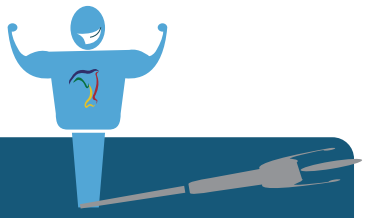
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

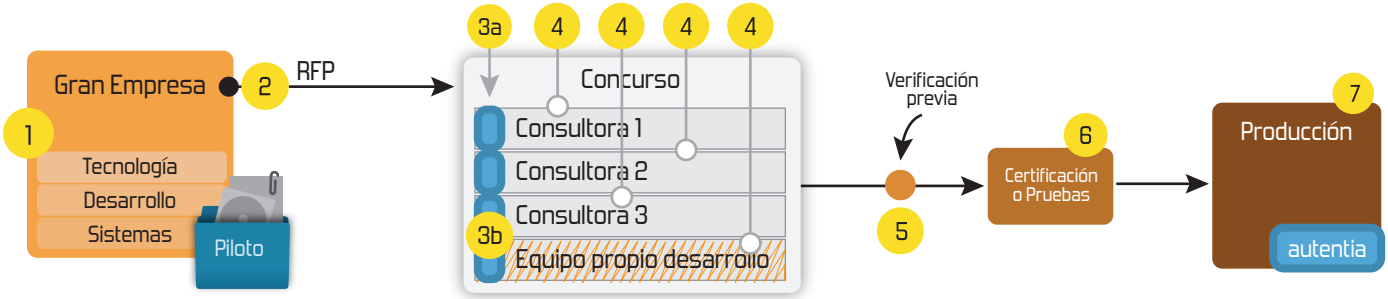
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Últimos tutoriales

2008-03-04
Introducción a JPivot

2008-03-03
Tablas dinámicas online

2008-02-29
Generación automática de gráficas en un web

2008-02-28
Manual de instalación de OpenCms 7

2008-02-28
Creación de un proyecto en SourceForge.net

2008-02-22
Lucene: Analyzers, stemming y búsqueda de documentos similares.

2008-02-22
Crear un logger utilizado a través de aspectos con Spring AOP.

2008-02-20
Primeros pasos con PostgreSQL en Debian

2008-02-17
Cómo realizar pruebas unitarias con Spring y JUnit4 utilizando Gienah

2008-02-15
Creación de una aplicación con Spring e Hibernate desde 0

Últimas ofertas de empleo

2008-02-28
Otras - Arquitecto / Aparejador - MADRID.

2008-02-27
T. Información - Analista / Programador - MADRID.

2008-02-26
Comercial - Ventas - ALICANTE.

2008-02-22
Atención a cliente - Call Center - MADRID.

2008-02-20
T. Información - Analista / Programador - MADRID.

Tutorial desarrollado por

Jose Manuel Sánchez Suárez

Consultor tecnológico de desarrollo de proyectos informáticos. Diseñador de Adictos Al Trabajo 2.0

Puedes encontrarme en Autentia

Somos expertos en Java/J2EE

Descargar este documento en formato PDF: luceneAnaLyzersStemmingMoreLikeThis.pdf

Fecha de creación del tutorial: 2008-02-22

Lucene: Analyzers, stemming y búsqueda de documentos similares.

0. Índice de contenidos.

1. Introducción

Lucene es un api para la recuperación de información, Information Retrieval (IR), distribuido bajo la Apache Software License.

Encaja perfectamente en el concepto de gestión documental (DMS) e incluso en la gestión de contenidos (CMS), puesto que un sistema de gestión documental requiere de la extracción del contenido de los documentos, la indexación de los mismos en un repositorio y la posibilidad de recuperarlos realizando búsquedas por su contenido textual.

No penséis en un sistema de gestión documental como en la mega-aplicación construida única y exclusivamente como "contenedor de documentos" para vuestra organización, cualquier aplicación tiene algo de gestión de documentos. Y, si tenemos en cuenta que buena parte del existo de esa gestión radicará en la capacidad de recuperar la información que se cataloga, después de leer este tutorial pensarás que Lucene debe formar parte de tu vida... :-D

Ya dimos, de la mano de Roberto Canales, los primeros pasos con Lucene en java, instalándolo, creando un índice, extrayendo el contenido de un pdf, indexándolo y recuperando la información del mismo.

En este tutorial vamos a ver cómo implementar un analizador semántico en nuestro idioma, potenciando la indexación y búsqueda, para terminar analizando la viabilidad de realizar búsquedas de documentos similares.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil Asus G1 (Core 2 Duo a 2.1 GHz, 2048 MB RAM, 120 GB HD).
- Sistema Operativo: GNU / Linux, Debian (unstable), Kernel 2.6.23, KDE 3.5
- JDK 1.5.0_14
- Eclipse Europa 3.3
- Lucene 2.2.0

3. Consideraciones previas.

Para la indexación y recuperación del contenido textual de los documentos que gestionamos nos bastaría con utilizar alguno de los analizadores que proporciona por defecto Lucene, pero si queremos potenciar las búsquedas de modo que no se produzca demasiado ruido en el resultado y para cumplir el objetivo de buscar documentos similares, tenemos que conseguir que los documentos pasen por un filtro lo más exhaustivo posible.

Podemos conseguirlo implementando los siguientes conceptos:

- **stopwords**: son una lista de palabras de uso frecuente que, tanto en la indexación como en la búsqueda, no se tienen en consideración, se omiten.
- **stemming**: es un método para obtener la raíz semántica de una palabra. Las palabras se reducen a su raíz o stem (tema), de modo que, si buscamos por "abandonados" encontrará "abandonados" pero también "abandonadas", "abandonamos", ... porque, en realidad, estamos buscando por "abandon".
- **modelo de espacio vectorial**: es un modelo algebraico utilizado para filtrar, indexar, recuperar y calcular la relevancia de la información. Representa los documentos con un lenguaje natural mediante el uso de vectores en un espacio lineal multidimensional. La relevancia de un documento frente a una búsqueda puede calcularse usando la diferencia de ángulos de cada uno de los documentos respecto del vector de busca, utilizando el producto escalar entre el vector de búsqueda.

A priori parece complejo, pero lo que cuesta es comprenderlo, vamos a ver cómo Lucene nos va a facilitar mucho la vida. Lo costoso que resulte implementarlo en tus desarrollos, eso solo lo sabes tú, aunque sino te haces una idea de ello... siempre nos puedes llamar para analizarlo y, en su caso, desarrollarlo.

A continuación, las dependencias que tendrá el proyecto en nuestro pom.xml, si no usáis maven, serán las librerías a importar manualmente:

view plain print ?

```
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.apache.lucene</groupId>
<artifactId>lucene-core</artifactId>
<version>2.3.1</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>org.apache.lucene</groupId>
<artifactId>lucene-snowball</artifactId>
<version>2.3.1</version>
</dependency>
<dependency>
<groupId>org.apache.lucene</groupId>
<artifactId>lucene-queries</artifactId>
<version>2.3.1</version>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.14</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>org.apache.poi</groupId>
<artifactId>poi</artifactId>
<version>3.0-FINAL</version>
</dependency>
```

4. SpanishAnalyzer.

Lucene provee varios analizadores por defecto, el StandardAnalyzer con un listado reducido de stopwords en inglés y podemos obtener una librería (lucene-analyzers) con analizadores en bastantes idiomas... casi todos menos en Castellano... :-(

Aún teniendo un analizador en el idioma requerido, la recomendación es que construyáis el vuestro propio para aumentar el listado de stopwords, si fuese necesario, y comprobar el algoritmo con el que se está realizando, si es que se realiza, el stemming.

Para la elaboración del listado de stopwords podemos acudir a páginas especializadas en IR o Text Mining (http://snowball.tartarus.org/ , http://www.unine.ch/info/clef/).

Nuestra clase SpanishAnalyzer hederará de **org.apache.lucene.analysis.Analyzer** y tendría el siguiente código fuente:

view plain print ?

```
01. package com.autentia.lucene.es;
02.
03. import java.io.File;
04. import java.io.IOException;
05. import java.io.Reader;
06. import java.util.HashSet;
07. import java.util.Set;
08.
09. import org.apache.lucene.analysis.Analyzer;
10. import org.apache.lucene.analysis.LowerCaseFilter;
11. import org.apache.lucene.analysis.StopFilter;
12. import org.apache.lucene.analysis.TokenStream;
13. import org.apache.lucene.analysis.WordlistLoader;
14. import org.apache.lucene.analysis.standard.StandardFilter;
15. import org.apache.lucene.analysis.standard.StandardTokenizer;
16.
17.
18. /** Filters {@link StandardTokenizer} with {@link StandardFilter}, {@link
19.  * LowerCaseFilter}, {@link StopFilter} and {@link SpanishStemFilter}. */
20.
21. /**
22.  * Analyzer for Spanish using the SNOWBALL stemmer. Supports an external list of stopwords
23.  * (words that will not be indexed at all).
24.  * A default set of stopwords is used unless an alternative list is specified, the
25.  * exclusion list is empty by default.
26.  *
27.  * @author jose
28.  */
29.
30. public class SpanishAnalyzer extends Analyzer {
31.
32.     /** An array containing some common Spanish words that are usually not
33.       * useful for searching. Imported from http://www.unine.ch/info/clef/.
34.       */
35.     // TODO: no pego en el tutorial el listado de stopWords utilizado para
36.     // no sobredimensionarlo, son 351 términos.
37.     public static final String[] SPANISH_STOP_WORDS = { "" };
38.
39.     /**
40.       * Contains the stopwords used with the StopFilter.
41.       */
42.     private Set<Object> stopTable = new HashSet<Object>();
43.
44.     /**
45.       * Contains words that should be indexed but not stemmed.
46.       */
47.     private Set<Object> exclTable = new HashSet<Object>();
48.
49.     /**
50.       * Builds an analyzer with the default stop words.
51.       */
52.     public SpanishAnalyzer() {
53.         stopTable = StopFilter.makeStopSet(SPANISH_STOP_WORDS);
54.     }
55.
56.     /** Builds an analyzer with the given stop words. */
57.     public SpanishAnalyzer(String[] stopWords) {
58.         stopTable = StopFilter.makeStopSet(stopWords);
59.     }
60.
61.     /**
62.       * Builds an analyzer with the given stop words from file.
63.       * @throws IOException
64.       */
65.     public SpanishAnalyzer(File stopWords) throws IOException {
66.         stopTable = new HashSet(WordlistLoader.getWordSet(stopWords));
67.     }
68.
69.     /** Constructs a {@link StandardTokenizer} filtered by a {@link
70.       * StandardFilter}, a {@link LowerCaseFilter}, a {@link StopFilter}
71.       * and a {@link SpanishStemFilter}. */
72.     public final TokenStream tokenStream(String fieldName, Reader reader) {
73.         TokenStream result = new StandardTokenizer(reader);
74.         result = new StandardFilter(result);
75.         result = new LowerCaseFilter(result);
76.         result = new StopFilter(result, stopTable);
77.         result = new SpanishStemFilter(result);
78.         return result;
79.     }
80. }
```

En la constante **SPANISH_STOP_WORDS** incluiremos el listado por defecto de stopWords en castellano que tendrá el analizador. Lo ideal sería que el listado final de stopWords se obtuviese de un recurso externo parametrizable (un fichero de propiedades, base de datos...).

En el método **tokenStream** es donde se filtra el contenido, para ello hemos incluido varios filters:

- StandardFilter: básicamente, elimina los signos de puntuación,
- LowerCaseFilter: convierte el contenido a minúsculas,
- StopFilter: filtrará el contenido con el listado de stopWords,
- SpanishStemFilter: objeto del siguiente punto del tutorial.

5. SpanishTemFilter.

Nuestro filtro de stemming heredará de **org.apache.lucene.analysis.TokenFilter** y tendrá el siguiente código fuente:

```
view plain print ?
01. package com.autentia.lucene.es;
02.
03. import java.io.IOException;
04.
05. import net.sf.snowball.ext.SpanishStemmer;
06.
07. import org.apache.lucene.analysis.Token;
08. import org.apache.lucene.analysis.TokenFilter;
09. import org.apache.lucene.analysis.TokenStream;
10.
11. /**
12.  * Spanish stemming algorithm.
13.  */
14. public final class SpanishStemFilter extends TokenFilter {
15.
16.     private SpanishStemmer stemmer;
17.     private Token token = null;
18.
19.     public SpanishStemFilter(TokenStream in) {
20.         super(in);
21.         stemmer = new SpanishStemmer();
22.     }
23.
24.     /** Returns the next input Token, after being stemmed */
25.     public final Token next() throws IOException {
26.         if ((token = input.next()) == null) {
27.             return null;
28.         }
29.         else {
30.             stemmer.setCurrent(token.termText());
31.             stemmer.stem();
32.             String s = stemmer.getCurrent();
33.             if ( !s.equals( token.termText() ) ) {
34.                 return new Token( s, token.startOffset(),
35.                 token.endOffset(), token.type() );
36.             }
37.             return token;
38.         }
39.     }
40.
41.     /**
42.       * Set a alternative/custom Stemmer for this filter.
43.       */
44.     public void setStemmer(SpanishStemmer stemmer) {
45.         if ( stemmer != null ) {
46.             this.stemmer = stemmer;
47.         }
48.     }
49. }
```

Utiliza la clase SpanishStemmer, de la librería lucene-snowball. Snowball es un lenguaje de programación para el manejo de strings que permite implementar fácilmente algoritmos de stemming.

6. Test unitario SpanishAnalyzerTest.

Desde Autentia insistimos mucho en la realización de test (unitarios y de integración) de nuestras aplicaciones.
Una vez implementado el analizador en castellano deberíamos probarlo, y qué mejor que un test de Junit que forme parte de nuestro código fuente?, con ello testeamos el funcionamiento actual y futuro (si alguien toca nuestro código, los tests deben seguir funcionando).

```
view plain print ?
01. package com.autentia.lucene;
02.
03. import junit.framework.TestCase;
04.
05. import java.io.StringReader;
06.
07. import org.apache.log4j.Logger;
08. import org.apache.lucene.analysis.Analyzer;
09. import org.apache.lucene.analysis.Token;
10. import org.apache.lucene.analysis.TokenStream;
11.
12. import com.autentia.lucene.en.EnglishAnalyzer;
13. import com.autentia.lucene.es.SpanishAnalyzer;
14.
15.
16. public class SnowballAnalyzerTest extends TestCase {
17.
18.     private static Logger logger = Logger.getRootLogger();
19.
20.     public SnowballAnalyzerTest(String name) {
21.         super(name);
22.     }
23.
24.     public void assertAnalyzesTo(Analyzer a, String input, String[] output)  throws Exception {
25.         TokenStream ts = a.tokenStream( "content", new StringReader(input));
26.         for (int i=0; i<output.length; i++) {
27.             Token t = ts.next();
28.             assertNotNull(t);
29.             assertEquals(output[i], t.termText());
30.         }
31.         assertNull(ts.next());
32.         ts.close();
33.     }
34.     public void testSpanish() throws Exception {
35.         Analyzer a = new SpanishAnalyzer();
36.         assertAnalyzesTo(a, "foo bar . FOO <> BAR",
37.             new String[] { "foo", "bar", "foo", "bar" });
38.         assertAnalyzesTo(a, "C.A.M.",
39.             new String[] { "cam" });
40.         assertAnalyzesTo(a, "C++",
41.             new String[] { "c" });
42.         assertAnalyzesTo(a, "\"QUOTED\" word",
43.             new String[] { "quot", "word" });
44.         assertAnalyzesTo(a, "El camino del hombre recto",
45.             new String[] { "camin", "hombr", "rect"});
46.         assertAnalyzesTo(a, "está por todos lados rodeado de la injusticia de los egoístas y la tiranía de los hombres mal
47.                 new String[] { "lad", "rod", "injustici", "egoist", "tiran", "hombr", "mal"});
48.     }
49. }
```

7. Búsqueda de documentos por similitud.

Tomando como base el modelo de espacio vectorial que Lucene implementa, podemos realizar una búsqueda por similitud teniendo en cuenta los siguientes parámetros:

- el número mínimo de ocurrencias de una palabra en un documento,
- el número mínimo de ocurrencias de un término, esto es, de la raíz semántica de una palabra en un documento,
- la longitud mínima de una palabra, el número mínimo de caracteres de una palabra para que sea tenida en cuenta. Fortaleciendo, de este modo, el listado de stop words.
- un número máximo de términos para componer la consulta.

En función a los mismos se generará una consulta que estará formada por los términos del vector más relevantes del documento, hasta llegar al número máximo parametrizado.

Lucene distribuye una librería (lucene-similarity) que contiene una clase que implementa ésta funcionalidad: **MoreLikeThis**. Vamos a ver su funcionamiento en el siguiente test:

view plain print ?

```
01. package com.autentia.lucene;
02.
03. import java.io.File;
04. import java.io.FileInputStream;
05. import java.io.IOException;
06.
07. import junit.framework.TestCase;
08.
09. import org.apache.commons.logging.Log;
10. import org.apache.commons.logging.LogFactory;
11. import org.apache.lucene.analysis.Analyzer;
12. import org.apache.lucene.document.Document;
13. import org.apache.lucene.document.Field;
14. import org.apache.lucene.document.Field.Index;
15. import org.apache.lucene.document.Field.Store;
16. import org.apache.lucene.document.Field.TermVector;
17. import org.apache.lucene.index.IndexReader;
18. import org.apache.lucene.index.IndexWriter;
19. import org.apache.lucene.search.Hits;
20. import org.apache.lucene.search.IndexSearcher;
21. import org.apache.lucene.search.Query;
22. import org.apache.lucene.search.similar.MoreLikeThis;
23. import org.apache.poi.hwpf.extractor.WordExtractor;
24.
25. import com.autentia.lucene.es.SpanishAnalyzer;
26.
27. /**
28.  * Test that verifies the search by similarity of documents.
29.  */
30. public class SimilarityDocumentSearchTest extends TestCase {
31.
32.     private static Log log = LogFactory.getLog(SimilarityDocumentSearchTest. class);
33.
34.     /** repository path */
35.     private static final String PATH = "localhost_index";
36.
37.     /** Name of the field in Lucene that contains the content of the document */
38.     private static final String FIELD_CONTENT_NAME = "content";
39.
40.     /** Name of the search fields in Lucene */
41.     private static final String[] DEFAULT_FIELD_NAMES = new String[] { FIELD_CONTENT_NAME  };
42.
43.     /** Ignore words that are less than it often in the document code */
44.     private static final int DEFALT_MIN_DOC_FREQ = 1;
45.
46.     /** Ignore terms that are less than it often in the document code */
47.     private static final int DEFAULT_MIN_TERM_FREQ = 1;
48.
49.     /** Maximum number of terms that will be included in the query */
50.     private static final int MAX_QUERY_TERMS = 1000;
51.
52.     /** Minimum length of a word to be taken into consideration */
53.     private static final int DEFAULT_MIN_WORD_LENGTH = 2;
54.
55.     /** Our SpanishAnalyzer */
56.     private static Analyzer spanishAnalyzer = new SpanishAnalyzer();
57.
58.     /** Number total of documents indexed */
59.     private int totalDocs = 0;
60.
61.     public SimilarityDocumentSearchTest(String name) {
62.         super(name);
63.     }
64.
65.     @Override
66.     protected void setUp() throws Exception {
67.         log.trace("Entering " + getName());
68.         createIndex(spanishAnalyzer);
69.         indexFiles("files");
70.         log.debug("'"+totalDocs+"' documents indexed.");
71.     }
72.
73.     @Override
74.     protected void tearDown() throws Exception {
75.         log.trace("Exiting " + getName());
76.         destroyIndex();
77.     }
78.
79.     /** run test */
80.     public void test_DocumentAnalyzer() throws Throwable {
81.         moreLikeThisAnalyzer(spanishAnalyzer, "files/original.doc");
82.     }
83.
84.     /** Search documents by similarity using the class {@link MoreLikeThis} */
85.     private void moreLikeThisAnalyzer(Analyzer analyzer, String original)  throws Throwable {
86.         log.trace("Entering");
87.
88.         IndexReader indexReader = IndexReader.open(PATH);
89.         IndexSearcher indexSearcher = new IndexSearcher(indexReader);
90.
91.         MoreLikeThis mlt = new MoreLikeThis(indexReader);
92.         mlt.setFieldNames(DEFAULT_FIELD_NAMES);
93.         mlt.setMinDocFreq(DEFALT_MIN_DOC_FREQ);
94.         mlt.setMinTermFreq(DEFAULT_MIN_TERM_FREQ);
95.         mlt.setMaxQueryTerms(MAX_QUERY_TERMS);
96.         mlt.setMinWordLen(DEFAULT_MIN_WORD_LENGTH);
97.         mlt.setAnalyzer(analyzer);
98.
99.         Query query= mlt.like( new FileInputStream(getClass().getClassLoader().getResource(original).getPath()) );
100.
101.         Hits hits = indexSearcher.search(query);
102.
103.         int len = hits.length();
104.
105.         log.debug("----------------------------------------" );
106.         log.debug("original :" + original);
107.         log.debug("query: " + query);
108.         log.debug("found: " + len + " documents");
109.         for (int i = 0; i < Math.min(25, len); i++) {
110.             Document d = hits.doc(i);
111.             log.debug("score   : " + hits.score(i));
112.             log.debug("name    : " + d.get("name"));
113.         }
114.         log.debug("----------------------------------------" );
115.         log.trace("Exiting");
116.     }
117.
118.     /** Created the index with the analyzer given as parameter*/
119.     private void createIndex(Analyzer analyzer) throws Exception {
120.         IndexWriter writer = new IndexWriter(PATH, analyzer, true);
121.         writer.setUseCompoundFile(false);
122.         writer.close();
123.         log.debug("Index created.");
124.     }
```

El test se ejecuta en un entorno en el que el directorio **src/test/resources/files** contiene una serie de documentos, en formato word, que serán indexados. Y se busca los documentos similares al documento Original.doc [línea 60].

A destacar:

- línea 70 y siguientes: en la que se crea una instancia de la clase MoreLikeThis y se setean los parámetros que hemos comentado para la generación de la consulta.
- línea 135: en la que se crea el campo del documento de Lucene que almacenará el contenido de nuestro documento, indicando para ello el tipo de almacenamiento que tendrá nuestro vector de términos (TermVector.WITH_POSITIONS_OFFSETS) y extrayendo el contenido de nuestro documento de Microsoft Word utilizando la última versión de la librería poi

Si tienes la oportunidad pruébalo con alguno de los documentos que manejas habitualmente, te sorprenderán los resultados.

Recomendamos utilizar una capa de indexación que encapsule la lógica de acceso a Lucene como lius, y si tu problemática es solo la extracción del contenido de los documentos (en diferentes formatos) puedes echar un ojo al proyecto tika de apache, basado en lius.

8. Conclusiones.

Hemos visto cómo realizar un analizador semántico en castellano para Lucene que implemente un listado elaborado de stopwords, y filtre el contenido de los términos, a indexar y buscar, en función de su raíz semántica. Con ello podemos comprobar cómo realizar búsquedas de documentos por simlitud, con un alto grado de acierto.

Y hemos potenciado la funcionalidad de Lucene, se te ocurre alguna aplicación práctica?, a nosotros sí.

Si necesitas un empujón a tus desarrollos, ampliar la funcionalidad de tus aplicaciones, un estudio de la viabilidad de implementar ésta tecnología u otras... llámanos!!!

Somos Autentia.

Tutoriales recomendados

Nombre	Resumen	Fecha	Visitas	pdf
Framework para indexación de documentos en Lucene: LIUS	En este tutorial vamos a ver como usar el framework LIUS (Lucene Index Update and Search) para indexar documentos en el motor de busqueda textual Lucene	2007-10-23	1350	pdf
Creación de documentos PDF en sitios web utilizando el componente AspPDF	En este tutorial se muestran las características generales del componente ASP-PDF que permite gestionar documentos PDF desde una aplicación web	2007-03-20	3533	pdf
Primeros pasos con el motor de busqueda Java Lucene	Os mostramos los primeros pasos con uno de los más extendidos motores de búsqueda dentro del mundo OpenSource y Java: Tambien veremos lo sencillo que es monitorizar los índices con Luke e indexar PDFs con PDFBox	2006-07-31	6750	pdf
Exportar PDF multiidioma con iReport	Este tutorial prentende solucionar los problemas que pueden ocasionarnos la exportación de informes en PDF usando la herramiento iReport en diferentes idiomas	2007-04-23	3470	pdf
Extracción de texto de documentos Office desde Java	En este tutorial vamos a ver como podemos extraer texto de los documentos de Office (DOC, XLS y PPT) desde Java	2007-05-22	4673	pdf
Análisis de rendimiento (Profiling) de aplicaciones web con eclipse	En este tutorial se va a explicar como analizar el rendimiento de nuestras aplicaciones web con una herramienta propia de Eclipse, llamada Eclipse TPTP.	2007-04-19	3861	pdf
Leer un documento de Microsoft Word usando la librería POI de jakarta	En este docuemento aprenderemos a leer un documento de Microsoft Word usando la librería POI de jakarta	2006-09-20	7118	pdf