Preprocesado de documentos

Parte II. Análisis del texto

October 9, 2025

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.

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 (`https://lucene.apache.org/`) para dicho proceso.

Las aplicaciones que basan sus funciones de búsqueda en Lucene pueden necesitar trabajar con documentos en varios formatos: HTML, XML, PDF, Word, por nombrar solo algunos. Lucene no se preocupa por el análisis de estos y otros formatos de documento, y es responsabilidad de la aplicación que utiliza Lucene usar un analizador adecuado para convertir el formato original a texto sin formato antes de pasarlo a Lucene, por ejemplo Tika.

Se recomienda consultar el javadoc de la versión de Lucene que se utilice (a día de hoy está la versión 10.3.0)

`https://lucene.apache.org/core/10_3_0/index.html`

en especial el paquete dedicado al análisis de textos, que se encarga de convertir el texto en tokens

`https://lucene.apache.org/core/10_3_0/core/org/apache/lucene/analysis/package-summary.html`

también es interesante este enlace

`https://lucene.apache.org/core/10_3_0/demo/index.html`

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 10.3.0. 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 para mejorar la calidad de la búsquedas.

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

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

- Indexación escalable y de alto rendimiento
  - Más de 800 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
  - Consultas 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, entre ellos la búsqueda vectorial.

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 eliminado 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/10_3_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,
  - ...

La lista de todas las clases que se pueden ver involucradas en un proceso de análisis la podemos encontrar en `https://lucene.apache.org/core/10_3_0/analysis/common/allclasses-index.html`

Finalmente, indicar que Lucene 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 StandadAnalyzer 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 construidos, 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 queremos ver con más detalle el resultado del mismo. Será de utilidad para saber exactamente qué tokens serán indexados finalmente.

5.1 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).

Un **Analyzer** será el componente encargado de construir un transformar un texto bruto en una secuencia de tokens, TokenStream, que pueda ser digerido por los procesos de indexación como de búsqueda. No se encarga de procesar el texto bruto, sino que combina normalmente tres etapas: primero un CharFilter que permite transformar el texto (por ejemplo borrando caracteres). El texto resultante es pasado a un Tokenizer, que divide el texto en unidades básicas (como palabras, números o símbolos); después una serie de TokenFilters, que aplican transformaciones sobre esos tokens (por ejemplo, convertir todo a minúsculas, eliminar palabras vacías, aplicar stemming o normalizar acentos); y finalmente entrega un TokenStream, que es la representación estandarizada y procesada del texto. De esta manera, el analyzer garantiza que tanto en la indexación como en la búsqueda se manejen las mismas reglas de normalización, permitiendo obtener resultados

más precisos y consistentes en las consultas.

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.

Si queremos utilizar una combinación concreta de CharFilter, Tokenizer y TokenFilter lo mas cómodo es crear una subclase de Analyzer. Sin embargo, antes de nada será conveniente considerar aquellas que nos proporciona Apache Lucene ya implementadas (ver Sección 4.1), de entre las que podemos destacar el StandardAnalyzer, que es utilizado en muchas aplicaciones.

Seleccionar el analizador correcto es crucial para la calidad de la búsqueda y también puede afectar la indexación y el rendimiento de la búsqueda. El analizador adecuado para nuestra aplicación dependerá del aspecto del texto de entrada y del problema que intente resolver. Siempre debemos tener cuidado con el análisis utilizado ya que tanto no realizar análisis como si este es excesivo puede perjudicar el rendimiento de la aplicación.

5.1.1 Ejemplo: Utilizar herramienta Luke

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 pluging Analyzer para ver que tokens son obtenidos por cada analizador.

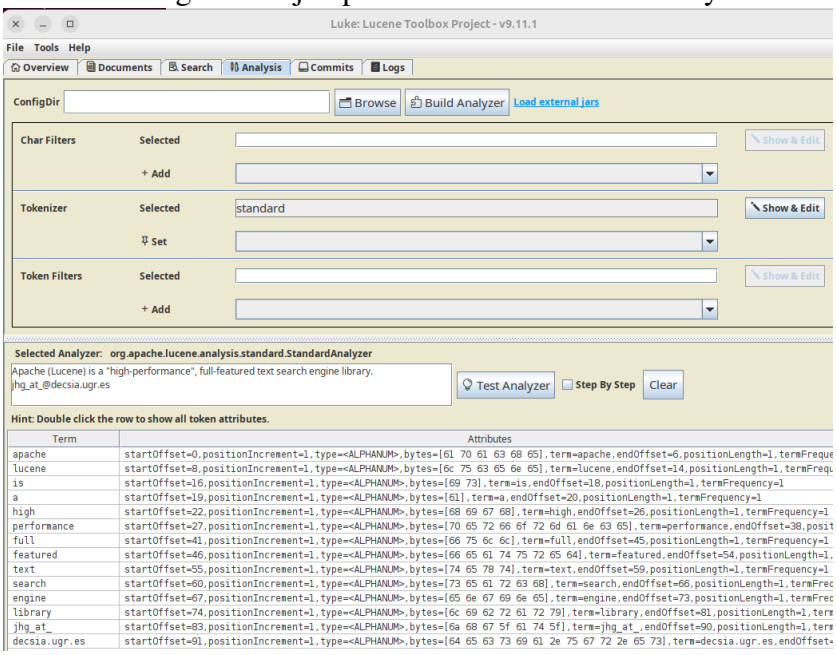
La últimas versiones de Lucene contienen integrado a Luke (lo podemos encontrar en el subdirectorio `luke` de la versión de Lucene que descarguemos) y ejecutar el fichero .jar.

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 los componentes del analizador que nos interesa dentro de cada uno de los desplegables 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.

Sólo a nives ilustrivo podremos crear nuestro "propio" analizador utilizando una secuencia de CharFilters, Tokenizers y Token Filters con Luke:

Para ello, comenzaremos añadiendo un `htmlStrip` como CharFilter. Este filtro podrá quitar las etiquetas html de la entrada. Además, como Token Filter añadiremos el `spanishLightStem` para realizar un stemming en castellano

Figure 1: Ejemplo de uso del StandardAnalyzer con Luke.



sobre el texto de entrada. Pulsamos el boton build analyzer y también indicamos que nos muestre el análisis step-by-step y vemos que nos debe salir algo parecido a la Figura 2.

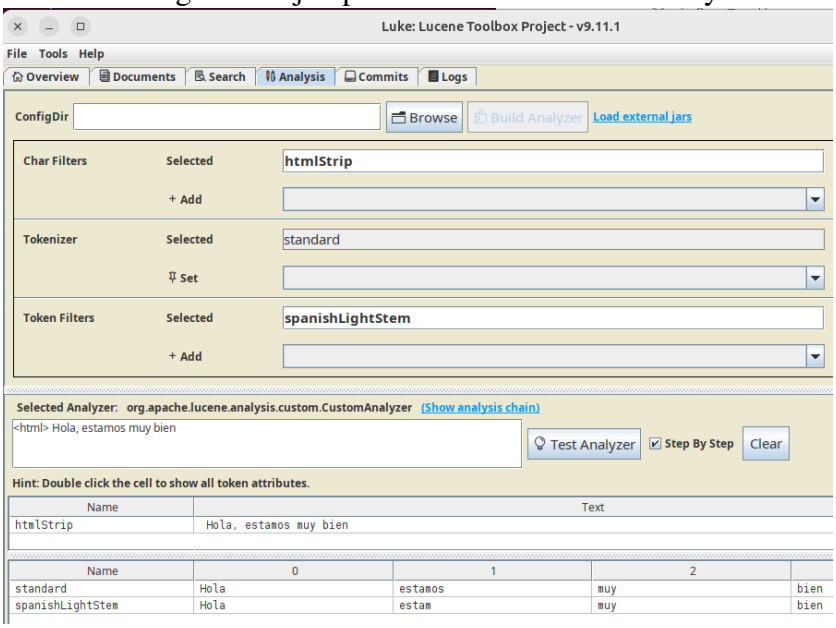
Otro ejemplo lo tenemos si queremos utilizar un CharFilter tipo `patternReplace`. Pare este último debemos editar los parámetros. Por ejemplo, podemos decir que reemplace las `a` por `x` en el texto de entrada. Para ello, debemos introducir en el nombre del primer parámetro `pattern` y el valor `a` y en el segundo parámetro `replacement` con valor `x`.

6 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, in-

Figure 2: Ejemplo de uso del StandardAnalyzer con Luke.



cluyendo las librerías de Lucene (En el ejemplo trabajamos con la versión 9.3.0 de Lucene), por tanto, tenemos que incluir en el classpath oen las dependencias del proyecto los siguientes paquetes:

- `lucene-core-x.x.x.jar`
- `lucene-analysis-common-x.x.x.jar`, que se encuentran dentro del directorio

  que se encuentra en el directorio `lucene-x.x.x/modules` de Lucene

El objetivo de la aplicación será listar por consola todos y cada uno de los tokens que se obtienen al aplicar un `WhitespaceAnalyzer`, junto al tipo del mismo y su desplazamiento (offSet) para lo que utilizaremos en el siguiente código:

```
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
9
10 import org.apache.lucene.analysis.tokenattributes.
       CharTermAttribute;
11 import org.apache.lucene.analysis.tokenattributes.
       OffsetAttribute;
12 import org.apache.lucene.analysis.tokenattributes.TypeAttribute;
13
14 public class Analizador {
15
16    public static void main(String[] args) throws IOException {
17      Analyzer an = new WhitespaceAnalyzer();
18      String cadena = "Ejemplo  23  de analizador + WhiteSpace ,
            lucene-10.3.0";
19
20      TokenStream stream = an.tokenStream(null, cadena);
21
22      CharTermAttribute cAtt = stream.addAttribute(
            CharTermAttribute.class);
23      TypeAttribute typeAttr = stream.addAttribute(TypeAttribute.
            class);
24
25      stream.reset(); // Pone el stream al principio (necesario)
26      while (stream.incrementToken()) {
27        System.out.println(cAtt.toString() + " - " + typeAttr.
              type());
28      }
29      stream.end();
30      stream.close();
31    }
32
33 }
```

Así, en la línea 17 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 20 de código donde, como resultado de proceso obten-

emos 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 a través de la API de `TokenStream` (ver documentación de la API). En nuestro caso, sólo recuperaremos el término en sí (`CharTermAttribute`) y el tipo (línea 22 y 23 del código). El método addAttribute de TokenStream chequea si la instancia de la clase se encuentra en el TokenStream devuelto por el analizador. En caso contrario, se crea una nueva instancia y la añade al token stream el atributo. Dicha instancia será posteriormente utilizada por los consultores para conocer información sobre los distintos tokens (línea 27).

Las líneas 25 a 30 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).

Utilizar otro analizador es fácil, por ejemplo, podemos modificar la línea 17 para utilizar un SimpleAnalyzer, StandardAnalyzer o un ShingleAnalyzerWrapper y el conjunto de tokens obtenido (y sus atributos) varían considerablemente.

Resultado WhitespaceAnalyzer

Separa por espacios en blanco

```
1 Ejemplo - word
2 23 - word
```

[1] `https://lucene.apache.org/core/10_3_0/core/org/apache/lucene/analysis/TokenStream.html`

```
3 de - word
4 analizador - word
5 + - word
6 WhiteSpace , - word
```

Resultado SimpleAnalyzer

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 - word
2  de - word
3  analizador - word
4  whitespace - word
5  lucene - word
```

Resultado StandardAnalyzer

Identifica palabras, números y acrónimos separando por espacios y signos de puntuación. También utiliza un conjunto de palabras vacías en Inglés.

```
1   ejemplo - <ALPHANUM>
2   23 - <NUM>
3   de - <ALPHANUM>
4   analizador - <ALPHANUM>
5   whitespace - <ALPHANUM>
6   lucene - <ALPHANUM>
7   10.3.0 - <NUM>
```

Resultado ShingleAnalyzerWrapper

Un ejemplo mas sofisticado lo tenemos si utilizamos el ShingleAnalyzerWrapper de Lucene. Es una herramienta muy útil cuando quieres generar n-grams de tokens (llamados shingles) a partir de un Analyzer existente (por defecto utiliza el

StandardAnalyzer y tamaños de n-gramas de 1 y 2). Esto es muy común para búsquedas de frases, sugerencias de búsqueda o para mejorar el scoring de coincidencias parciales. En su configuración por defecto la salida que nos daría es

```
ejemplo - <ALPHANUM>
ejemplo 23 - shingle
23 - <NUM>
23 de - shingle
de - <ALPHANUM>
de analizador - shingle
analizador - <ALPHANUM>
analizador whitespace - shingle
whitespace - <ALPHANUM>
whitespace lucene - shingle
lucene - <ALPHANUM>
lucene 10.3.0 - shingle
10.3.0 - <NUM>
```

7 Listado de Tokenizers, Char Filters y Token Filters disponibles en Lucene 9.3.0

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

7.0.1 CharFilters

	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

7.0.2 Tokenizers

	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

7.0.3 Token Filters

	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.TokenOffsetPayloadTokenFilterF
stemmerOverride	org.apache.lucene.analysis.miscellaneous.StemmerOverrideFilterFacto
delimitedPayload	org.apache.lucene.analysis.payloads.DelimitedPayloadTokenFilterFact
hindiStem	org.apache.lucene.analysis.hi.HindiStemFilterFactory
flattenGraph	org.apache.lucene.analysis.core.FlattenGraphFilterFactory
commonGrams	org.apache.lucene.analysis.commongrams.CommonGramsFilterFactor
snowballPorter	org.apache.lucene.analysis.snowball.SnowballPorterFilterFactory
spanishLightStem	org.apache.lucene.analysis.es.SpanishLightStemFilterFactory
scandinavianFolding	org.apache.lucene.analysis.miscellaneous.ScandinavianFoldingFilterF
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.DelimitedTermFrequencyTo
limitTokenPosition	org.apache.lucene.analysis.miscellaneous.LimitTokenPositionFilterFac
greekLowercase	org.apache.lucene.analysis.el.GreekLowerCaseFilterFactory
hyphenationCompoundWord	org.apache.lucene.analysis.compound.HyphenationCompoundWordTo
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.CommonGramsQueryFilterF
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.LimitTokenCountFilterFactor
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.LimitTokenOffsetFilterFactor
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.HyphenatedWordsFilterFacto
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.WordDelimiterGraphFilterFa
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.PatternCaptureGroupFilterFactor
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.NumericPayloadTokenFilterFac
fixBrokenOffsets	org.apache.lucene.analysis.miscellaneous.FixBrokenOffsetsFilterFac
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.TruncateTokenFilterFacto
asciiFolding	org.apache.lucene.analysis.miscellaneous.ASCIIFoldingFilterFactor
synonymGraph	org.apache.lucene.analysis.synonym.SynonymGraphFilterFactory
dropIfFlagged	org.apache.lucene.analysis.miscellaneous.DropIfFlaggedFilterFacto
portugueseLightStem	org.apache.lucene.analysis.pt.PortugueseLightStemFilterFactory
removeDuplicates	org.apache.lucene.analysis.miscellaneous.RemoveDuplicatesTokenF
latvianStem	org.apache.lucene.analysis.lv.LatvianStemFilterFactory
scandinavianNormalization	org.apache.lucene.analysis.miscellaneous.ScandinavianNormalizatio
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.DictionaryCompoundWordTo
bengaliNormalization	org.apache.lucene.analysis.bn.BengaliNormalizationFilterFactory
finnishLightStem	org.apache.lucene.analysis.fi.FinnishLightStemFilterFactory
typeAsSynonym	org.apache.lucene.analysis.miscellaneous.TypeAsSynonymFilterFac
fingerprint	org.apache.lucene.analysis.miscellaneous.FingerprintFilterFactory
concatenateGraph	org.apache.lucene.analysis.miscellaneous.ConcatenateGraphFilterFa
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.CapitalizationFilterFactor
turkishLowercase	org.apache.lucene.analysis.tr.TurkishLowerCaseFilterFactory

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 CharFilter, 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[2]:

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

Muchas veces la forma mas inmediata puede ser utilizar el CustomAnalyzer (`https://lucene.apache.org/core/10_3_0/analysis/common/org/apache/lucene/analysis/custom/CustomAnalyzer.html`). También, podemos crear nuestro propia clase Analyzer, como se indica en la documentación de Lucene, `https://lucene.apache.org/core/9_11_0/`

[2] `https://lucene.apache.org/core/10_3_0/core/org/apache/lucene/analysis/Analyzer.html`

`core/org/apache/lucene/analysis/package-summary.html`

Ejemplo

Supongamos el caso en que queremos crear 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 para un determinado lenguaje
- Utilizar el stemmer de snowball, SnowballFilter para un determinado lenguaje

```
1
2  public class MultiLanguageAnalyzer   extends Analyzer  {
3
4
5     private final String language;
6     private final CharArraySet stopwords;
7
8     public MultiLanguageAnalyzer(String language) {
9        this.language = language.toLowerCase();
10       switch (language) {
11          case "spanish":
12                stopwords = SpanishAnalyzer.getDefaultStopSet();
13                break;
14          case "french":
15                stopwords = FrenchAnalyzer.getDefaultStopSet();
16                break;
17          case "english":
18          default:
19              stopwords = EnglishAnalyzer.getDefaultStopSet();
20              break;
21         }
```

```
22      }
23
24      @Override
25       protected TokenStreamComponents createComponents(String
            fieldName) {
26       Tokenizer source = new StandardTokenizer();
27
28       TokenStream filter = new LowerCaseFilter(source);
29       filter = new NumerosFilter(filter); //Filtro para quitar
            numeros 2.35 3,235
30
31       // Seleccionamos palabras vacias dependiendo del lenguaje
32       filter = new StopFilter(filter, stopwords);
33
34        // Seleccionamos  stemmer dependiendo del lenguaje
35       switch (language) {
36           case "spanish":
37             filter = new SnowballFilter(filter, new SpanishStemmer
                  ());
38             break;
39           case "french":
40             filter = new SnowballFilter(filter, new FrenchStemmer
                  ());
41             break;
42
43           case "english":
44           default:
45              filter = new SnowballFilter(filter, new
                   EnglishStemmer());
46              break;
47           }
48
49        return new TokenStreamComponents(source, filter);
50        }
51  }
```

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 utilizado el stemmer de Snowball, que aplica un conjunto de reglas para reducir un término a su raíz[3].

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 si, 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,

[3]En la versión de Lucene 10 ya viene incorporado su propio paquete de Snowball (lucene-analyzers-common), y no es necesario incluir otra dependencia snowball.

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 pare 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 bonico
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/10_3_0/analysis/common/org/apache/lucene/analysis/hunspell/package-summary.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 suficos (.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 LowerCaseFilter( source);
10   ...
11   result = new StopFilter( result, stopwords);
12   result = new HunspellStemFilter(result,dic,true,true);
13   return new TokenStreamComponents(source, result);
14 };
```

10 Query Suggestion: Ejemplo de uso de Analizadores

En muchos casos, nos limitamos a escoger un determinado analizador (o implementar uno propio), pero su efecto en la calidad de la recuperación de información es de gran importancia. Por ello, en esta práctica pretendemos que el estudiante comprenda su importancia cuando consideramos un ejemplo sencillo como es la sugerencia de consultas.

El autocompletado o sugerencia de consulta (Query Suggestion) es una funcionalidad muy útil para mejorar la experiencia de búsqueda al sugerir consultas basadas en términos parciales que un usuario escribe en un motor de búsqueda.

Lucene nos proporciona la distintas herramientas para poder realizar dicha tarea, lucene-suggest. En esencia, para implementar dicha funcionalidad debemos crearnos un índice que incluya las palabras clave (términos) o frases que pretendamos sugerir. Una vez creado, se utilizará este índice para recomendar las sugerencias a la consulta que está realizando el usuario.

En este proceso, el analizador utilizado a la hora de diseñar la consulta presenta un papel clave, ya que nos permite realizar sugerencias considerando sinónimos, palabras vacías, formas canónicas de los términos y cualquier otro filtro utilizado en el análisis y realizar las sugerencias teniendo en cuenta en lugar de las palabras en sí, los tokens internos que se obtienen.

En esta práctica se pretende profundizar en el estudio de los analizadores y su efecto en las recomendaciones. Para ello, se entrega un software base sobre el que realizar nuestras pruebas. (fichero `suggestion.java`) en la plataforma Prado, así como distintos ficheros con datos para entrenarlo).

Para usarlo, es necesario proporcionar al suggester el conjunto de ítems a recomendar, que es el conjunto completo de cadenas y pesos que se pueden sugerir. Los ítems pueden proceder de cualquier parte; normalmente se suelen utilizar los logs de consultas, dando un mayor peso a aquellas consultas que aparecen con más frecuencia. Si se consideran películas como ítems a recuperar, se puede utilizar todos los títulos de películas con un peso según su valoración o si se consideran libros podemos considerar la popularidad de ventas.

También deberemos proporcionar un analizador, que se utiliza para analizar cada ítem a recomendar. A nivel interno los elementos se indexan los tokens que nos devuelve el analizador y es lo que utilizamos para realizar los emparejamientos.

En el momento de la búsqueda, la consulta entrante es procesada por el mismo (u otro) analizador y se busca los emparejamientos en el índice siguiendo distintas estrategias que pasamos a considerar:

10.0.1 Recomendaciones basadas en prefijo

En este caso, a partir de un texto (la consulta sin completar del usuario) se buscan aquellas otras consultas en el sistema que comiencen con este texto. Internamente los elementos se almacenan en un autómata finito con pesos (FST, Finite State Transducer). FST permite hacer de forma rápida las búsquedas prefijo (elementos cuyo prefijo empareje con la consulta). De esto se encarga el `AnalyzingSuggester`

10.0.2 Recomendaciones que consideran distancias entre palabras

Es parecido al anterior, salvo que se consideran como emparejamientos válidos aquellos que están a una determinada distancia de edición. De esto se encarga el `FuzzySuggester`.

10.1 Recomendacciones Infijos

En este caso, no es necesario que los prefijos de la consulta emparejen con los prefijos de los datos almacenados (por ejemplo, consultas previas). De esto se encarga el `AnalyzingInfixSuggester`

En este caso, el conjunto de elementos a recomendar se guarda en un índice Lucene y las sugerencias que se realizan dependen del los tokens que emparejen con la consulta.

La salida está ordenado por el peso que se le da a cada entrada a la hora de construir el "suggester", como podría ser la popularidad.

10.1.1 Recomendar el siguiente término en una consulta

Este tipo de recomendaciones se centra en la recomendación de el siguiente término que con más probabilidad podemos esperar que aparezca en la consulta, más que el hecho de recomendar la consulta en sí.

Para ello, un `FreeTextSuggester` crea un **modelo de lenguaje** sencillo y la sugerencia que se realiza es el término que es más probable que aparezca en el lenguaje que hemos aprendido. La idea es similar a la que implementa Google es

estos casos (`https://googleblog.blogspot.com/2011/04/more-predictions-in-a`
`html`)

Para crear el modelo debemos entrenarlo con datos representativos de lo que sería el problema considerado.

11 Software entregado

En Prado podremos encontrar el fichero `sugestion.java` junto a un fichero con consultas sobre el lenguaje de programación Java que puede servirnos para ilustrar como funciona un sistema de recomendación de consultas.

Con respecto a la sugerencia del siguiente término, cada grupo deberá considerar ampliar el fichero de entrenamiento dado utilizando un libro de tamaño considerado.

12 Se pide

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

2. Diseñar un analizador propio. En este caso, asumiremos que trabajamos con la red social `X`, donde `@cadena` y `#cadena` serán considerados como tokens válidos. Además, en dicho analizador se permitirá que ademas de almacenar los emojis como tokens, se almacene también su significado en castellano. Por ejemplo, para `:-)` se incluirá la palabra `feliz`, o `:-)` se incluirá `triste` como sinónimo.

3. Realizar un estudio empírico del uso de los distintos analizadores para el problema de la sugerencia de consultas. En este caso, vamos a trabajar con los datos que utilizaremos para la mayoría de las prácticas. En concreto, nos referimos al conjunto de datos LOS ANGELES AIRBNB DATA (JUNE 2025) obtenidos de la plataforma Kaggle

`https://www.kaggle.com/datasets/wafaaalayoubi/los-angeles-airbnb-data`

Este conjunto de datos contiene información pública disponible en el proyecto Inside Airbnb[4] para la ciudad de Los Angeles, California. El conjunto de datos fue recogido el 17 de Junio de 2025 y proporciona una visión detallada de los hospedajes de Airbnb en ese momento.

El conjunto de datos viene en un fichero `listings.csv` (que también lo dejaremos en Prado). El fichero tiene un total de 79 columnas con un total de unos 45.000 recursos.

En concreto, estamos interesados en proporcionar sugerencias de búsqueda para dos campos concretos que se encuentran en la columnas `host_neighbourhood` y `amenities`. En este caso, estamos asumiendo que un usuario a la hora de realizar una búsqueda podrá indicar datos para estos campos y en cada momento se le sugerirán los elementos mas relevantes a la consulta que esté haciendo.

Se valorará especialmente la capacidad de realizar un análisis crítico sobre la importancia de los analizadores en el proceso de sugerencia de consultas. Específicamente, se debe considerar cómo el uso de diferentes analizadores afecta la calidad, precisión y relevancia de las sugerencias. Es fundamental que el análisis incluya:

- Comparación entre varios analizadores.
- Discusión sobre cómo cada analizador maneja aspectos como el stemming, lematización, generación de n-gramas, entre otros.
- Reflexión sobre la idoneidad de cada analizador en este contexto de búsqueda.

El análisis deberá ir más allá de una simple descripción técnica y enfocarse en evaluar el impacto práctico en la experiencia del usuario y la efectividad

[4]Inside Airbnb is an independent, non-commercial project that provides data and advocacy about Airbnb's impact on communities around the world. This dataset is sourced from the Inside Airbnb project. The data is used under the Creative Commons Attribution 4.0 International License.

del sistema de sugerencias.

12.1 Fecha de entrega

- 22 de Octubre.