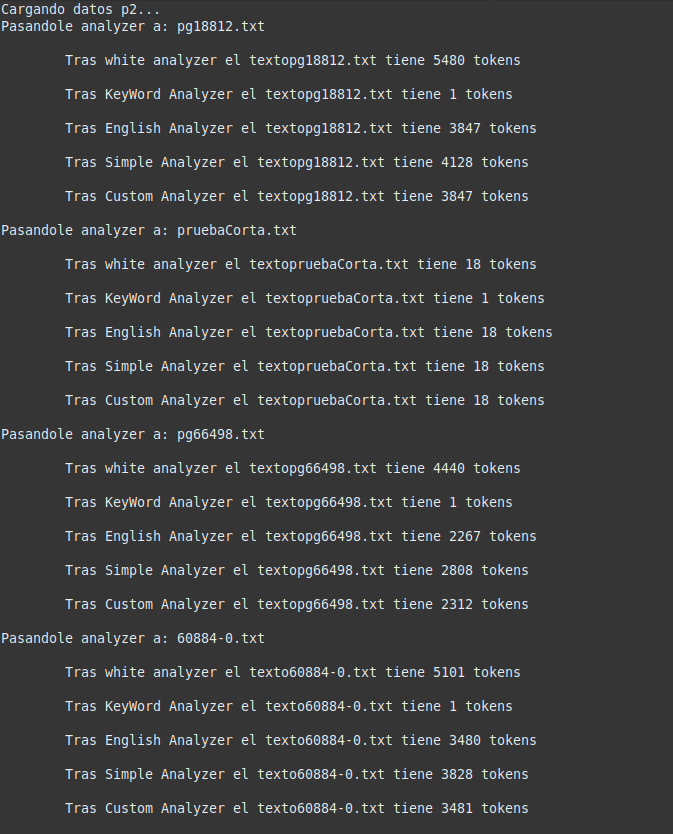
# En primer lugar veremos algunos de los principales analyzers que forman parte de la biblioteca de lucene, y como en función de que elijamos obtendremos una cantidad distinta de tokens.

Para ello se ha realizado un pequeño programa el cual leerá los documentos de un directorio dado como parámetro de entrada del programa, y para todos los elementos que aparezcan se obtendrá el texto interno (usando tika como veíamos en la práctica anterior).

Tras obtener el texto usando las funciones de la biblioteca de Lucene creamos e inicializamos los nuevos Analyzers y los TokenStreams. Tras ello usaremos un bucle para cada uno de los Streams de Tokens nuevos que nos irá dando uno a uno los distintos tokens obtenidos del texto inicial.



Una vez tenemos los Tokens los metemos un un hashmap y usamos algunas funciones de las implementadas en la práctica anterior para pasar los datos a un csv, que nos permitirá comparar las diferentes técnicas usadas sobra la misma palabra en función del analyzer elegido. Además nuestro programa mostrará por pantalla para cada uno de los textos el número de tokens resultantes. Como se muestra a continuación.



Con ello podemos observar cómo en función de la composición de cada uno de los analyzers tendremos una cantidad muy variable de tokens, desde el menos restrictivo que es el white space analyzer, el cual hace un procesamiento simple separando simplemente los elementos por espacios. Llegando al más restrictivo como es el KeyWord, que deja el texto intacto en un token, muy útil para aquellos caso en los que necesitemos mantener el texto tal y como lo obtenemos.

# Uso de Token Filters

Estructura básica:

Analyzer anStandardFilter = new Analyzer() {

@Override

protected TokenStreamComponents createComponents(String fieldName) {

Tokenizer source = new StandardTokenizer();

TokenStream result = new StandardFilter(source);

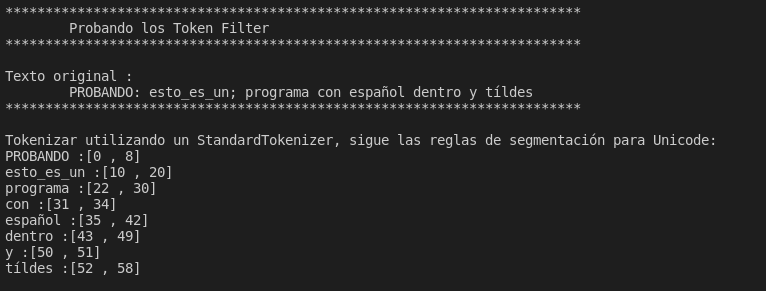
return new TokenStreamComponents(source, result);

}

};

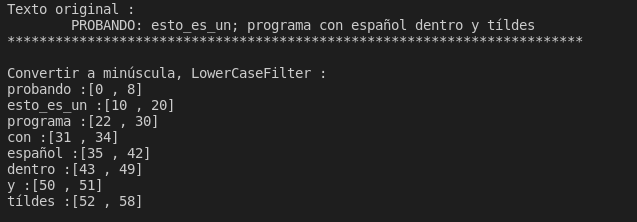
tokenizeString(anStandardFilter, texto);

## StandardFilter



sigue las reglas de segmentación para Unicode http://unicode.org/reports/tr29/

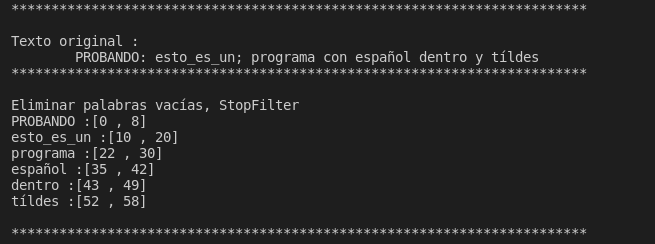
## LowerCaseFilter



Todo a minuscula

## StopFilter

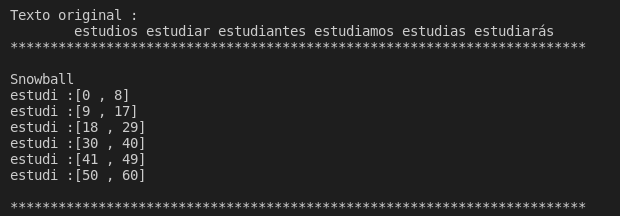
TokenStream result = new StopFilter(source, SpanishAnalyzer.getDefaultStopSet());



Elimina las palabras vacias

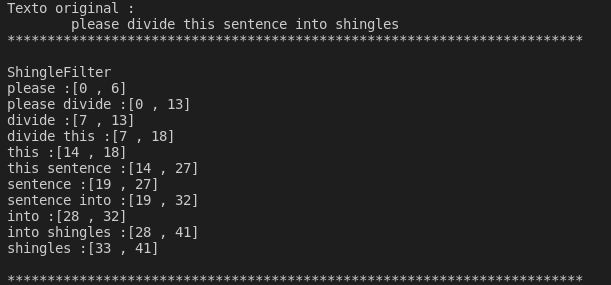
## SnowballFilter

TokenStream result = new SnowballFilter(source, new SpanishStemmer());



Se queda con la raiz

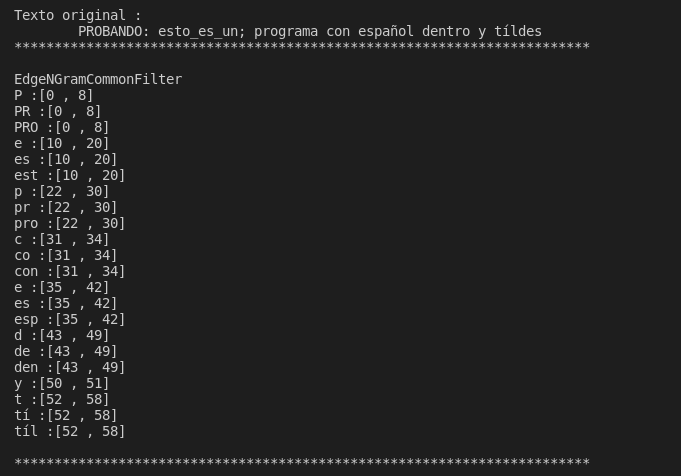
## ShingleFilter



Encadena los tokens agregandolos los pares

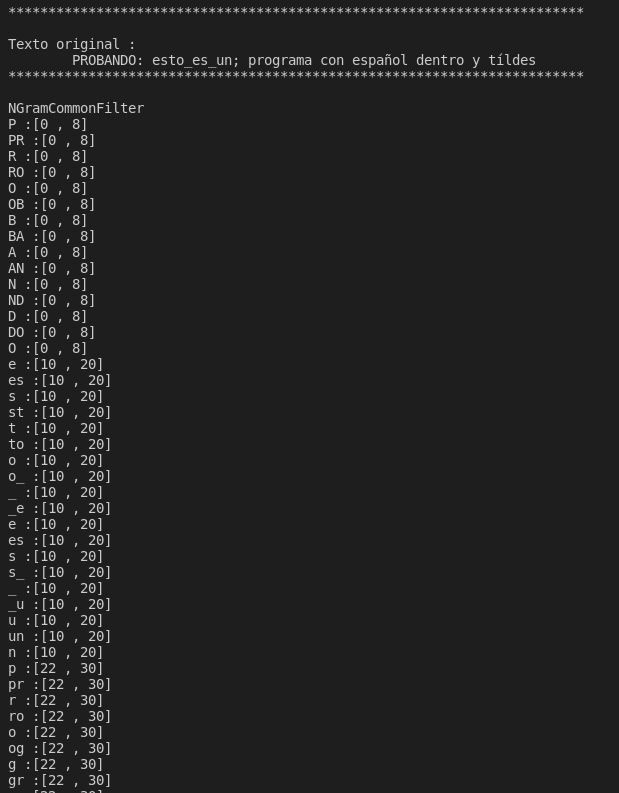
## EdgeNGramCommonFilter

TokenStream result = new EdgeNGramTokenFilter(source, 1, 3);



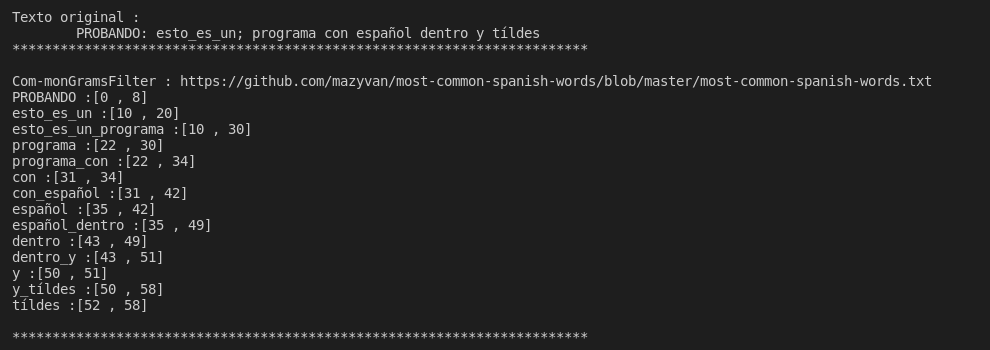
Separa desde este caso el primer carácter al tercer caracter

## NGramTokenFilter



Combinación de los dos anteriores

## CommonGramsFilter



Usando un diccionario sacado de : <https://github.com/mazyvan/most-common-spanish-words/blob/master/most-common-spanish-words.txt>

Con las palabras más comunes del español

## SynonymFilter

SynonymMap.Builder builder = new SynonymMap.Builder(true);

CharsRef jose = new CharsRef("Jose");

CharsRef pepe = new CharsRef("Pepe");

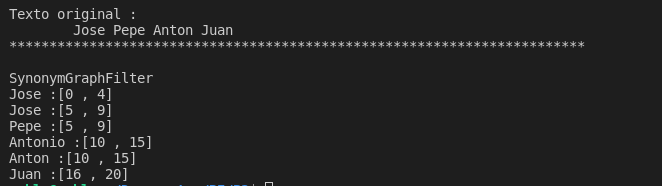
CharsRef antonio = new CharsRef("Antonio");

CharsRef anton = new CharsRef("Anton");

builder.add(pepe, jose, true);

builder.add(anton, antonio, true);

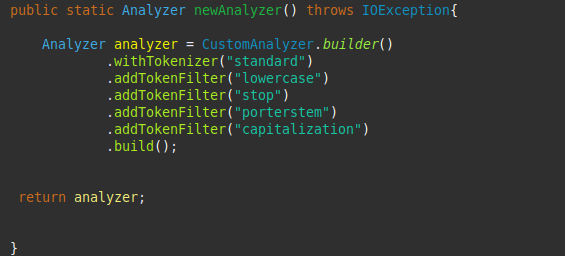
SynonymMap map = builder.build();



Cuando aparece Pepe se guarda Pepe y a la vez su sinónimo Jose, al igual con Anton, Antonio no aparece en el texto pero cuando el token es Anton se guarda también Antonio

# Analizador propio

Una de las opciones interesantes que tenemos con Lucene es que dependiendo de nuestras necesidades podremos usar toda su potencia moldeada para nuestro problema concreto. Una de las cosas que podemos moldear es un analyzer propio. Hay varias maneras de realizarlo, una puede ser una clase que extienda la clase analyzer de Lucene, en nuestro caso hemos optado por una función que devuelve un analyzer personal creado con CustomAnalyzer.builder(). Esto nos da la opcion a crear un analyzer y pasarle por simples parametros el tipo de de tokenizer y token filters que queremos en este caso.



El uso de la implementación es muy fácil, llamaremos a la función en vez de realizar la inicialización con new básica que hacíamos en los casos iniciales.

# Analizador específico

De manera análoga al punto anterior tenemos opciones gracias a Lucene de construir un TokenFilter que se acomode a nuestras necesidades. En este caso usamos extend TokenFilter para crear una subclase heredada de esta.

Como se indicaba en la práctica, este tokenfilter quitara las palabras de una longitud menor a 4, y en caso de ser más largas nos devolverá los cuatro últimos términos.

A continuación podemos ver la implementación del mismo.

public class CuatroFilter extends TokenFilter {

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

/\*\* Sole constructor \*/

public CuatroFilter(TokenStream in) {

super(in);

}

@Override

public final boolean incrementToken() throws IOException {

if (input.incrementToken()) {

if (termAtt.length() <= 3) {

termAtt.setEmpty();

termAtt.resizeBuffer(0);

termAtt.setLength(0);

// return false;

} else {

char[] t = termAtt.buffer();

t[0] = t[t.length - 1];

t[1] = t[t.length - 2];

t[2] = t[t.length - 3];

t[3] = t[t.length - 4];

termAtt.resizeBuffer(4);

}

return true;

} else

return false;

}

}