UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA INDUSTRIAL
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISEÑO E IMPLEMENTACIÓN DE UN METABUSCADOR DE PÁRRAFOS PARA LA RECUPERACIÓN DE DOCUMENTOS SIMILARES EN LA WEB

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INDUSTRIAL E INGENIERO CIVIL EN COMPUTACIÓN

FELIPE JOSÉ BRAVO MÁRQUEZ

PROFESOR GUÍA:
SEBASTIÁN ALEJANDRO RÍOS PEREZ

MIEMBROS DE LA COMISIÓN:
LUIS GUERRERO BLANCO
JUAN MUARICIO MARÍN CAIHUAN
GASTÓN ANDRÉS L'HUILLIER CHAPARRO

SANTIAGO, CHILE
OCTUBRE 2010

ESTE TRABAJO HA SIDO FINANCIADO POR EL PROYECTO FONDEF
DO8I-1015 TITULADO DOCODE

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INDUSTRIAL E
INGENIERO CIVIL EN COMPUTACIÓN
FELIPE BRAVO MÁRQUEZ
FECHA: 28/09/2010
PROF. GUIA: SR. SEBASTIÁN RÍOS

DISEÑO E IMPLEMENTACIÓN DE UN METABUSCADOR DE PÁRRAFOS PARA LA RECUPERACIÓN DE DOCUMENTOS SIMILARES EN LA WEB

La recuperación de documentos similares a un documento dado en la Web es un problema no resuelto por los motores de búsqueda tradicionales. Esto se debe, a que los motores de búsqueda están pensados para resolver necesidades de información de usuarios basadas en conjuntos pequeños de palabras clave. En este trabajo se define el problema de recuperación de documentos similares como el proceso en que un usuario ingresa un párrafo a un sistema de información y éste le retorna los documentos con mayor similitud a éste en la Web. Los documentos recuperados son rankeados por medio de una métrica de similitud estimada por el sistema. La resolución del problema, podría ser utilizado en contextos como la detección de plagio, el análisis de impacto de documentos en la Web y la recuperación de ideas similares. En este trabajo en particular, se pretende resolver el problema en el contexto de la detección de plagio en documentos digitales en el marco del proyecto FONDEF titulado DOCODE.

Se propone una metodología basada en modelos de lenguaje generativos y metabuscadores. Los modelos de lenguaje son utilizados como generadores de consultas aleatorizadas sobre el texto del documento entregado, donde se propone un modelo que extrae términos relevantes sin reposición llamado Modelo de Lenguaje Hipergeométrico. El conjunto de consultas generado intenta ser una representación de la información relevante del documento. Posteriormente, cada consulta es enviada como entrada a una lista de motores de búsqueda de la Web. Para luego combinar los resultados de cada búsqueda en una única respuesta. A este proceso se le conoce como metabúsqueda. Finalmente, los resultados son ordenados por relevancia y presentados al usuario. Para estimar la relevancia entre el documento de entrada y los documentos encontrados se propone una función de scoring basada en la ley de Zipf, la cual considera los ranking locales de aparición de cada resultado, la confianza en los motores de búsqueda utilizados y la cantidad instancias de búsqueda en que éstos son recuperados.

Se definen los requerimientos de software junto a un análisis preliminar de las características de éste donde se define una arquitectura basada en capas. La capa de aplicación de la herramienta es diseñada acorde a una arquitectura orientada a servicios, de manera que pueda interoperar con otros sistemas. La herramienta se diseña en base al paradigma de orientación a objetos y el uso patrones de diseño conocidos. Esto se realiza para permitir la mantenibilidad y extensibilidad del modelo al uso de diversas estrategias para la generación de consultas, rankeo de resultados y para permitir al metabuscador hacer uso de diversos motores de búsqueda externos. La capa interfaz se diseña como una interfaz Web donde el usuario ingresa el párrafo dentro de un cuadro de texto, permitiéndole a éste además, evaluar la calidad de los resultados entregados. Los resultados son registrados dentro de la capa de datos, para poder evaluar continuamente la calidad predictiva del modelo al adaptar sus parámetros al conocimiento entregado por los usuarios.

Una vez implementada la herramienta, se realizan una serie de experimentos basados en métricas de recuperación de información para evaluar la calidad del modelo en la herramienta implementada. Los resultados muestran que la propuesta es capaz de resolver el problema de recuperación de documentos similares con altos niveles de precisión. Además se demuestra que la combinación de varios motores de búsquedas mejora la calidad de los resultados entregados.

Finalmente, se evalúan la calidad del diseño y la implementación del software. Para el caso del diseño de software la evaluación se realiza en base a métricas de orientación a objetos, y para el caso de la implementación se evalúan la funcionalidad y el rendimiento en base a casos de pruebas. En ambos casos, los resultados obtenidos comprueban la extensibilad y mantenibilidad del diseño junto al cumplimiento de los requerimientos funcionales y no funcionales establecidos.

Dedicado a mis padres por su apoyo incondicional.

Índice general

1. Introducci[Pleaseinsertintopreamble]n **1**
1.1. Marco Conceptual . . . 2
1.1.1. Recuperación de Información . . . 2
1.1.2. Índice Invertido . . . 2
1.1.3. Motores de Búsqueda . . . 2
1.1.4. Modelo Vectorial TF-IDF . . . 3
1.1.5. Modelos de Lenguaje . . . 4
1.1.6. Medidas de Evaluación de Sistemas de Recuperación de Información 5
1.1.7. Servicio Web . . . 5
1.2. Planteamiento del Problema y Motivación . . . 6
1.3. Contexto de la Investigación . . . 6
1.4. Hipótesis de Investigación . . . 7
1.5. Objetivo General . . . 7
1.5.1. Objetivos Específicos . . . 7
1.6. Metodología . . . 8
1.7. Alcances . . . 8
1.8. Resultados Esperados . . . 9

2. Trabajo Relacionado **10**
2.1. Proceso de Fingerprinting de un Documento . . . 10
2.2. Metabuscadores . . . 10
2.3. Recuperación de Documentos Similares usando Metabuscadores . . . 11
2.4. Herramientas existentes para la Recuperación de Documentos Similares . . 11

3. Requerimientos de Software y Análisis Preliminar **13**
3.1. Descripción del Software . . . 13
3.2. Actores del sistema . . . 14
3.3. Funciones del Sistema . . . 14
3.4. Atributos no funcionales . . . 15
3.5. Casos de Uso . . . 15
3.6. Modelo de Análisis . . . 17
3.7. Diagrama de secuencia . . . 19
3.8. Capas del Sistema . . . 20

4. Modelo Lógico **21**
4.1. Variables del Proceso . . . 21
4.2. Modelo de Lenguaje Hipergeométrico . . . 22
4.2.1. Ejemplo de cálculo de probabilidades de MLH . . . 23
4.2.2. Algoritmo para la Generación de Consultas de MLH . . . 23

4.2.3. Consideraciones del MLH 24
4.3. Diseño Lógico del Metabuscador 25
4.3.1. Una función de score Zipf-Like 25
4.3.2. Algoritmo de Metabúsqueda 27

5. Diseño e Implementación de Software 29
5.1. Arquitectura 29
5.1.1. Marco de Referencia J2EE 30
5.2. Diseño Orientado a Objetos 31
5.2.1. Diseño de Clases 31
5.3. Patrones de Diseño 38
5.3.1. Strategy 38
5.3.2. Factory Method 39
5.3.3. Abstract Factory 39
5.3.4. Iterator 40
5.4. Interfaz de Usuario 40
5.5. Modelo de Datos para el almacenamiento de Ejecución 42
5.5.1. Tablas del Modelo 42
5.6. Implementación 44
5.6.1. Capa Interfaz 44
5.6.2. Capa Aplicación 45
5.6.3. Capa Datos 45

6. Evaluación del Modelo Propuesto 46
6.1. Diseño del Experimento 46
6.2. Criterio de Evaluación 46
6.3. Resultados y Discusiones 47

7. Evaluación del Software 50
7.1. Evaluación con Métricas de Orientación a Objetos 50
7.1.1. Resultados de la evaluación 51
7.1.2. Discusión de valores obtenidos 51
7.2. Pruebas 53
7.2.1. Prueba de Funcionalidad 53
7.2.2. Prueba de Tiempo 54

8. Conclusiones 55

Índice de cuadros

1.1. Ejemplo de probabilidades de términos 4

6.1. Número de párrafos, consultas y documentos recuperados por tipo usados en el proceso de evaluación. 47
6.2. Valores de los parámetros usados en el experimento. 47
6.3. Puntaje promedio para DEPs y no DEPs por ranking 48
6.4. Precision at k por tipo de documento de entrada 48

7.1. Métricas de orientación a objetos de clases 52
7.2. Tiempos de ejecución 54

Índice de figuras

3.1. Caso de uso del sistema 16
3.2. Diagrama de Análisis 18
3.3. Diagrama de secuencia 19
3.4. Capas de la Herramienta 20

5.1. Arquitectura de DOCODE-lite 30
5.2. Marco de Referencia J2EE 31
5.3. Diagrama de Clases 32
5.4. Cuadro de texto donde el usuario inserta el párrafo 41
5.5. Evaluación de un Resultado 41
5.6. Modelo de Datos de registro del Proceso 44

6.1. Precision at k para distintos tipos de documentos de entrada. 49
6.2. Precisión de los resultados entregados por un único motor de búsqueda . . . 49

Capítulo 1

Introducción

El desarrollo y la masificación de motores de búsqueda en la Web como Google, Bing o Yahoo! han permitido hoy en día a millones de usuarios resolver sus necesidades de información [9]. La mayor parte de los usuarios al verse enfrentados a representar una necesidad de información dentro de la interfaz de un motor de búsqueda, tienden a buscar en su mente cuáles son las palabras que mejor la representan. A estas palabras se les conoce como *key terms*. Sin embargo, no todas las consultas realizadas a un motor de búsqueda, buscan resolver los mismos tipos de necesidades de información. En [15] se propone la siguiente clasificación de consultas:

1. Consultas Informacionales: Cuando se busca información general sobre un tema específico. Por ejemplo la consulta *colegio en Rancagua* busca recuperar todos los documentos relacionados con colegios ubicados en Rancagua.

2. Consultas Navegacionales: Cuando se busca una página particular que el usuario tiene en mente. Por ejemplo si un usuario quiere llegar a la página Web del Banco de Chile e ingresa la consulta *Banco de Chile*.

3. Consultas transaccionales: Cuando el usuario busca realizar una acción con su búsqueda, como la compra de productos o la descarga de un archivo.

En [12] se analiza el problema de recuperar documentos similares a un documento dado en la Web. En este caso, se tiene una diferencia significativa respecto de las consultas realizadas en motores de búsqueda tradicionales, puesto que se usa como entrada un documento completo en vez de una consulta formada por un conjunto de *key terms*. Este problema puede ser considerado como un tipo de necesidad de información diferente. A lo largo de este trabajo, será llamado como el problema de *recuperación de documentos similares* RDS. Soluciones de RDS podrían ser aplicados en los siguientes contextos:

1. *Detección de Plagio*: Se usa un documento de originalidad sospechosa de entrada, al recuperar documentos similares a éste, probablemente se encontrará su fuente.

2. *Análisis de Impacto*: Documentos de noticias, leyes o entradas de Blog, podrían ser usados como entrada para recuperar así sus múltiples versiones en la Web. Se podrían encontrar todas las referencias que se le hacen, permitiendo realizar un análisis del impacto del documento en la Web.

3. *Recuperación de Ideas Similares*: Una idea, una patente, un poema o una nueva teoría, podrían ser utilizadas para la recuperación de documentos similares, encontrando así documentos que traten temas similares al entregado. El autor no tendría

que definir su idea en términos claves para entregarla a un buscador, basta que use su texto como entrada.

Los motores de búsqueda tradicionales podrían resolver el problema RDS permitiendo a los usuarios entregar documentos completos de entrada. En muchos enfoques de ranking de consultas, éstas son tratadas como documentos y comparadas con todos los documentos indexados en la colección. Luego se computan sus similitudes y se rankean los resultados por similitud. Sin embargo, los motores de búsqueda soportan consultas de un largo máximo definido, esto se debe a que las consultas largas requieren mucho procesamiento, dificultan el proceso de caching de consultas, y generalmente están compuestas por muchos términos irrelevantes [13].

En este trabajo se pretende desarrollar una metodología parar convertir un documento sospechoso en un conjunto de consultas que sean capaces de representar la información relevante de éste. Luego, enviar las consultas a los motores de búsqueda más conocidos, mezclar los resultados recuperados y asignarles puntaje por similitud al documento de entrada. Esta metodología pretende ser una solución al problema de recuperación de documentos similares en el contexto de la detección de plagio de documentos.

1.1. Marco Conceptual

1.1.1. Recuperación de Información

La recuperación de información es la ciencia de buscar información dentro de una colección de documentos. El contenido de los documentos puede ser estructurado, semi-estructurado o no estructurado. En este trabajo se pretende recuperar información de documentos escritos en lenguaje natural, principalmente documentos escritos en español. Un documento en lenguaje natural no posee una estructura formal [15].

1.1.2. Índice Invertido

El vocabulario de una colección de documentos es el conjunto de todos los términos diferentes presentes en éste. La lista de posteo de un término, es una lista con todos los documentos en los que éste aparece. Luego, un índice invertido básico es una estructura de datos que mapea cada término del vocabulario a su lista de posteo. Existen estructuras más complejas que consideran la posición en la que ocurre el término en el documento. A ese tipo de índices se les llama índices invertidos posicionales [4].

1.1.3. Motores de Búsqueda

Un motor de búsqueda es un sistema de recuperación de información diseñado para la búsqueda de información en la Web [9]. Sus componentes básicos son:

- Crawler: Un robot que navega la Web según una estrategia definida. Generalmente comienza navegando por un conjunto de páginas semilla y continua navegando por sus hipervínculos.
- Indexador: Encargado de mantener un índice invertido con el contenido de las páginas recorridas por el Crawler.
- Máquina de consultas: Encargado de procesar las consultas y buscar en el índice los documentos con mayor similitud a ella.

- Función de score: Es la función que tiene la máquina de consulta para computar la similitud entre la consulta y los documentos indexados. La función es usada para rankear los documentos por su similitud con la consulta entregada por un usuario.
- Interfaz: Interactúa con el usuario, recibe la consulta como entrada y retorna los documentos rankeados por similitud.

En [4] se proponen las siguientes dos alternativas de arquitectura para un motor de búsqueda:

Arquitectura Centralizada

El índice invertido se mantiene de forma centralizada, todas las consultas las procesa un único procesador.

Arquitectura Distribuida

El índice invertido es distribuido en distintos procesadores. El sistema encargado de recibir las consultas y distribuirlas a los distintos procesadores es el *broker*. La forma de trabajar del broker depende de la manera en que se distribuya el índice.

- **Índice particionado por Términos**: Los términos y sus listas de posteo están distribuidas en distintos procesadores. Entonces cuando llega una consulta, el broker envía cada término al procesador correspondiente, luego cada procesador le retorna al broker la lista de posteo encontrada. El broker realiza una intersección de las listas de posteo y rankea.
- **Índice particionado por Documentos**: El broker repliega la consulta concurrentemente a los P procesadores. Cada procesador retorna sus k documentos más relevantes de la consulta al broker. Finalmente el broker realiza un ranking global de los k mejores de los $P \times k$ documentos recibidos.

1.1.4. Modelo Vectorial TF-IDF

Una forma popular de representar los documentos y consultas, es mediante el modelo vectorial. En este modelo el documento es modelado como un vector de pesos de los términos que contiene [21].

$$d_i \rightarrow \overrightarrow{d_i} = (w(t_1, d_i), ..., w(t_n, d_i)) \tag{1.1}$$

El peso de cada término en el documento esta condicionado por su frecuencia en el documento $tf_{i,d}$ y la cantidad de documentos en los que aparece el término en la colección n_i. La idea central es que un término mientras mayor sea su frecuencia en el documento y menor sea su n_i, mayor es la información que aporta al documento. Puesto que un término que aparece en demasiados documentos tiende a ser una palabra común del lenguaje como los artículos y preposiciones que no aportan información relevante. De esta forma se calcula el peso de un término t_i en d $w(t_i, d)$ como:

$$w(t_i, d) = \frac{tf_{i,d} \times idf_i}{|\overrightarrow{d}|} = \frac{tf_{i,d} \times log\frac{N}{n_i}}{\sqrt{\sum_{s=i}^{k}(tf_{i,d} \times log\frac{N}{n_i})^2}} \tag{1.2}$$

término	probabilidad
esto	0.15
es	0.2
un	0.1
modelo	0.02
de	0.2
lenguaje	0.02
STOP	0.31

Cuadro 1.1: Ejemplo de probabilidades de términos

A $log\frac{N}{n_i}$ se le llama idf_i por que representa la frecuencia inversa de documentos que contienen el término i en la colección. Luego una consulta q se puede modelar como un vector de términos al igual que un documento. La similitud de q y un documento d_i se mide con una similitud vectorial llamada similaridad coseno:

$$sim(d,q) = \overrightarrow{d} \cdot \overrightarrow{q} = \sum wd_i \times wq_i \tag{1.3}$$

Usando la similitud coseno como función de rankeo se puede utilizar el modelo vectorial en una máquina de búsqueda [15].

1.1.5. Modelos de Lenguaje

Los modelos de lenguaje son una alternativa al modelo vectorial para modelar documentos en máquinas de búsqueda. Tienen una función que asigna una medida de probabilidad sobre todos los strings construíbles a partir de un alfabeto σ [15, 25] y cumplen con la siguiente condición:

$$\sum_{s \in \sigma^*} P(s) = 1 \tag{1.4}$$

En particular el modelo de lenguaje M_d de un documento d es una distribución de probabilidad de todos los términos del vocabulario V al cual el documento pertenece.

$$\sum_{s \in V^*} P(s) = 1 \tag{1.5}$$

Un modelo de lenguaje puede utilizarse para medir la similitud de una consulta q con un documento d. Esta similitud permite utilizarlos para rankear consultas en una máquina de búsqueda. Se define $P(q|M_d)$ como la probabilidad de que una consulta q haya sido generada por el modelo de lenguaje M_d de d. La forma de rankear con modelo de lenguaje es usando la probabilidad de generación como medida de similitud.

Otra cualidad de los modelos de lenguaje es que pueden utilizarse para generar oraciones que pertenezcan al modelo de lenguaje. Debido a esta propiedad es que se le llama a los modelos de lenguaje como modelos generativos.

Un modelo generativo requiere una probabilidad adicional llamada la probabilidad de parada $P(STOP)$. Un ejemplo, sea M un modelo de lenguaje unigrama donde las probabilidades de generar cada término se encuentran en el cuadro 1.1.5:

La probabilidad de que M genere el String *lenguaje de modelo* sería:

$$P(lenguaje) \times P(1 - STOP) \times P(de) \times P(1 - STOP) \times P(modelo) \times P(STOP) \tag{1.6}$$

Lo que equivale a:

$$0{,}02 \times 0{,}69 \times 0{,}2 \times 0{,}69 \times 0{,}02 \times 0{,}31 = 0{,}000011807 \tag{1.7}$$

Tipos de Modelos de Lenguaje

La forma en que se construyen las probabilidades sobre las secuencias de términos condicionan el tipo de modelo de lenguaje. Una propiedad utilizada para descomponer las probabilidades de secuencias de eventos en probabilidades condicionales sucesivas de eventos anteriores es la regla de la cadena presentada a continuación:

$$P(t_1, t_2, t_3, t_4) = P(t_1)P(t_2|t_1)P(t_3|t_1, t_2)P(t_4|t_1, t_2, t_3) \tag{1.8}$$

El modelo de lenguaje más simple, es el modelo de lenguaje unigrama que asume independencia entre los términos:

$$P(t_1, t_2, t_3, t_4) = P(t_1)P(t_2)P(t_3)P(t_4) \tag{1.9}$$

Existen también modelos más complejos como el bigrama, donde cada término está condicionado sólo por el término anterior:

$$P(t_1, t_2, t_3, t_4) = P(t_1)P(t_2|t_1)P(t_3|t_2)P(t_4|t_3) \tag{1.10}$$

1.1.6. Medidas de Evaluación de Sistemas de Recuperación de Información

Para evaluar sistemas de recuperación de información, es necesario medir la relevancia de los documentos recuperados frente a una consulta particular. Las medidas de evaluación más comunes son:

1. **Precisión**: La precisión P es la fracción de documentos recuperados que son relevantes:
$$\textbf{Precision} = \frac{\#\text{ documentos relevantes recuperados}}{\#\text{documentos recuperados}} \tag{1.11}$$

2. **Recall**: El recall R es la fracción de documentos relevantes recuperados sobre todos los documentos relevantes existentes para la consulta en la colección:
$$\textbf{Recall} = \frac{\#\text{documentos relevantes recuperados}}{\#\text{documentos relevantes en la colección}} \tag{1.12}$$

A medida aumenta el número de documentos recuperados, la precisión decrece y en recall aumenta. Existen colecciones de referencias con consultas y documentos previamente evaluadas por relevancia. También se puede evaluar una máquina de búsqueda realizando un conjunto de consultas y medir la relevancia de los resultados con información experta.

1.1.7. Servicio Web

La *W3C*[1] define un servicio Web como : "Un sistema de software interoperable diseñado para dar apoyo a la interacción máquina a máquina sobre una red". La comunicación entre

[1] WWW Consortium

el consumidor y el proveedor del servicio se realiza mediante mensajes *SOAP*[2] transportados a través del protocolo *HTTP*, y serializados en XML y otros estándares Web. Un Web Service se describe por el estándar *WSDL*[3] y es almacenado en un repositorio *UDDI*[4] [18].

1.2. Planteamiento del Problema y Motivación

La Web se ha convertido en una fuente de información universal y democrática donde todos los usuarios son libres de acceder a la información y compartir nuevo contenido. Situándose en el contexto de la educación, la masificación de la Web y los motores de búsqueda han permitido a los estudiantes un acceso a información bibliográfica mucho mayor al disponible en sus hogares o establecimientos educacionales. Esto ha tenido una implicancia positiva, puesto que los estudiantes pueden hoy en día informarse y ampliar su conocimiento sin salir de sus casas. Pero por otro lado ha dificultado la labor de los docentes a la hora de verificar la autenticidad de los trabajos realizados, más si consideramos que los alumnos tienden a tener un mayor dominio de la tecnología que sus docentes.

Cuando un docente corrige el trabajo de un alumno y tiene sospecha de autenticidad, debe darse el trabajo de consultar en algún buscador cada una de las frases sospechosas. Luego, debe verificar cada uno de los resultados obtenidos esperando haber encontrado el documento original. Consideremos ahora que el profesor debe corregir 20 trabajos, si siguiese la misma estrategia que sigue para verificar un único párrafo nos encontramos en la situación que el profesor invierte más tiempo en verificar autenticidad que en corregir el trabajo mismo. Existen algoritmos eficientes para detectar patrones de plagio dentro de una colección de documentos. Estos algoritmos, son útiles para comparar un conjunto de documentos entre sí, como por ejemplo todos los trabajos de una materia. Pero para comparar el documento con una colección tan grande como la Web, se requieren otras estrategias.

Una posible solución podría ser, diseñar un crawler que indexe una parte significativa de la web y así comparar todo el documento sospechoso frente a todos los documentos indexados, por ejemplo usando el modelo vectorial. Pero los recursos necesarios para indexar la Web no son alcanzables. Otra alternativa sería ingresar el documento completo en motores de búsqueda comerciales, pero éstos aceptan consultas de cantidad limitada de términos, lo que obligaría a particionar el documento en varias consultas volviendo así al problema del docente previamente descrito.

La motivación de éste trabajo es entregar una herramienta capaz de resolver el problema de la recuperación de documentos similares de manera rápida y precisa, con aplicabilidad en la detección de plagio de documentos. Si bien el contexto del proyecto es la detección de plagio, este trabajo está orientado a recuperar documentos similares en la Web, que como se mencionó anteriormente, puede también ser aplicado en contextos diferentes a la detección de plagio.

1.3. Contexto de la Investigación

Esta investigación es financiada por FONDEF[5]. El proyecto, lleva el nombre de DOCODE[6](código DO8I-1015) y fue adjudicado al Departamento de Ingeniería Industrial de

[2]Simple Object Access Protocol
[3]WEB Service Description Language
[4]Universal Description and Discovery Introduction Interface
[5]Fondo de Fomento al Desarrollo Científico y Tecnológico
[6]Document Copy Detection

la Universidad de Chile. El objetivo principal del proyecto, es contribuir a mejorar la calidad de la educación a través de un mejor control del fenómeno de la copia en documentos electrónicos. Para ello, busca transformarse en un sistema detector de copia eficaz para el idioma español que verifique si un documento digital es original, comparando sus textos con fuentes en la Web y en bases documentales propietarias.

1.4. Hipótesis de Investigación

A lo largo de esta investigación, se busca validar o refutar la siguiente hipótesis de investigación:

"Al enviar un conjunto de consultas generadas a partir de extracciones aleatorias de términos de un párrafo dado a una lista de motores de búsqueda de la Web, para luego combinar y rankear los resultados recuperados mediante una función de score que usa solamente información retornada por cada instancia de búsqueda, se soluciona el problema de recuperación de documentos similares con altos niveles de precisión".

1.5. Objetivo General

El objetivo principal de este trabajo, es diseñar e implementar una herramienta computacional capaz de recuperar documentos similares a un párrafo dado por un usuario en la Web, donde los documentos encontrados se encuentren ordenados por similitud.

1.5.1. Objetivos Específicos

1. Diseñar e implementar un modelo probabilístico basado en modelos de lenguaje generativos, que sea capaz de genear un conjunto de consultas a partir de un párrafo dado. La idea es que estas consultas contengan la información más relevante de éste.

2. Diseñar e implementar un Metabuscador que envié el conjunto de consultas generados a los motores de búsquedas más conocidos de la Web (Google, Yahoo!, Bing) y combine los resultados en una única respuesta.

3. Diseñar e implementar una función de score que le asigne puntaje a los documentos encontrados por similitud al documento entregado.

4. Diseñar e implementar la herramienta usando orientación a objetos y patrones de diseño conocidos. De manera tal que el modelo pueda ser extendido a distintos modelos de generación de consultas, de scoring de resultados y motores de búsqueda usados en el proceso de metabúsquedas.

5. Implementar la herramienta bajo estándares de Web Service para permitir la interoperabilidad con otros sistemas.

6. Diseñar e implementar una interfaz para que un usuario pueda utilizar la aplicación desde un navegador Web.

7. Diseñar e implementar un mecanismo que permita la evaluación continua de los resultados obtenidos utilizando el conocimiento de los usuarios.

8. Probar y evaluar la herramienta usando métricas conocidas.

9. Explicar los resultados obtenidos.

1.6. Metodología

Para cumplir con los objetivos propuestos, se propone una metodología iterativa, que combina las etapas de un desarrollo de Software tradicional con las etapas del método científico necesarias para validar una hipótesis de investigación. Las etapas de la metodología se presentan a continuación:

- Investigación de trabajo relacionado: En esta etapa se realiza toda la documentación necesaria para concretar el trabajo. Principalmente se basará en publicaciones de recuperación de información, metabuscadores, generación de consultadas a partir de documentos y sistemas de detección de plagio.
- Levantamiento de Requerimientos de Software: Se levantan los requisitos funcionales y no funcionales del sistema para que pueda cumplir con los objetivos de la investigación. Luego se detallan los requierimientos mediante de casos de uso los cuales son refinados identificando clases de análisis y un modelo de secuencia.
- Diseño del modelo lógico: Se plantea un modelo formal para la generación de consultas y propagación de consultadas a los motores de búsqueda. El modelo será fundamentado matemáticamente junto a los pseudocódigos de los algoritmos que se propongan.
- Diseño de Software: Se lleva a cabo un diseño orientado a objetos provisto de un diagrama de clases que soporte el modelo propuesto en la etapa anterior. Se propone además un diseño de la interfaz de usuario que tendrá el sistema y un modelo de datos que soporte el registro de la ejecución de éste.
- Implementación de la herramienta: En esta etapa se procede a desarrollar la herramienta usando el lenguaje de programación Java.
- Evaluación del modelo propuesto: Se definen métricas de recuperación de información para evaluar la calidad predictiva del modelo. Se construye el conjunto de datos para realizar los experimentos. Posteriormente se realizan los experimentos, se calculan las métricas y se discuten los resultados obtenidos.
- Evaluación de diseño e implementación: Se evalúan el diseño de la aplicación mediante métricas de orientación a objetos y la implementación de ésta mediante casos de prueba.
- Conclusiones: Se realizan todas las conclusiones pertinentes basadas en los resultados obtenidos y la investigación en general.

1.7. Alcances

Al final de este trabajo se espera haber desarrollado tanto un modelo teórico bien fundamentado para solucionar el problema de la recuperación de documentos similares, como un software operativo capaz de soportarlo. Se espera también validar el modelo con resultados experimentales y la herramienta con métricas conocidas. Además se considera dentro de los alcances que la herramienta sea extensible al uso de variados motores de búsqueda, estrategias de generación de consultas y funciones de scoring de resultados.

1.8. Resultados Esperados

Como resultado principal se espera que el modelo desarrollado sea capaz de resolver el problema y validar así la hipótesis de investigación previamente descrita. A grandes rasgos, se espera que cuando un usuario entregue un párrafo a la herramienta, los resultados recuperados sean efectivamente similares a éste y el puntaje asginado tenga relación con la similitud sintáctica entre ambos documentos. Se espera además, que el modelo funcione correctamente para un párrafo de entrada en vez de un documento completo. Puesto que un párrafo contiene generalmente una unidad única de información, entonces un documento con párrafos generados a partir de muchos documentos diferentes generarían ruido a la hora de encontrar sus documentos similares.

Capítulo 2

Trabajo Relacionado

En este capítulo, se revisan investigaciones previas sobre los tópicos de mayor relevancia para este trabajo. Se consideran tópicos como la extracción de información relevante a partir de documentos, metabuscadores y herramientas o modelos de recuperación de documentos similares basados en procesos de generación de consultas y uso motores de búsqueda comerciales.

2.1. Proceso de Fingerprinting de un Documento

Los fingerprints de un documento son pedazos de texto recuperados a partir del contenido de éste. Se define entonces, al proceso de fingerprinting de un documento, como la generación pequeños pedazos de texto que sean capaces de representarlo. Algunos enfoques determinísticos propuestos en [12, 20, 24] son: secuencias de palabras limitadas por un largo de caracteres máximo, términos de mayor frecuencia, oraciones con términos no léxicos (como UFO, OVNI,etc..), términos menos frecuentes, o términos con mayor valor de *tf-idf*. La mayoría de estos enfoques son usados para seleccionar oraciones completas en vez de conjuntos no contiguos de palabras. En [20] se proponen modelos generadores probabilísticos de fingerprints como, funciones generadoras uniformes, generadores por frecuencia o generadoras por *tf-idf*.

2.2. Metabuscadores

Una metabuscador es una máquina de búsqueda que replica la consulta de un usuario en varios motores de búsqueda. Realiza un proceso de parsing de los resultados encontrados y luego un proceso de postfiltro donde se combinan los resultados repetidos. Finalmente tiene una función de scoring para rankear los resultados encontrados y mostrarlos al usuario final.

Como se plantea en [19], los metabuscadores proveen una interfaz unificada donde los usuarios ingresan consultas. El motor propaga la consulta a una lista de buscadores, combina los resultados, entregándole al usuario una única respuesta. Numerosas propuestas de metabuscadores han sido desarrolladas en [3, 22, 23] junto a variadas estrategias de scoring.

En [17] se propone la idea de utilizar directamente los ranking locales entregados por cada motor de búsqueda en la función de scoring. En [3], diferentes enfoques de scoring son analizados, por ejemplo Borda-fuse es un enfoque que considera cada resultado como un voto democrático, también conocido como Borda count. La forma de asignar puntaje

a una *URL* recuperada con Borda count se define a continuación:

$$BordaScore(u) = \sum_{i=1}^{n}(c - rank_i) \tag{2.1}$$

donde u representa a una *URL* encontrada un total de n veces en la posiciones $rank_i$ $\forall i \in [1, \ldots n]$ para las diferentes instancias de búsqueda y c representa el valor de ranking más bajo del cual se recuperan documentos.

Luego en el enfoque weighted Borda-fuse se considera que no todos los buscadores deben ser tratados de igual manera, asignándole factores de confianza a los buscadores. Considerando los mismos parámetros del caso anterior su función de score se define de la siguiente manera:

$$WBordaScore(u) = \sum_{i=1}^{n} w_i(s)(c - rank_i) \tag{2.2}$$

donde $w_i(s)$ es un valor entero $\in [0, \ldots 1]$ que representa la confianza que se tiene a los resultados retornados por el motor de búsqueda s.

Finalmente Bayes-fuse hace uso de la teoría probabilística basada en inferencia Bayesiana para estimar la relevancia de un resultado para una consulta a partir de los resultados retornados por cada buscador.

2.3. Recuperación de Documentos Similares usando Metabuscadores

Se identifican dos investigaciones donde se trata el problema de recuperación de documentos similares[12, 24]. Estos trabajos proponen procesos de fingerprint para representar el documento de entrada en conjuntos de términos relevantes. Ambos hacen uso de arquitecturas de metabuscadores para recuperar listas extensas de candidatos a documentos similares.

Por un lado en [24] se utiliza los snippets[1] de cada resultado comparándolos con el documento de entrada mediante la similitud coseno del modelo vectorial. Por otro lado en [12] se descargan los candidatos a documentos sospechosos y se comparan con el de entrada haciendo uso de algoritmos de similitud de texto como los Patricia trees o los k-gramas.

2.4. Herramientas existentes para la Recuperación de Documentos Similares

Se han encontrado en la Web, diversas soluciones para el problema de recuperación de documentos similares en el contexto de la detección de plagio en documentos digitales. Por ejemplo, la herramienta *eTBLAST*[2] [14], permite comparar documentos sospechosos en colecciones particulares de documentos sobre temas específicos.

Por otro lado, la herramienta *Duplichecker*[3] hace uso de motores de búsqueda comerciales para encontrar los documentos. Sin embargo, no realiza un proceso de combinación de los resultados entregados por distintas instancias de búsqueda en una única respuesta.

[1] Snippet es el pequeño resumen del texto que entrega cada buscador

[2] `http://etest.vbi.vt.edu/etblast3/`

[3] `http://www.duplichecker.com`

Además, el proceso de fingerprinting se basa en tokenizar el texto por frases y convertir cada frase en una consulta.

Finalmente, la herramienta *Plagium*[4] parece hacer uso de procesos elaboradas de generación de consultas, haciendo uso de un único motor de búsqueda (Yahoo!). Lamentablemente, no existe documentación sobre la manera en que genera las consultas ni sobre su función de scoring de resultados.

La herramienta a desarrollar en este trabajo, difiere de las mencionadas anteriormente por los siguientes motivos:

- El proceso de generación de consultas se basa principalmente en la extracción probabilística de términos del texto entregado, en vez de pedazos contiguos de texto como el caso de los fingerprints. Esta característica se basa en la idea de que un documento no contiene necesariamente términos en el mismo orden que sus similares, además muchos motores de búsqueda no consideran el orden de los términos al hacer uso de técnicas como el modelo vectorial.

- Las consultas son enviadas a varios motores de búsqueda y los resultados son combinados automáticamente. Esto se propone como solución al problema del docente planteado en el capítulo 1, evitando que éste tenga enviar consultas a distintos motores de búsqueda y verificar manualmente los documentos encontrados que aparecen varias veces.

[4] `http://www.plagium.com/`

Capítulo 3

Requerimientos de Software y Análisis Preliminar

En este capítulo se describen las principales características del software, identificando los actores y requerimientos del sistema. Posteriormente, los requerimientos son estructurados y refinados mediante la construcción de casos de uso, modelos de análisis y la confección de un diagrama de secuencia. Finalmente se describe la arquitectura general de la aplicación.

3.1. Descripción del Software

Como se mencionó en el capítulo 1, el software se encuentra dentro del proyecto de investigación titulado DOCODE[1]. DOCODE es una herramienta comercial para la detección de plagio de documentos digitales cuyos usuarios meta son los docentes de establecimientos educacionales. La herramienta permite la carga de documentos de trabajos realizados por alumnos en tareas entregadas a cursos completos. Luego compara las similitudes entre todos los documentos entregados para una misma tarea y además con documentos sospechosos recuperados de la Web. El sistema posee los principales atributos de un motor de búsqueda tradicional como un crawler y un indexador.

La herramienta a desarrollar en este trabajo recibe el nombre de DOCODE-lite y es una extensión gratuita del proyecto DOCODE. DOCODE-lite es una herramienta Web que permite a los usuarios ingresar párrafos de los cuales se espera encontrar documentos similares en la Web. Los documentos sospechosos son recuperados a partir de consultas realizadas a motores de búsqueda comerciales y la similitud al párrafo de entrada es calculada al combinar los ranking locales de cada instancia de búsqueda. De esta forma, se rankean los resultados por una estimación de la relevancia y se puede realizar un corte para evitar retornar al usuario documentos no relevantes para el párrafo entregado. Una vez retornados los documentos recuperados al usuario, éste puede evaluar la calidad de los resultados mediante la asignación de puntaje, el cual es guardado en una base de datos. Dicha operación es opcional y permite evaluar posteriormente la calidad predictiva del modelo junto a la relación existente entre la relevancia estimada y la real.

La herramienta DOCODE-lite contribuye al desempeño de DOCODE de las siguientes maneras:

- DOCODE recupera documentos sospechosos a los documentos entregados desde la

[1] `http://www.docode.cl/`

Web seleccionado párrafos desde los documentos y realizando peticiones al metabuscador de DOCODE-lite.

- La base de datos con las evaluaciones de los usuarios de DOCODE-lite permite identificar portales en la Web usados para plagio, éstos sitios pueden ser usados como semilla para el Crawler de DOCODE.
- DOCODE-lite es gratuito y su uso puede ayudar a difundir el producto DOCODE.

3.2. Actores del sistema

- **Usuario**: El usuario interactúa con el sistema por medio de una interfaz donde le entrega el párrafo del cual espera recuperar sus documentos similares. El usuario puede además evaluar la calidad de los resultados retornados.
- **Motor de búsqueda externo**: Representa a cualquier motor de búsqueda comercial al cual la herramienta le envía las consultas generadas y recupera las respuestas retornadas.

3.3. Funciones del Sistema

A continuación se definen los requerimientos funcionales del sistema:

- **R1. Ingreso de párrafo**: El sistema debe proveer una interfaz Web que permita al usuario ingresar el texto del cual se recuperarán sus documentos similares.
- **R2. Generación de consultas**: El sistema debe proveer un mecanismo de generación de consultas con la información relevante del texto entregado.
- **R3. Envío de consultas a motores de búsqueda**: El sistema debe proveer un mecanismo de comunicación con motores de búsqueda comerciales para enviarle las consultas generadas y recuperar sus resultados retornados.
- **R4. Combinación de resultados**: El sistema de proveer un mecanismo para combinar los resultados retornadas en las distintas búsquedas identificando los resultados que apunten a una misma *URL*.
- **R5. Scoring de resultados combinados**: El sistema de proveer un mecanismo para asignar puntaje de similitud entre los documentos recuperados y el texto entregado por el usuario.
- **R6. Generación de reportes**: El sistema debe construir un reporte al usuario que contenga el listado de documentos encontrados rankeados por similitud. Debe contener la *URL* de los documentos encontrados, su título y el puntaje asignado.
- **R7. Asignación de relevancia**: El sistema debe proveer una interfaz que permita al usuario evaluar la relevancia de cada resultado entregado.
- **R8. Registro de ejecución**: El sistema debe registrar el proceso completo de recuperación de documentos similares. Se deben guardar el documento de entrada, los motores de búsqueda utilizados, los documentos encontrados, el puntaje de similitud asignado por el sistema y la relevancia asignada por el usuario. Estos registros deben permitir hacer un análisis posterior de la calidad predictiva del modelo.

3.4. Atributos no funcionales

A continuación se identifican los requerimientos no funcionales que debe tener la aplicación:

1. **Rendimiento**: El sistema debe ser capaz de generar el reporte en una cantidad prudente de tiempo para el usuario, sujeta al hardware disponible.
2. **Orientación a Objetos**: La modelación de las entidades participantes del sistema deberá ser orientada a objetos utilizando patrones de diseño conocidos.
3. **Extensibilidad**: El sistema debe ser fácilmente extensible a distintas estrategias de generación de consultas, a distintas funciones de scoring de resultados y al uso de diversos motores de búsqueda.
4. **Interoperabilidad**: El sistema debe ser implementado bajo estándares de Web Service para que pueda ser utilizado como servicio tanto por la interfaz como por otros sistemas.
5. **Portabilidad**: La interfaz del sistema debe ser plenamente funcional en Firefox 3.0, Microsoft Explorer 7 y superiores.

3.5. Casos de Uso

En esta sección se enuncian los casos de uso del usuario y del sistema. El diagrama se aprecia en la figura 3.1.

1. **Caso de uso**: Ingresar párrafo
   - **Actores**: Usuario
   - **Propósito**: Capturar el párrafo del cual se recuperarán sus documentos similares.
   - **Resumen**: El usuario ingresa en un cuadro de texto el párrafo del cual quiere recuperar documentos similares y comienza su procesamiento mediante un botón de envío.
   - **Tipo**: Primario y esencial
   - **Referencias Cruzadas**: R1
2. **Caso de uso**: Mostrar resultados
   - **Actores**: Usuario, Motor de búsqueda externo
   - **Propósito**: Procesar el párrafo entregado por el usuario y entregarle a éste un reporte con los documentos similares recuperados.
   - **Resumen**: El sistema genera un conjunto de consultas a partir del texto entregado. Las consultas son enviadas a un conjunto de motores de búsqueda externos. Los resultados de todas las instancias de búsqueda son combinados agrupando a los que apuntan a una misma *URL*. El sistema le asigna puntaje a los resultados combinados de acuerdo a una similitud. estimada con el texto entregado. Los resultados son rankeados por puntaje y mostrados al cliente en la interfaz Web.

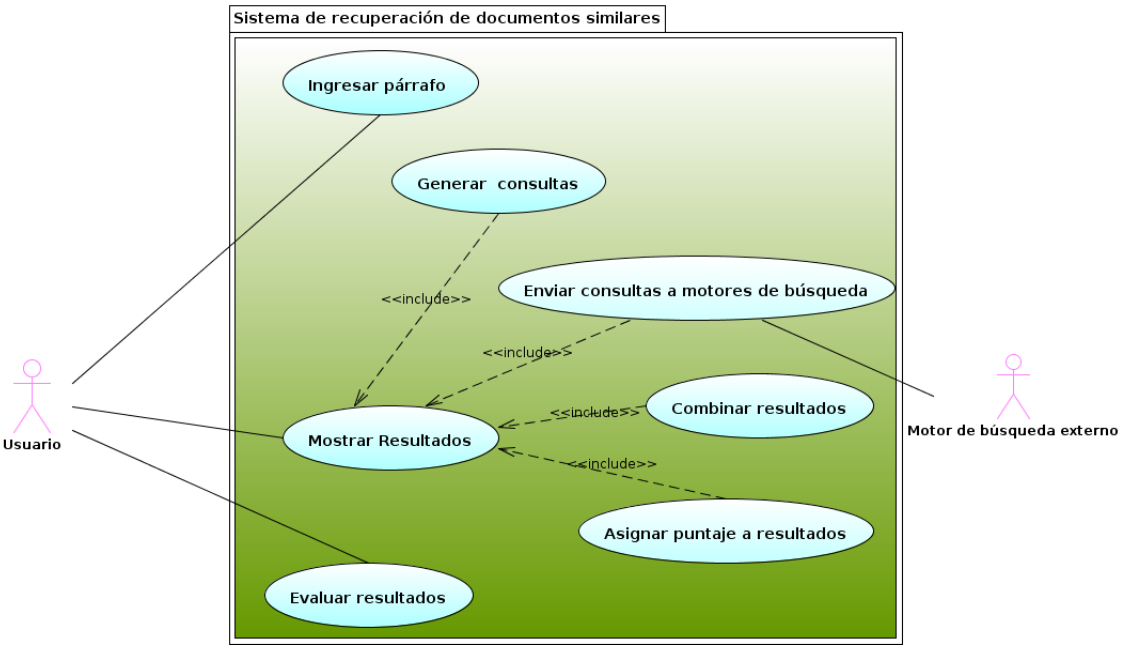

Figura 3.1: Caso de uso del sistema

- **Relaciones de inclusión**:
  - Generar consultas
  - Enviar consultas a motores de búsqueda
  - Combinar resultados
  - Asignar puntaje a resultados
- **Tipo**: Primario y esencial
- **Referencias Cruzadas** R2, R3, R4, R5, R6

3. **Caso de uso**: Evaluar Resultados

   - **Actores**: Usuario
   - **Propósito**: Capturar la evaluación de relevancia del resultado por parte del usuario.
   - **Resumen**: El usuario evalúa la similitud real entre el texto ingresado y el documento recuperado.
   - **Tipo**: Secundario
   - **Referencias Cruzadas** R7, R8

3.6. Modelo de Análisis

Un modelo análisis permite estructurar y refinar los requerimientos de una aplicación. En este modelamiento se definen las abstracciones e interacciones principales del sistema. El análisis es realizado mediante la construcción de clases de análisis, las cuales pueden ser clasificadas en tres categorías:

1. Clases de borde: Son las clases que modelan la interacción entre el sistema y sus actores.
2. Clases entidad: Son las clases que modelan información que sea persistente en la aplicación.
3. Clases de control: Son clases que modelan acciones de coordinación, secuencias, transacciones y control sobre los otros objetos

A continuación se levantan las clases de análisis del sistema.

- **Clases de borde**
  - **Interfaz Web**: Representa la interfaz con la cual el usuario interactúa con el sistema, ya sea para ingresar el texto a procesar o para evaluar los resultados recuperados.
  - **Comunicador motor de búsqueda externo**: Representa la interacción entre el sistema y los motores de búsqueda externos utilizados.
- **Clases entidad**
  - **párrafo**: Representa el texto ingresado por el usuario al sistema.
  - **consulta**: Representa una consulta generada a partir del texto del párrafo.
  - **respuesta**: Representa una respuesta retornada por un motor de búsqueda externo para una consulta particular.
  - **meta-respuesta**: Representa un conjunto de respuestas retornadas por los motores de búsqueda por las consultas realizadas que cumplen las condición de apuntar a una misma *URL*.
  - **reporte**: Representa a una lista de meta-respuestas recuperadas rankeadas por puntaje asignado de similitud con el texto de entrada.
- **Clases de control**
  - **Controlador**: Coordina todo el proceso de recuperación de documentos similares. Llama al generador de consultas y al metabuscador para que envíe las consultas generadas al conjunto de motores de búsqueda externos. Construye el reporte final para el usuario.
  - **Generador de consultas**: Construye el conjunto de consultas a partir del texto entregado.
  - **Metabuscador**: Envía el conjunto de consultas al conjunto de motores de búsqueda externos, agrupa las respuestas retornadas, construye el conjunto de meta-respuestas y les asigna puntaje por similitud al texto entregado por el usuario.

La interacción entre las clases de análisis se aprecia en la figura 3.2:

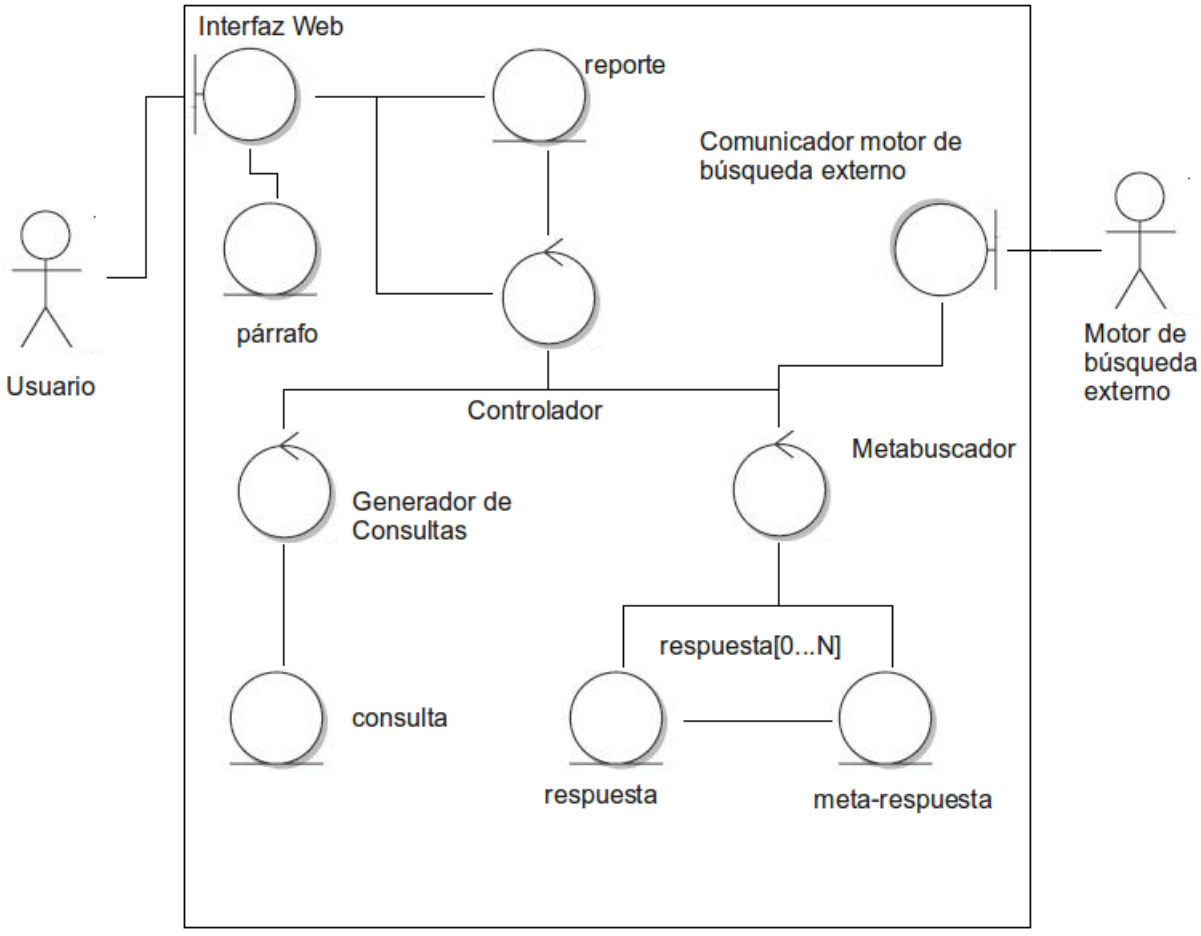

Figura 3.2: Diagrama de Análisis

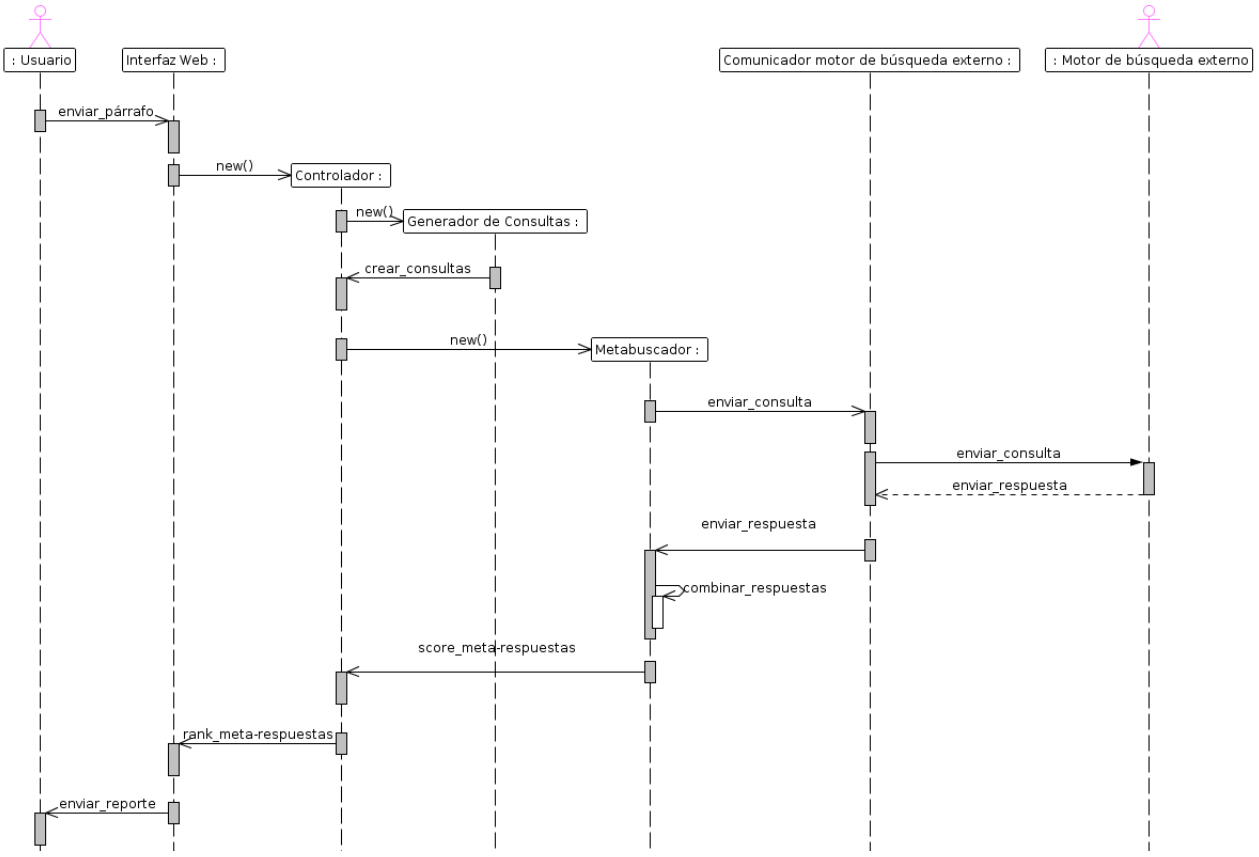

Figura 3.3: Diagrama de secuencia

3.7. Diagrama de secuencia

En esta sección se detalla el flujo de interacción entre las clases de análisis mediante el desarrollo de un diagrama de secuencia (Fig. 3.3). El proceso comienza cuando el *usuario* ingresa un *párrafo* en la *Interfaz Web* de la aplicación. Luego se llama al *Controlador* que a su vez crea una instancia del *Generador de Consultas* la cual retorna el conjunto de *consultas* creado a partir del *párrafo*. El *Controlador* crea una instancia del *Metabuscador* con las *consultas* generadas. Las *consultas* son enviadas al *Comunicador motor de búsqueda externo* quien a su vez las envía al *Motor de búsqueda externo* correspondiente. Las *respuesta* retornadas por el *Motor de búsqueda externo* son retornadas al *Metabuscador* quien las combina identificando a las que apuntan a una misma *URL* para construir *meta-respuestas*. Luego el *Metabuscador* realiza scoring sobre las *meta-respuestas* y las retorna al *Controlador*. El cual las rankea y las retorna a la *Interfaz Web*, quien le muestra al *usuario* un *reporte* final con los documentos similares al *párrafo* recuperados.

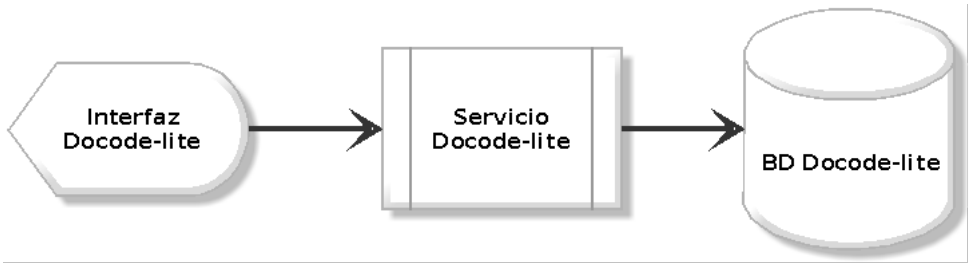

Figura 3.4: Capas de la Herramienta

3.8. Capas del Sistema

El sistema se diseña en base a las tres capas del patrón *layers* presentadas en (Fig. 3.4) y definidos a continuación:

- **Capa Interfaz**: Representa la interfaz de usuario de Docode-lite en la cual el usuario ingresa el párrafo sospechoso. La interfaz consume el servicio contenido en la capa de aplicación.
- **Capa Aplicación**: Representa el servicio Web Docode-lite, el servicio recibe el párrafo y ejecuta todo el proceso definido en el diagrama de secuencia retornado al consumidor una lista de meta-respuestas. Puede ser consumido por la interfaz de DOCODE-lite o por el sistema DOCODE.
- **Capa Datos**: En esta capa se almacena el párrafo de entrada y los resultados recuperados por el servicio. Además, en caso que el usuario evalúe la calidad de un resultado en la interfaz, éste valor es almacenado en la base de datos.

Capítulo 4

Modelo Lógico

En este capítulo se propone un modelo para la recuperación de documentos similares en la Web. Se formalizan las clases de análisis y el diagrama de secuencia propuesto en el capítulo 3 con estrategias para la generación de consultas y para el scoring de resultados finales. El modelo se basa en un proceso secuencial compuesto por las siguientes tareas:

1. Ingreso del párrafo sospechoso por parte del usuario.
2. Generación del conjunto de consultas.
3. Asignación de cada consulta generada a algún motor de búsqueda comercial.
4. Envío de consultas a los distintos motores de búsqueda.
5. Recuperación de los resultados retornados por cada instancia de búsqueda. En esta etapa sólo se recupera la primera página de resultados retornados por cada buscador.
6. Combinación de resultados.
7. Asignación de Puntaje a los resultados combinados.
8. Retorno de resultados ordenados decrecientemente por puntaje.

4.1. Variables del Proceso

A continuación se definen formalmente, las variables que participan en el proceso:

- **Documento sospechoso** : Se llama D al párrafo inicial que entrega el usuario como entrada.
- **Conjunto de Consultas Generadas**: Se llama Q al conjunto de consultas generadas a partir de D.
- **Conjunto de Motores de búsqueda utilizados por el Metabuscador**: Se llama S al conjunto de buscadores utilizados.
- **Respuesta particular para una consulta de un buscador**: Se define como una *queryAnswer* $\omega_{s,q,r}$, a una respuesta entregada por el motor de búsqueda $s \in S$ para la búsqueda $q \in Q$ rankeada en la posición r. Cada $\omega_{s,q,r}$ apunta a un documento de la Web en una URL particular. Es importante considerar que dos elementos $\omega 1_{s1,q1,r1}$ y $\omega 2_{s2,q2,r2}$ con $s2 \neq s1$ o $q2 \neq q1$ pueden perfectamente apuntar a una misma URL.

Puesto que fueron encontrados ya sea por distintas consultas o en distintos motores de búsqueda. De lo anterior, se desprende que para todo r, con s, q fijos, no pueden haber dos elementos $\omega 1_{s,q,r1}$ y $\omega 2_{s,q,r2}$ que apunten a una misma *URL*. Esto se debe a que un motor de búsqueda nunca mostrará dos veces una misma *URL* para una consulta en una misma lista de resultados.

- **Respuesta agregada**: Se define como una *metaAnswer* Ω a un conjunto de objetos *queryAnswer* $\omega_{s,q,r}$ agregados desde distintas instancias de búsqueda que cumplen con la condición de apuntar a una misma *URL*.

A continuación se define lógicamente el modelo.

4.2. Modelo de Lenguaje Hipergeométrico

Para que el Metabuscador pueda encontrar documentos similares al documento original, se deben generar consultas a partir de un Documento D que contengan toda la información relevante de éste. La estrategia se basa en construir a partir de D un modelo de lenguaje generativo M_D capaz de generar un conjunto Q de consultas. Donde cada consulta $q \in Q$ sea una combinación de términos de D.

En este trabajo, se propone un nuevo modelo de lenguaje llamado *Modelo de Lenguaje Hipergeométrico (MLH)*. Esta extensión de modelos de lenguaje se utiliza para la generación de consultas a partir de un documento dado.

El MLH está inspirado en la distribución hipergeométrica multivariada de probabilidades [10]. Esta distribución provee una propiedad de no reposición, la cual consiste en que cada vez que se extrae un elemento, éste no puede volver a ser extraído. Aumentando así, las probabilidades de ocurrencia de los otros términos al realizar una nueva extracción. Esta propiedad se usa en la generación de consultas tomando el supuesto de que un término nuevo aporta más información para una consulta que uno repetido. Además, como los motores de búsqueda permiten consultas con un largo máximo permitido, el hecho de no repetir términos ayuda a generar consultas cortas con un mayor número de términos relevantes de D.

La idea de utilizar una distribución de probabilidades hipergeométrica para la extracción de información relevante sobre colecciones de documentos se basa en una investigación propuesta en [2] .

Se define el vocabulario V extraído a partir del documento D con un tola de m términos diferentes, de la siguiente manera:

$$V = \{t_1, \ldots, t_m\} \tag{4.1}$$

además se tiene un vector $\overrightarrow{\omega}$ con valores positivos w_i para cada término t_i. Estos pesos pueden ser asignados por diferentes enfoques de asignación como *tf*, *idf*, *tf-idf* entre otros [20].

La consulta generada q se modela como una secuencia de punteros al vocabulario V definida de la siguiente manera:

$$q = s_1, \ldots s_n \tag{4.2}$$

Donde cada $s_j \in q$ es un número que toma valores en $\{1 \ldots m\} \in V$. Por la regla de la cadena de probabilidades, se define la probabilidad de generar una consulta q dado el modelo de lenguaje M_D como:

$$P(q|M_D) = P(s_1|M_D)P(s_2|s_1, M_D) \cdot \cdots \cdot P(s_n|s_1, \ldots, s_{n-1}, M_D) \tag{4.3}$$

Formalmente un evento de parada $STOP$ debiese ser incluido, pero se omite sin pérdida de generalidad en el resto de la investigación.

En MLH, la probabilidad de extraer el token s_j dado M_D se estima como:

$$\hat{P}(s_j|M_D) = \frac{w_{s_j}}{||\overrightarrow{w}||_1}, \text{ donde } ||\overrightarrow{w}||_1 = \sum_{i=1}^{m} |w_i| \tag{4.4}$$

luego la probabilidad de extraer un token s_k, dado una consulta acumulada $q = s_1 \ldots s_n$ y M_D se determina en la siguiente expresión:

$$P(s_k|q, M_D) = \begin{cases} 0 & \text{if } \exists s_j \in q, s_k = s_j \\ \frac{w_{s_k}}{||\overrightarrow{w}||_1 - \sum_{j=1}^{n} w_{s_j}} & \text{caso contrario} \end{cases} \tag{4.5}$$

A continuación se muestra un ejemplo de cálculo de probabilidad de generación de una consulta q para un documento D.

4.2.1. Ejemplo de cálculo de probabilidades de MLH

Sea el documento D = (dog dog cat dog dog car cat), su vocabulario extraído V = {car,cat,dog}. Si se crea M_D en base a MLH usando la frecuencia de los términos como enfoque de asignación de pesos, $\overrightarrow{w}$ estaría determinado por $\overrightarrow{w} = \{1, 2, 4\}$, con $||\overrightarrow{w}||_1 = 7$.

La probabilidad de generar la consulta q = (cat dog car) dado M_D se modela usando la regla de la cadena de la siguiente manera:

$$\hat{P}(q|M_D) \quad = \quad \hat{P}(\text{cat}|M_D)\hat{P}(\text{dog}|\text{cat}, M_D) \times \hat{P}(\text{cat}|\text{dog}, \text{car}, M_D)$$

donde

$$\hat{P}(\text{cat}|M_D) = \frac{2}{7} \tag{4.6}$$

$$\hat{P}(\text{dog}|\text{cat}, M_D) = \frac{4}{7-2} = \frac{4}{5} \tag{4.7}$$

$$\hat{P}(\text{car}|\text{dog}, \text{cat}, M_D) = \frac{1}{7-2-4} = 1 \tag{4.8}$$

finalmente, se tiene

$$\hat{P}(q|M_D) = \frac{2}{7} \times \frac{4}{5} \times 1 = \frac{8}{35} \tag{4.9}$$

4.2.2. Algoritmo para la Generación de Consultas de MLH

Una de las propiedades más importantes del MLH es su función generadora de consultas basada en la distribución de probabilidades previamente descrita. El algoritmo procede a extraer términos desde el vocabulario del documento de entrada asignándole mayor probabilidad de extracción a los términos más relevantes. Cumple con la propiedad de no reposición al extraer un nuevo término.

Los parámetros del algoritmo son:

- El documento de entrada D

- El largo mínimo de las consultas a generar *minLength*
- El largo máximo de las consultas a generar *maxLength*
- El enfoque de asignación de pesos *weightApproach*. Este parámetro, flexibiliza la manera en que se asigna el nivel de información aportado a cada términos del documento.

Algorithm 4.2.1: HLM-QueryGenerator

Data: $D, minLength, maxLength, weightApproach$
Result: q

```
1  Initialize q = {};
2  V ← ExtractVocabulary(D);
3  w⃗ ← ExtractWeigthVector(D,weigthApproach);
4  Multinomial m ← CreateMultinomial( w⃗ / ||w⃗||_1 ) ;
5  length ←getRandomNumber(minLength,maxLength) ;
6  i ← 0 ;
7  while i < length and w⃗.size() > 0 do
8      s ← extractRandomElement(m) ;
9      q ← q + V[s] ;
10     w⃗.remove(s) ;
11     V.remove(s) ;
12     m ← CreateMultinomial( w⃗ / ||w⃗||_1 );
13     i ← i + 1;
14 return q;
```

El algoritmo 4.2.1, modela la extracción de cada término con una serie de distribuciones multinomiales, de manera tal que después de cada extracción se cree una nueva distribución que excluya los términos ya extraídos. El largo de las consultas generadas es un número aleatorio uniformemente distribuido entre *minLength* y *maxLength*.

4.2.3. Consideraciones del MLH

El comportamiento de un MLH depende del enfoque de asignación de pesos que se usa. Por ejemplo, considerar el *idf* del modelo vectorial de cada término, aseguraría que los términos que aparecen en menos documentos de alguna colección de referencia tengan mayor probabilidad de aparecer, puesto que al aparecer en menos documentos son más exclusivos y generalmente aportan mayor información. Para aplicar este factor, sería necesario contar con una colección de referencia que contenga todos los términos del vocabulario de los documentos a procesar. Como esto se encuentra fuera del alcance de esta investigación, se propone utilizar un enfoque basado en las frecuencia de los términos en el documento. Una recomendación importante para mejorar la calidad de las consultas generadas, es eliminar de D todos los términos *stopwords* antes de crear M_D. Los términos *stopwords* son todas las palabras del idioma del documento que no aportan información y que tienden a aparecer en todos los documentos, como los artículos y preposiciones. La eliminación de los términos *stopwords* incrementaría entonces, las probabilidades de ocurrencia de los términos que sí aportan información.

4.3. Diseño Lógico del Metabuscador

En esta sección se especifica de qué manera el conjunto de consultas Q generado con el algoritmo 4.2.1 es replicado al conjunto S de motores de búsqueda. El hecho de utilizar más de un motor de búsqueda, se basa en la idea de que la cobertura de la Web es potenciada al unir los índices invertidos de éstos. El modelo propuesto se basa en los siguientes supuestos:

1. El conjunto de consultas generadas Q es una buena representación de D y posee toda su información relevante.
2. Los S motores de búsqueda no consideran el orden de los términos de la consulta, entonces el hecho de que cada consulta q tenga un ordenamiento aleatorio de términos no interfiere en los resultados.
3. Los motores de búsqueda S tienen en conjunto indexada una parte significativa de la Web.
4. Mientras más veces y más arriba en los ranking locales se encuentre una *URL*, mayor será su similitud con D.
5. No todos los motores de búsqueda entregan resultados relevantes con la misma frecuencia.

En este modelo, todas las consultas se generan por un mismo modelo de lenguaje. Lo que implica, que si dentro del conjunto de índices invertidos de los S motores de búsqueda usados se encuentran documentos con una alta similitud a D, éstos debiesen ser recuperados muchas veces y dentro de los primeros resultados.

Una de las características fundamentales de un metabuscador es su función de scoring de resultados. A continuación se propone una función que le asigna puntaje a cada *queryAnswer* recuperada y otra que le asigna puntaje a cada repuesta agregada *metaAnswer*, la asignación de puntaje busca estimar la similitud entre los documentos recuperados y el entregado. Ambas funciones están inspiradas en la ley de Zipf.

4.3.1. Una función de score Zipf-Like

La ley de Zipf, propuesta por *George Kingsley Zipf* en [26], se usa para el análisis de frecuencia de aparición de términos dentro de una colección de documentos. Plantea básicamente que la frecuencia de aparición de un término en una colección es inversamente proporcional a su ranking en una tabla ordenada de frecuencias. Lo que significa que la palabra más frecuente aparece aproximadamente el doble que la segunda y así sucesivamente. Esta ley está relacionada con el principio del mínimo esfuerzo, puesto que nos dice que las personas escriben sus ideas usando muchas veces unas pocas palabras[4, 9, 27]. La ley de Zipf ha sido usada para estudiar fenómenos de diversa índole. Algunos ejemplos son el número de enlaces que salen y entran de una página, o la frecuencia de palabras usadas en las consultas a buscadores Web.

En [16] se plantea que si f representa la popularidad de un elemento, r su ranking relativo, f y r están relacionados bajo la ley de Zipf de la siguiente manera:

$$f = \frac{c}{r^{\beta}} \tag{4.10}$$

donde c es una constante y $\beta > 0$. Si $\beta = 1$, entonces f sigue exactamente la ley de Zipf, si no se dice que sigue una distribución Zipf-like.

En [16] se usa la ley de Zipf para modelar la popularidad de un sitio en la Web. Se usa para afirmar que la frecuencia con que es visitada la r-ésima página más popular es proporcional a $1/r$. Dicha investigación se usa como referencia para postular que la relevancia de la respuesta de un buscador para una consulta de un usuario puede ser modelada también bajo esta ley.

La relevancia de un resultado retornado por un motor de búsqueda representa qué tan bien resuelve la necesidad de información del usuario [15]. En este trabajo se modela como un número entre 0 y 1, proponiéndose de esta manera, que la relevancia de una *queryAnswer* para un usuario siga una distribución Zipf-like. Por lo tanto, la función de *score* para una *queryAnswer* $\omega_{s,r,q}$ se define de la siguiente manera:

$$score(\omega_{s,r,q}) = \frac{c_s}{r^{\beta_s}} \tag{4.11}$$

Donde c_s representa la relevancia promedio de la primera respuesta del motor de búsqueda s y β_s representa una constante de decaimiento de la relevancia de los resultados de s al avanzar hacia los documentos rankeados más abajo. Al usar $\beta_s = 1$, una respuesta que salió segunda tendría la mitad de puntaje que una que salió primera. Luego $\beta_s = 0$ implicaría que todas las respuestas son igual de relevantes.

Como todas las Q consultas son generadas a partir del mismo modelo de lenguaje M_D, se puede decir que todas representan una misma necesidad de información. Entonces si una URL es apuntada por más de una *queryAnswer* la probabilidad de que haya similitud entre el documento D y el documento contenido en la *URL* aumenta. Se propone entonces la siguiente función de *metaScore* para una *metaAnswer* Ω :

$$metaScore(\Omega) = \frac{1}{|Q|} \sum_{\omega \in \Omega} Score(\omega) \tag{4.12}$$

La función de *meta-score* está normalizada por la cantidad de consultas realizadas $|Q|$, de manera tal que el puntaje tenga un valor entre 0 y 1. La única forma de que una *metaAnswer* tenga un score de 1 sería que ésta sea encontrada en todas las instancias de búsquedas en el primer lugar usando $c_s = 1$ para todo s. Los parámetros del modelo, permiten manipular el nivel de confianza que se tiene a los motores utilizados. Además, es importante considerar, que mientras mayor sea la cantidad de consultas realizadas mayor será el nivel de confianza de nuestra solución, pues mayor será la probabilidad de que las consultas generadas contengan toda la información relevante de D.

Un ejemplo de asignación de Score

Sea el documento de entrada D=*El Mercurio Online, el portal con las noticias de Santiago de Chile*. Luego de filtrar el documento con una lista de *stopwords*, se obtiene el conjunto Q generado por M_D compuesto por las consultas $q1$ =*Chile noticias Mercurio*, $q2$ =*Online noticias Santiago* y $q3$ =*portal Mercurio noticias*. De lo anterior, se desprende que $|Q| = 3$. Luego el conjunto de motores de búsqueda S se compone de $s1 = Google$, $s2 = Yahoo!$ y $s3 = Bing$. Donde $c_{s1} = 0{,}95, c_{s2} = 0{,}93$ y $c_{s2} = 0{,}9$. Las constantes de decaimiento β_s valen 0,5 para todo s. Sin pérdida de generalidad se asigna en el proceso de meta-búsqueda las consultas q_i a cada s_i para todo $i \in [1, 2, 3]$. Una vez procesados todos los conjuntos de *queryAnswer* y construido el conjunto de objetos *metaAnswer*, se sabe que en particular una *metaAnswer* Ω apunta a la URL `www.emol.com` y que se compone de los objetos *queryAnswer* $\omega 1_{s1,q1,1}$,$\omega 2_{s2,q2,5}$ y $\omega 3_{s3,q3,2}$. Lo que implica, que la *URL* `www.emol.com` fue encontrada para las tres consultas. Donde al consultar en Google apareció en el primer lugar, en Yahoo! en el quinto y en Bing en el segundo.

El puntaje asignado a cada *queryAnswer* se calcularía usando las estimación Zipf-like (ecuación 4.11) de la siguiente manera:

$$score(\omega 1_{s1,q1,1}) = \frac{0{,}95}{1^{0{,}5}} = 0{,}950 \tag{4.13}$$

$$score(\omega 2_{s2,q2,5}) = \frac{0{,}93}{5^{0{,}5}} = 0{,}416 \tag{4.14}$$

$$score(\omega 3_{s3,q3,2}) = \frac{0{,}9}{2^{0{,}5}} = 0{,}636 \tag{4.15}$$

entonces el puntaje de Ω sería:

$$metaScore(\omega) = \frac{1}{3} \times (\frac{0{,}95}{1^{0{,}5}} + \frac{0{,}95}{5^{0{,}5}} + \frac{0{,}9}{2^{0{,}5}}) = 0{,}670417 \tag{4.16}$$

Método de Ajuste de parámetros del Modelo

Para cada motor de búsqueda s, c_s y β_s son parametrizables. Encontrar sus valores óptimos requeriría de un estudio exhaustivo de la calidad promedio de los resultados de cada buscador junto con el decaimiento de éstos al avanzar en el ranking. Como el comportamiento de los buscadores varía con el tiempo, una estimación formal de los parámetros se encuentra fuera del alcance de este trabajo. En está investigación, los parámetros serán asignados por inspección. Los cuales podrán ser optimizados al analizar los resultados en la parte experimental.

4.3.2. Algoritmo de Metabúsqueda

En esta sección, se detalla cómo la función de scoring se relaciona con el proceso de metabúsqueda de las consultas. Los parámetros del algoritmo son: el conjunto de consultas Q, el conjunto de motores de búsqueda S, los parámetros de confianza de cada buscador c_s y β_s, y un parámetro $requestTimeOut$ que especifica el tiempo máximo de espera para cada petición a realizar.

Una función asignadora se encarga de mapear el conjunto de consultas Q a un conjunto de tuplas (s, c_s^*, β_s^*). De manera tal que para cada consulta $q \in Q$ se sepa a qué motor de búsqueda será enviada y con qué parámetros de confianza se asignará puntaje a sus resultados. La función asignadora se implementa con una estructura de datos de *Diccionario* llamado *queryMap*, que tiene las consultas como llaves y las tuplas como valores. Para cada una de las consultas llaves de *queryMap* se realiza una petición asíncrona al motor de búsqueda s correspondiente. Las consultas se realizan de manera asíncrona para evitar caer en estados *busy-wate*, como se propone en [24].

Luego, para cada petición efectiva realizada, el motor de búsqueda respectivo responde un objeto *resultPage* que es una página con los resultados. Cada *resultPage* se almacena en una lista llamada *pages*. La cardinalidad de *pages*, representa la cantidad de peticiones efectivas realizadas. El parámetro *requesTimeOut* es usado como tiempo máximo de espera para cada petición asíncrona. En caso de que se exceda este tiempo, el proceso de petición es anulado.

El algoritmo recupera los objetos *queryAnswer* de cada elemento de la lista *pages*. A cada *queryAnswer* le asigna puntaje usando la función 4.11 normalizado por la cardinalidad de *pages*.

Una nuevo diccionario llamado *urlMap* es usado para mapear cada documento de la Web recuperado a su respectiva *metaAnswer*. En este diccionario se mezclan todas los

objetos *queryAnswer* que apuntan a un mismo documento en la Web. El diccionario usa *URLs* como llaves y objetos *metaAnswer* como valores.

Entonces el algoritmo itera por todos los objetos *queryAnswer* recuperados y busca en el diccionario *queryMap* la *URL* apuntada. En caso de que el diccionario no contenga la *URL*, se crea un nuevo objeto *metaAnswer* que apunta a la *URL* y que tenga como puntaje el puntaje del objeto *queryAnswer*. Luego la *URL* es ingresada como nueva llave a *queryMap* con la nueva *metaAnswer* como valor. En caso de que la *URL* exista en el diccionario, se recupera su *metaAnswer* y se le agrega a su puntaje el puntaje de la *queryAnswer* de manera de ir calculando *metaScore* de manera incremental.

Finalmente se recuperan todos los objetos *metaAnswer* de *queryMap* y se ordenan decrecientemente por su puntaje. La lista de objetos *metaAnswer* ordenada es retornada. El pseudo-código del algoritmo de metabúsqueda se presenta a continuación:

Algorithm 4.3.1: MetaSearch

Data: $Q, S, c_s^*, \beta_s^*, requestTimeOut$
Result: Ω^* $metaAnswers$

Initialize $Dictionary < url, \Omega > urlMap$;
Initialize $Dictionary < q, (s, c_s, \beta_s) > queryMap$;
Initialize $resultPage^*$ $pages \leftarrow \{\}$;
$queryMap \leftarrow assign(Q, S, c_s^*, \beta_s^*)$;
foreach $q_i \in Q$ **do**
    $pages.add(asyncRequest(q_i, queryMap.get(q_i)))$;
$wait$(response of all requests or $requestTimeOut$);
foreach $page \in pages$ **do**
    $queryAnswerSet \leftarrow page.getAnswers()$;
    foreach $\omega \in queryAnswerSet$ **do**
        $\omega.setScore(\frac{\omega.c_s}{(\omega.r)^{(\omega.\beta_s)}})$;
        if $urlMap.contains(\omega.getLink)$ **then**
            Ω $metaAns \leftarrow urlMap.get(\omega.getLink)$;
            $score \leftarrow metaAns.getScore + \frac{\omega.getScore}{pages.size}$;
            $metaAns.setScore(score)$;
        else
            Initialize $metaAnswer$;
            $metaAnswer.setLink(\omega.getLink)$;
            $metaAnswer.setScore(\frac{\omega.getScore}{pages.size})$;
            $urlMap.add(\omega.getLink, metaAnswer)$;
$metaAnswerSet \leftarrow urlMap.getValues()$;
$rankElementsByScore(metaAnswerSet)$;
return $metaAnswerSet$;

Capítulo 5

Diseño e Implementación de Software

En este capítulo se detalla el diseño y la implementación de la herramienta realizada en base a los componentes definidos en el capítulo 3 y los algoritmos planteados en el capítulo 4. El diseño es elaborado mediante la elaboración de una arquitectura, un diseño orientado a objetos, una interfaz de usuario y un modelo de datos para almacenar los registros de uso del sistema para su posterior evaluación. Finalmente, la implementación realizada es explicada mediante las principales librerías y herramientas utilizadas.

5.1. Arquitectura

Se propone una arquitectura para la herramienta (Fig. 5.1) formada por componentes, basada en el diseño de capas elaborado en el capítulo 3. Cada componente se responsabiliza del control de las tareas necesarias para resolver el problema de recuperación de documentos similares en la Web. Para permitir interoperabilidad con otras aplicaciones se plantea un diseño orientado a servicios basado en Web Services. De esta manera el servicio puede ser embebido en otras aplicaciones. En este caso se usa como cliente del servicio a la interfaz Web que se le da al usuario. El servicio en sí, es el proceso de recuperación de documentos similares.

La interacción entre los componentes se explica a continuación: El proceso comienza con el párrafo de entrada que entrega el usuario en la interfaz, y termina con una lista de documentos de la Web rankeados por similitud como salida. La entrada es convertida en un conjunto de consultas, generadas por el proceso aleatorio que le asigna mayores probabilidades de ocurrencia a los términos más relevantes. Luego las consultas son enviadas en paralelo a una lista customizable de motores de búsqueda. Finalmente los documentos de la Web recuperados son rankeados por la estrategia propuesta en el modelo y retornados al cliente del servicio, que en este caso es el usuario de la interfaz. Se permite también en la interfaz, que el usuario evalúe la calidad de los resultados encontrados, a este proceso se le conoce como *relevance feedback*. Tanto los párametros del proceso, los documentos recuperados como las evaluaciones del usuario son guardados en una base de datos, de manera de poder evaluar la efectividad de la herramienta a posteriori.

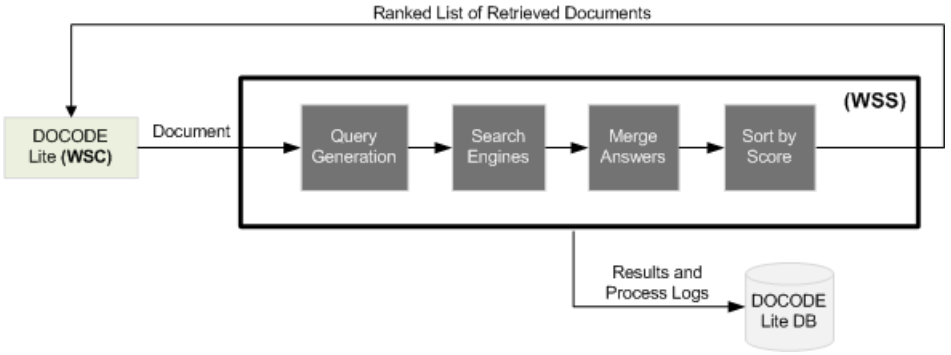

Figura 5.1: Arquitectura de DOCODE-lite

5.1.1. Marco de Referencia J2EE

El marco de referencia J2EE [1] es un marco referencial para el desarrollo de aplicaciones empresariales en el lenguaje de programación Java. Como aprecia en la figura 5.2[1], el marco se basa en una arquitectura de capas similar a la planteada en el capítulo 3. A continuación se explica la arquitectura de la herramienta basada en las capas propuestas por el marco de referencia. Mayores detalles sobre la implementación de cada una de las capas, se encuentran en la sección 5.6.

- Capa Cliente: En esta capa se consideran los sistemas externos que consumen el servicio provisto por la aplicación. En particular el software DOCODE hará uso del motor de metabúsquedas DOCODE-lite para recuperar documentos similares a los documentos que procesa. La manera de comunicarse será por medio de mensajes *SOAP/WSDL* con el servicio Web de la capa de negocios de la herramienta.

- Máquina servidor J2EE: El servidor de la aplicación contiene por un lado una capa Web JSP[2] para presentar un metabuscador de uso gratuito para los usuarios y una capa negocio basada en un servicio Web que recibe las peticiones vía mensajes SOAP y ejecuta el proceso principal. El servicio principal es llamado mediante un clase en *Java* que controla todo el proceso de recuperación de documentos similares. La comunicación con el servidor de bases de datos se realiza mediante un mapeo a objetos provisto por una capa de persistencia.

- Servidor de Base de Datos: Se usa un motor de bases de datos *MySQL*. Esta capa no es fundamental para el servicio en sí, principalmente se encarga de registrar todos los datos generados en el proceso para permitir la evaluación predictiva del modelo. Los sitios a los cuales los usuarios evalúan con altos niveles de similitud a los párrafos entregados, se utilizan como semillas para el Crawler del proyecto DOCODE. Esto permitiría al proyecto, poseer una muestra representativa de los portales Web usados en el plagio de documentos digitales.

[1] `http://www.exforsys.com/tutorials/j2ee/j2ee-overview.html`
[2] Java Server Pages

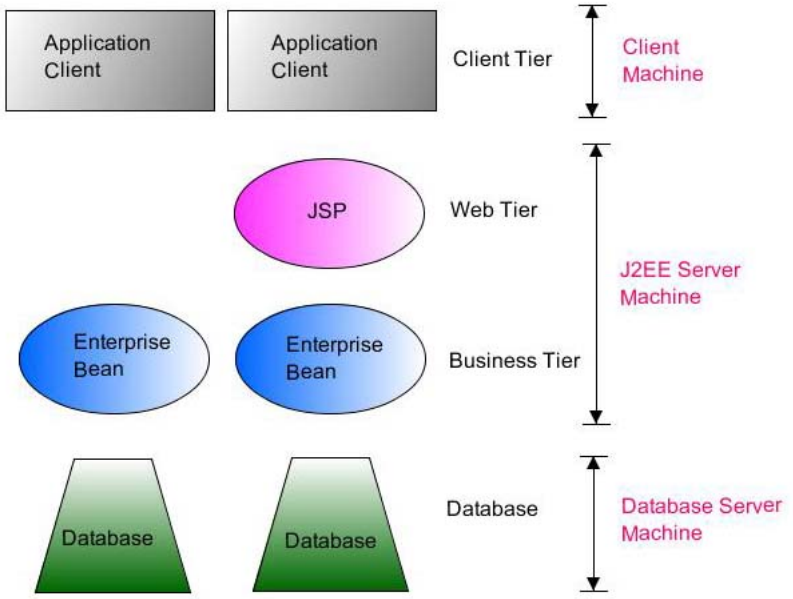

Figura 5.2: Marco de Referencia J2EE

5.2. Diseño Orientado a Objetos

Para la capa de negocios de la aplicación, se propone un diseño orientado a objetos. La principal razón de orientar el diseño bajo este paradigma, es permitir que el sistema pueda ser extendido a distintas estrategias de generación de queries y de scoring de resultados. Además se espera del diseño permita agregar nuevos motores de búsqueda a utilizar sin necesidad de modificar la estructura del código. Se contempla en el diseño, la generación de consultas mediante MLH(4.2.1) y una estrategia alternativa basada en construir queries a partir de las oraciones del párrafo. Además para el scoring de resultados finales se incluyen además del modelo propuesto Zipf-like, estrategias como la similitud vectorial entre el párrafo y los snippets recuperados y el método Weighted Borda Fuse.

Estos modelos alternativos incluidos en el diseño tanto para la generación de consultas como para el scoring de resultados, pretender facilitar el desarrollo de investigaciones futuras al trabajo realizado.

5.2.1. Diseño de Clases

A continuación se presentan las principales clases de la herramienta. El diagrama de clases puede ser apreciado en (Fig. 5.3).

- *RetrieveService*: Controla y ejecuta el servicio de recuperación de documentos similares. Recibe el documento a buscar, la función de *score* a usar, la cantidad de resultados a entregar y una lista de objetos *ServiceParameter* que definen donde buscar y como generar las consultas.
  - Atributos:
    - *private Collection<ServiceParameter> serviceParameters*: Conjunto de parámetros del servicio.

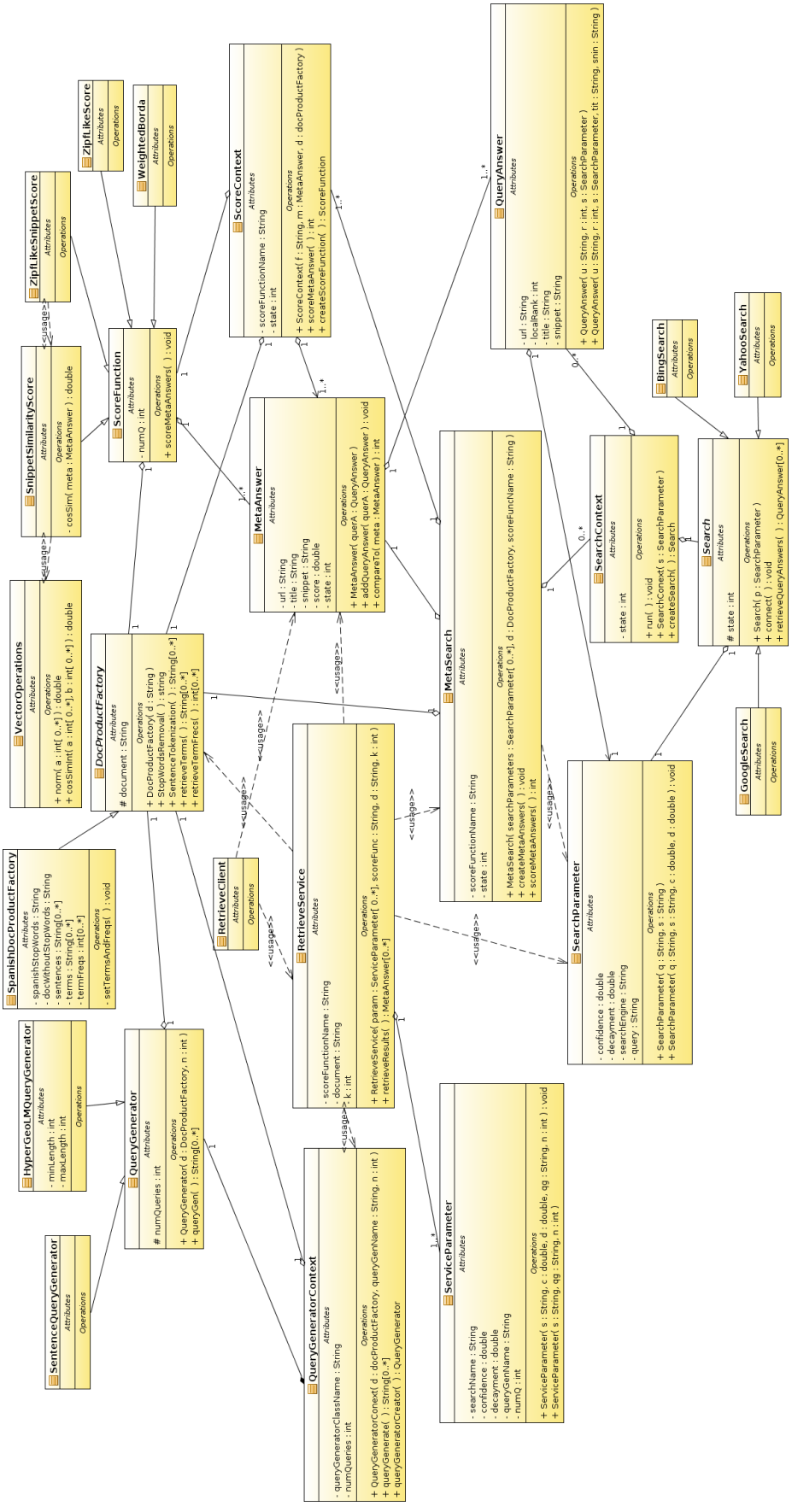

Figura 5.3: Diagrama de Clases

- *private String scoreFunctionName*: Estrategia de asignación de puntaje de similitud entre las respuestas agregadas y el documento de entrada.
- *private String document*: Documento de entrada.
- *private int k*: Cantidad máxima de resultados a recuperar.
- *private Collection<MetaAnswer> metaAnswers*: Conjunto de respuestas agregadas encontradas.

Operaciones:

- *void executeService()*: Ejecuta el servicio de recuperación de documentos similares. Parte llamando al proceso de generación de consultas y luego con las consultas generadas al proceso de metabúsqueda. Finalmente ordena decrecientemente los resultados obtenidos por score.

- *ServiceParameter*: Representa una unidad de información necesaria para que el servicio sepa el número de consultas a generar usando alguna estrategia parametrizable, sabiendo además en que motor de búsqueda enviar las consultas con la información respectiva de cuanta confianza se tiene en los resultados que éste entregue.

  - Atributos
    - *private String searchName*: Nombre del motor de búsqueda a usar.
    - *private double confidence*: Valor entero $\in [0, 1]$ que representa la relevancia promedio de la mejor respuesta del motor de búsqueda a utilizar.
    - *private double decayment*: Constante de decaimiento de la relevancia de los resultados entregados por el motor de búsqueda mientras se baja en el ranking.
    - *private String queryGenName*: Nombre de la clase que usada como estrategia generadora de consultas.
    - *private int numQ*: Cantidad de consultas a generar usando los parámetros del objeto.

- *DocProductFactory*: Es una fábrica abstracta de productos relacionados con el texto del documento de entrada. Provee operaciones de borrado de las palabras que no aportan información *stopwords*, tokenización por frases, extracción del vocabulario del documento y extracción de las frecuencias de aparición de los términos de éste.

  - Atributos:
    - *protected String document*: Documento a analizar.
  - Operaciones:
    - *public abstract String StopWordsRemoval()*: Retorna el documento con las palabras que no aportan información eliminadas.
    - *public abstrac String[ ] SentenceTokenization()*: Retorna un arreglo con las frases del documento.
    - *abstract String[ ] retrieveTerms()*: Retorna un arreglo con el vocabulario del documento.
    - *public abstract int[ ] retrieveTermFrecs()*: Retorna un arreglo con la frecuencia de cada término del vocabulario en el documento.
  - Implementaciones:

    - SpanishDocProductFactory: Implementa *DocProductFactory* usando *stopwords* en español. Implementa las operaciones usando expresiones regulares y el framework *Apache Lucene*.

- *QueryGenerator*: Representa una estrategia abstracta para generar queries a partir de un documento. Todas las implementaciones hacen uso de una fábrica *DocProductFactory* para realizar la generación. Entrega al cliente la lista de consultas generadas.
    - Atributos
        - *protected int numQueries*: Número de consultas a construir.
        - *protected DocProductFactory docFac*: Objeto fábrica con el documento de entrada, sus productos son utilizados en el proceso de generación de consultas.
    - Operaciones:
        - *public Collection<String> queryGen()*: Retorna las consultas generadas en una colección de String.
    - Implementaciones:
        - *HyperGeoLMQueryGenerator*: Asigna probabilidades a los términos según sus frecuencias de aparición. Luego realiza una extracción aleatoria de términos según las probabilidades asignadas. La extracción de términos es sin reposición. Los términos *stopwords* son eliminados.
        - SentenceQueryGenerator: Crea Strings a partir de una tokenización por frases.

- *QueryGeneratorContext*: Crea una instancia de *QueryGenerator* y la asigna a alguna de sus implementaciones mediante reflexión. Recibe como parámetros el nombre de la implementación a usar y la cantidad de queries a generar.
    - Atributos
        - *private String queryGeneratorClassName*: Nombre de la clase que implementa la estrategia de generación de consultas a usar.
        - *private QueryGenerator queryGen*: Estrategia con la cual se generan las consultas.
        - *private int numQueries*: Número de consultas a generar.
        - *private DocProductFactory docFac*: Fábrica de productos de texto, instanciada con el documento de entrada, utilizada en el proceso de generación de consultas.
    - Métodos
        - *private QueryGenerator queryGeneratorCreator()*: Método de factoría que crea una instancia de *QueryGenerator* según la implementación especificada en *queryGeneratorClassName*.
        - *public Collection<String> queryGenerate()*: Retorna el conjunto de consultas generadas por la instancia de *QueryGenerator* creada mediante el método de factoría.

- *MetaSearch*: Encargado de controlar y ejecutar las múltiples búsquedas a realizar. Una vez realizadas las búsquedas procede a agregar los resultados (*QueryAnswer*) que apuntan a una misma *URL* en objetos *MetaAnswer*, luego les asigna puntaje

según alguna estrategia de score. Recibe una lista de *SearchParameter* como parámetros de búsqueda, el nombre de la función de score y la fábrica de productos del documento de entrada.

- Atributos:
  - *private Collection<SearchContext> searchContextList*: Conjunto de contextos de búsquedas a realizar.
  - *private Collection<MetaAnswer> metaAnswers*: Conjunto de respuestas agrupadas por URL.
  - *private String scoreFunctionName*: Nombre de la clase que implementa una estrategia para asignar puntaje a las respuestas agregadas.
  - *private DocProductFactory docFac*: Contiene el texto del documento de entrada.
- Métodos
  - *public int runSearchs()*: Ejecuta concurrentemente el conjunto de contextos de búsquedas.
  - *public void createMetaAnswers()*: Agrupa las respuestas retornadas por cada búsqueda en *metaAnswers*.
  - *public int scoreMetaAnswers()*: Crea un objeto *ScoreContext* usando las variables de instancia correspondientes como parámetros y lo utiliza para asignarle puntaje a *metaAnswers* con la estrategia respectiva.

- *SearchParameter*: Representa un parámetro de búsqueda. Contiene la *query*, el motor de búsqueda y sus factores de confianza. Estos atributos son equivalentes a los de *ServiceParameter*.

- *Search*: Representa el proceso abstracto de realizar una búsqueda en un motor de búsqueda. Se construye a partir de un *SearchParameter*. Esta encargado de conectarse con el motor de búsqueda, enviar la consulta y recuperar los resultados que éste responda.

  - Atributos:
    - *protected SearchParameter parameter*: Parámetro de búsqueda.
  - Métodos:
    - *public int connect()*: Realiza la conexión al motor de búsqueda especificado en *parameter*.
    - *public Collection<QueryAnswer> retrieveQueryAnswers()*: Recupera las respuestas de la consulta y construye un conjunto de objetos *QueryAnswer*.
  - Implementaciones:
    - YahooSearch: Implementa la búsqueda al motor de búsqueda Yahoo! usando la *Yahoo Boss Search Api*.
    - GoogleSearch: Implementa la búsqueda a Google.
    - BingSearch: Implementa la búsqueda a Bing.

- *SearchContext*: Encargado de crear la instancia de la implementación correspondiente de *Search* dentro de *MetaSearch*. La creación la realiza por medio de la información del objeto *SearchParameter*, que le entrega el nombre del motor de búsqueda a usar. Luego como los nombres de las implementaciones calzan con los nombre

de los buscadores, hace uso de reflexión para asignar la implementación de *Search* correspondiente, a la instancia de éste. Además extiende la clase *Thread* para que *MetaSearch* pueda ejecutar las búsquedas de manera concurrente.

- Atributos:
  - *private SearchParameter parameter*: Parámetro de búsqueda.
  - *private Search search*: Objeto de búsqueda.
  - *private Collection<QueryAnswer> queryAnswers*: Conjunto de respuestas retornadas por el motor de búsqueda a la consulta especificados en *parameter*.
- Métodos:
  - *private Search searchCreator()*: Método de factoría para crear un objeto *Search* usando la implementación y los parámetros especificados en *parameter*.
  - *void run()*: Usa el método de factoría para crear un objeto *Search* asignado a su implementación respectiva,luego realiza la conexión del objeto y recupera los objetos *QueryAnswer* respectivos. Este método puede ser ejecutado de manera concurrente.

- *QueryAnswer*: Representa la respuesta entregada por un motor de búsqueda a una consulta particular. Agrega un objeto *SearchParameter* que contiene información de la consulta y el motor de búsqueda que la generaron. Contiene además la URL del documento que apunta, el ranking local donde apareció, el título del documento que apunta y un resumen de éste (sinppet).

  - Atributos:
    - *private String url*: URL del documento encontrado.
    - *private int localRank*: Posición en donde el resultado fue encontrado.
    - *private String title*: Título del documento encontrado.
    - *private String snippet*: Texto resumen del documento encontrado.
    - *private SearchParameter parameter*: Parámetro de búsqueda con que se encontró el documento.

- *MetaAnswer*: Representa una respuesta agregada. Agrega entonces una lista de objetos *QueryAnswer* que cumplen la condición de apuntar a una misma URL. Contiene entonces toda la información que contienen sus objetos *QueryAnswer* como URL, título y snippet. Posee además un atributo *score* que representa el puntaje de similitud que posee el documento apuntado por la *MetaAnswer* y el documento inicial de búsqueda.

  - Atributos:
    - *private Collection<QueryAnswer> queryAnswers*: Conjunto de respuestas apuntando a una misma URL.
    - *private String url*: URL apuntada por el conjunto de respuestas.
    - *private String snippet*: Concatenación de snippets de todos las respuestas agregadas.
    - *private double score*: Puntaje asignado por una función de score que representa la similitud entre el documento apuntado y el documento de entrada.

Métodos:

- *public int compareTo(MetaAnswer meta)*: Compara el puntaje del objeto con otro de la misma clase. Se usa para ordenar las respuestas por similitud al documento de entrada.

- *ScoreFunction*: Representa una función abstracta que asigna puntajes de similitud a un conjunto de objetos *MetaAnswer* con un documento de entrada. Agrega un objeto *DocProductFactory* si es que alguna de sus implementaciones requieren operaciones sobre el documento de entrada.

  - Atributos:
    - *protected Collection<MetaAnswer> metaAnswers*: Conjunto de respuestas agregadas a las que se les debe asignar puntaje en un atributo *score*.
    - *protected DocProductFactory docFac*: Instancia de fábrica de productos del documento de entrada, que provee productos que pueden ser utilizados en el proceso de scoring.
    - *protected int numQ*: Número efectivo de consultas realizadas en el proceso de metabúsqueda.
  - Operaciones:
    - *void scoreMetaAnswers()*: Asigna puntaje a todos los objetos *MetaAnswer* de *metaAnswers*. Esta función debe ser implementada por toda implementación de la clase.
  - Implementaciones:
    - *ZipfLikeScore*: Asigna puntaje usando los objetos *QueryAnswer* que agrega cada *MetaAnswer* usando los ranking locales y los factores de confianza de los buscadores. El puntaje lo asigna usando una función basada en la ley de Zipf.
    - *WeightedBorda*: Toma cada *QueryAnswer* de la *MetaAnswer* como un voto democrático para asignar puntaje. Los votos van ponderados por el factor de confianza en el motor de búsqueda que la encontró.
    - *SnippetSimilarityScore*: Usa la similitud vectorial coseno entre el vector de frecuencias de términos del documento de entrada y el vector del snippet de la *MetaAnswer*.
    - *ZipfLikeSnippetScore*: Asigna puntaje como una combinación lineal entre el puntaje *Zipf-Like* y la *Snippet Similarity*.

- *ScoreContext*: Es agregado por *MetaSearch* para asignarle puntaje a los objetos *MetaAnswer* encontrados mediante la estrategia de score entregada por el cliente. Contiene un objeto *ScoreFunction* que se asigna a la implementación respectiva mediante un método de factoría.

  - Atributos:
    - *private String scoreFunctionName*: Nombre de la estrategia de scoring a usar.
    - *private Collection<MetaAnswer> metaAnswers*: Conjunto de respuestas agregadas a las que se les debe asignar puntaje.
    - *private DocProductFactory docFac*: Contiene el documento de entrada y sus productos pueden ser usados en las estrategias de scoring.

- *private int numQ*: Número efectivo de consultas realizadas.
- *private ScoreFunction scoreFunction*: Estrategia de scoring a usar.

- Operaciones:
  - *private ScoreFunction scoreCreator()*: Crea un objeto *ScoreFunction* usando la implementación especificada en *scoreFunctionName*.
  - *public int scoreMetaAnswers()*: Usa el método de factoría para instanciar *scoreFunction* para luego ejecutar su operación *scoreMetaAnswers* y así asignarle puntaje a *metaAnswers*.

- *VectorOperations*: Conjunto de operaciones estáticas para realizar cálculos con vectores. Se usan para obtener similitudes vectoriales de texto.

  - Operaciones:
    - *public static double norm(int[] a)*: Retorna la norma euclidiana de un arreglo de *Integers*.
    - *public static double cosSimInt(int[] a, int[] b)*: Retorna la similitud coseno entre dos arreglos de *Integers*.

5.3. Patrones de Diseño

El diseño orientado a objetos permite principalmente representar el problema en cuestión en base a la identificación de tipos, subtipos y sus relaciones existentes. Uno de los objetivos principales de este enfoque, es que el Software sea reusable, extensible y mantenible a nivel de código [11]. Los patrones de diseño, se definen en [8] como descripciones para comunicar objetos y clases orientadas a resolver problemas de diseño en contextos particulares. A grandes rasgos, entregan una buena solución a problemas de diseño encontrados en repetidas ocasiones. A continuación se enuncian cuatro patrones de diseño utilizados en el diseño propuesto junto a la explicación de la manera en que fueron incluidos en el diseño de la herramienta.

5.3.1. Strategy

Strategy es un patrón de comportamiento que define una familia de algoritmos, los encapsula y los hace intercambiables [8]. La estructura del patrón se basa en un *contexto*, que agrega una *estrategia* abstracta, la cual es implementada por las distintas estrategias. En el diseño propuesto, el patrón fue utilizado en las siguientes tres ocasiones.

Generación de consultas

En la generación de consultas, se define la clase *QueryGenerator* como abstracta. Donde el método *queryGen* es implementado por cada estrategia de generación de consultas (*HyperGeoLMQueryGenerator*, *SentenceQueryGenerator*) que hereda de *QueryGenerator*. Luego, el contexto de generación de consultas *QueryGeneratorContext* agrega un objeto *QueryGenerator* y decide dinámicamente que estrategia utilizar según sus variables de instancia.

Uso de distintos motores de búsqueda

Se define la clase *Search* como abstracta. Tanto la conexión al motor de búsqueda como el proceso de parsing de resultados se definen como operaciones abstractas (*connect* y *retrieveQueryAnswers*). Luego cada implementación de *Search*, (*GoogleSearch*, *YahooSearch* y *BingSearch*), implementan la estrategia de buscar en el motor de búsqueda y de recuperar las respuestas respectivas. El contexto de búsqueda es implementado en *SearchContext* que agrega una instancia de *Search* y asigna el objeto dinámicamente a la implementación correspondiente.

Scoring de respuestas agregadas

Se define la clase *ScoreFunction* como abstracta. El método *scoreMetaAnswers* es implementado por cada uno de sus hijos (*ZipfLikeScore*, *WeightedBordaScore*, *SnippetSimilarityScore* y *ZipfLikeSnippetScore*). Luego el contexto (*ScoreContext*) agrega un objeto *ScoreFunction* que dinámicamente es asignado a la implementación que corresponda a la estrategia de scoring a usar.

5.3.2. Factory Method

Factory Method es un patrón creacional, que define una interfaz para crear un objeto, dejando a las subclases decidir qué clase instanciar. Su estructura general es de una clase creadora provista de un método de factoría que retorna un objeto producto. El objeto producto es abstracto y puede tener varias implementaciones. El método factoría crea un objeto producto usando alguna implementación esperada. Luego, para decidir qué implementación usar existen métodos de factoría parametrizados.

En el diseño propuesto, se usan métodos de factoría en todas las ocasiones que se usa el patrón *Strategy*. Cada contexto donde se usa el patrón *Strategy* provee un método creador del objeto correspondiente. Luego, cada método creador crea el objeto con la implementación correspondiente a la información que contiene el estado del contexto. La creación de objetos de cada método creador se lleva a cabo haciendo uso de reflexión, como se propone en [7]. La reflexión permite crear objetos conociendo el nombre de su clase, información contenida en el estado de todos los contextos donde se usa el patrón *Factory Method*.

5.3.3. Abstract Factory

Abstract Factory es otro patrón creacional, que provee una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Su estructura general es de una clase *AbstractFactory* cuyas implementaciones implementan métodos de creación de productos. Un producto es generalmente una clase abstracta, entonces cada método de creación de las implementaciones de *AbstracFactory* debiese crear un producto específico.

En el diseño propuesto, se define *DocProductFactory* como una fábrica abstracta de productos creados a partir del texto del documento de entrada. Los productos son concretos y no requieren ser reimplementados, pues son principalmente arreglos de Strings. Los productos son arreglos de frases, de términos o el mismo texto entrada con un borrado de *stopwords*. La fábrica es implementada en *SpanishDocProductFactory* orientada a trabajar con texto en español.

5.3.4. Iterator

Iterator es un patrón de comportamiento que provee una manera de acceso secuencial a elementos agregados sin conocer su representación. Para el caso de *Java*, cualquier implementación de la interfaz *Iterable*$<T>$ permite obtener un iterador.

En el diseño propuesto, *MetaSearch* agrega una colección de objetos *QueryAnswer* los cuales itera usando el iterador de la clase *Collection* que implementa *Iterable* para construir la colección de objetos *MetaAnswer* en el método *createMetaAnswers*. Por otro lado, cada implementación de *ScoreFunction* agrega una colección de objetos *MetaAnswer*. Para asignar el puntaje, utiliza el iterador provisto por *Collection* para iterar sobre los objetos y aplicar su estrategia de scoring.

5.4. Interfaz de Usuario

Cuando un usuario accede a un sistema de recuperación de información, éste por lo general posee una compresión difusa de como resolver su necesidad de información por medio del sistema. De hecho, en [4] se cataloga al proceso de búsqueda de información por parte de un usuario dentro de una interfaz como impreciso. Por este motivo, es de suma importancia que la interfaz facilite al usuario la formulación de consultas y la comprensión de los resultados recuperados.

Para este trabajo, se propone una interfaz de usuario, basada en un cuadro de texto donde el usuario ingrese un párrafo como entrada. La petición de búsqueda se inicializa mediante un botón como se aprecia en la figura 5.4. Luego los resultados son desplegados en la misma página bajo del cuadro de texto, permitiendo al usuario realizar nuevas búsquedas en la misma interfaz.

Para cada resultado, se muestra el título del documento, la URL donde éste se encuentra, y los motores de búsqueda donde fue encontrado. Pueden ser evaluados por el usuario con estrellas, las estrellas van desde 0 hasta 5 y se asignan haciendo click con el mouse. Representan la relevancia que asigna el usuario a éstos. La evaluación de los resultados se aprecia en la figura 5.5.

Figura 5.4: Cuadro de texto donde el usuario inserta el párrafo

Figura 5.5: Evaluación de un Resultado

5.5. Modelo de Datos para el almacenamiento de Ejecución

En está sección se propone un modelo de datos relacional encargado de registrar toda la ejecución del proceso de recuperación de documentos, con sus parámetros y resultados. En el modelo se registra el párrafo de entrada, los parámetros utilizados para la generación de consultas, los motores de búsqueda utilizados con sus respectivos parámetros de confianza, los documentos de la Web encontrados junto a su puntaje asignado por el sistema y el relevance feedback entregado por el usuario.

Como se mencionó en el capítulo 3, la idea de registrar los datos generados durante el proceso se orienta a poder evaluar continuamente la calidad predictiva del modelo y poder optimizar así los parámetros de éste. Además el conocimiento entregado por los usuarios permite identificar sitios en la Web frecuentemente usados para el plagio de documentos, los cuales podrían ser usados como páginas semilla para el Crawler del proyecto DOCODE.

5.5.1. Tablas del Modelo

A continuación se definen las tablas del modelo de datos juntos a sus atributos y relaciones. El modelo se aprecia en la figura 5.6.

- *ProcessInstance*: Representa la instancia de ejecución del servicio por parte de un cliente.
  - Atributos:
    - *idProcessInstance (INT)*: Identificador del proceso.
    - *execTime (TIMESTAMP)*: Tiempo de ejecución del proceso.
    - *k (INT)*: Cantidad máxima de documentos recuperados por instancia.
    - *document (TEXT)*: Texto entregado por el usuario con el párrafo sospechosos.

  Relaciones:
    - *idScoringFunction (INT)*: Identificador de la función de scoring de resultados utilizado, se modela con una relación uno a uno con la tabla *ScoringFunction*

- *ProcessParameter*: Representa una unidad de parámetro del servicio.
  - Atributos:
    - *idProcessParameter (INT)*: Identificador de la tupla.
    - *numQueries (INT)*: Cantidad de consultas generadas.
    - *beta (DOUBLE)*: Factor de decaimiento de la confianza del motor de búsqueda utilizado.
    - *alfa (DOUBLE)*: Factor de confianza en el motor de búsqueda usado.
    - *minLength (INT)*: Largo mínimo usado para generar las consultas.
    - *maxLength (INT)*: Largo máximo usado para generar las consultas.

  Relaciones:
    - *idProcessInstance (INT)*: Identificador de la instancia del proceso en la cual fue utilizado el parámetro. Se modela como una relación uno es a muchos con la tabla *ProcessInstance*.

    - *idSearchEngine (INT)*: Identificador del motor de búsqueda al cual se le enviaron las consultas generadas. Se modela como una relación uno a uno con la tabla *SearchEngine*.
    - *idQueryGenerator (INT)*: Identificador de la función de la estrategia utilizada para generar las consultas, se modela como una relación uno a uno con la tabla *QueryGenerator*.

- *QueryGenerator*: Representa a una estrategia de generación de consultas.
  - Atributos:
    - *idQueryGenerator (INT)*: Identificador de la estrategia.
    - *className (VARCHAR)*: Nombre de la clase que implementa la estrategia.

- *ScoringFunction*: Representa a una función de scoring de resultados.
  - Atributos:
    - *idScoringFunction (INT)*: Identificador de la función de scoring.
    - *className (VARCHAR)*: Nombre de la clase que implementa la función de scoring.

- *SearchEngine*: Representa a un motor de búsqueda.
  - Atributos:
    - *idSearchEngine (INT)*: Identificador del motor de búsqueda.
    - *name (VARCHAR)*: Nombre del motor de búsqueda.

- *MetaAnswer*: Representa una respuesta agregada recuperada en el proceso.
  - Atributos:
    - *idMetaAnswer (INT)*: Identificador de la tupla.
    - *rank (INT)*: Posición en el ranking de todas las meta respuestas recuperadas.
    - *score (DOUBLE)*: Puntaje asignado a la meta respuesta.
    - *url (TEXT)*: URL del documento encontrado.
    - *title (TEXT)*: Título de la página encontrada.
    - *relevanceFeedBack (DOUBLE)*: Puntaje de similitud asignado por el usuario.

  Relaciones:
    - *idProcessInstance (INT)*: Identificador de la instancia del proceso a la cual pertenece la respuesta agregada. Se modela como una relación uno es a muchos con la tabla *ProcessInstance*.

- *QueryAnswer*: Representa una respuesta particular retornada por un motor de búsqueda.
  - Atributos:
    - *idQueryAnswer (INT)*: Identificador de la respuesta.
    - *localRank (INT)*: Ranking local de la respuesta.

  Relaciones:

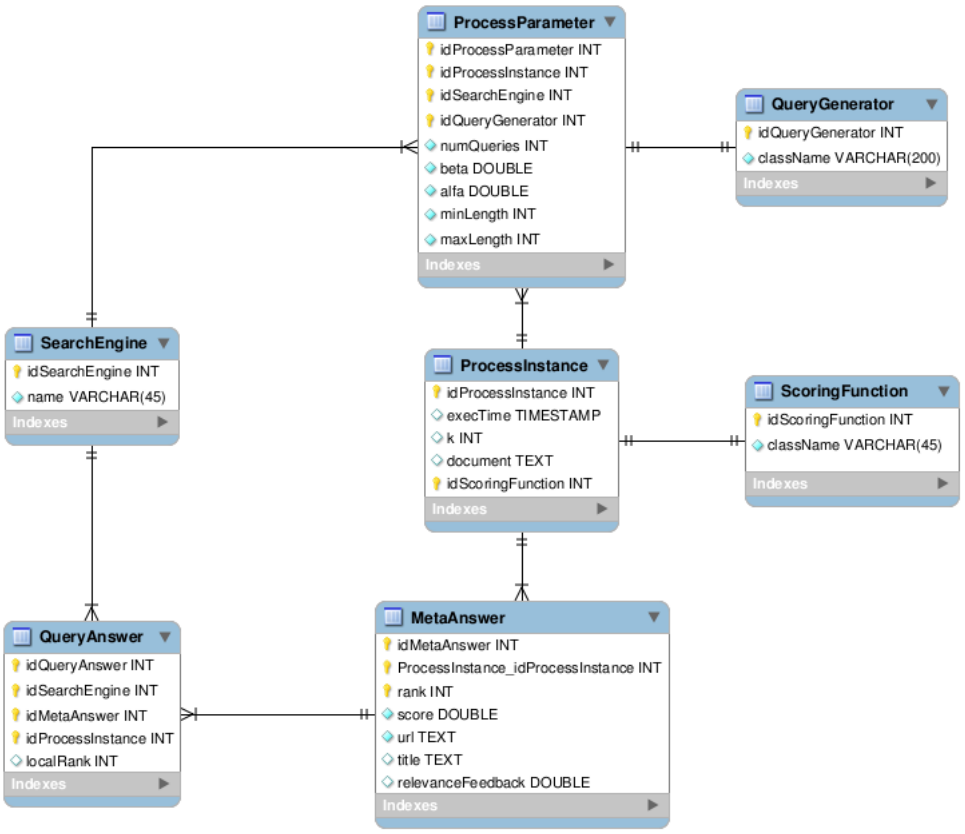

Figura 5.6: Modelo de Datos de registro del Proceso

- *idMetaAnswer (INT)*: Identificador de la repuesta agregada a la que pertenece. Se modela como una relación uno es a muchos con la tabla *MetaAnswer*.
- *idSearchEngine (INT)*: Identificador del motor de búsqueda que obtuvo la respuesta. Se modela como una relación uno es a muchos con la tabla *SearchEngine*.

5.6. Implementación

En esta sección, se detallan las principales características sobre la implementación de la herramienta titulada *Docode Lite*. La herramienta fue implementada en base al modelo lógico y el diseño de Software previamente desarrollados.

A continuación se describe la implementación de cada una de las capas del sistema propuestas en el capítulo 3.

5.6.1. Capa Interfaz

La interfaz se implementó por medio de JavaServer Pages (JSP). Como servidor Web se usó *Apache Tomcat 6.0*. La estructura del sitio se basa en una única página inicial donde los resultados encontrados se despliegan sobre la misma. Para que los resultados aparezcan

en la misma página de la petición se usó *AJAX*[3] con el framework *Prototype*[4]. De igual manera, se usó *AJAX* para permitir la evaluación de los resultados por parte del usuario.

5.6.2. Capa Aplicación

El desarrollo se realizó en el lenguaje de programación Java (*JDK 1.6.0*). El proceso de generación de consultas se implementó tomando como base el algoritmo 4.2.1, usando el framework *Apache Lucene (2.9.0)*. El parsing de los resultados para los motores de búsqueda *Google* y *Bing* fue realizado con la librería de Java *HTMLCleaner*. Mientras que para *Yahoo!* se hizo uso de la API *Yahoo! Boss*[5]. La implementación del Web service se realizó con el motor de Web Services/WSDL/SOAP *Axis2 1.5*.

5.6.3. Capa Datos

El modelo de datos se desarrolló con *MySQL 5.1.37*. Luego todas las transacciones realizadas con la base de datos desde la aplicación se hicieron por medio de *Java Persistence API*, que permite mapear las tablas a objetos, facilitando su manipulación.

[3] Asynchronous JavaScript And XML
[4] `www.prototypejs.org/`
[5] `http://developer.yahoo.com/search/boss/`

Capítulo 6

Evaluación del Modelo Propuesto

En este capítulo, se realiza la evaluación del modelo propuesto para la recuperación de documentos similares. Se diseña un experimento sobre un conjunto de datos construido manualmente. El experimento es evaluado mediante métricas conocidas. Finalmente se discuten los resultados obtenidos.

6.1. Diseño del Experimento

Se recolectó manualmente una muestra de párrafos extraídos de documentos de la Web. El número de párrafos recolectados, fue de 160. Todos los párrafos fueron seleccionados desde diversos sitios Web en español, donde la URL de cada uno de los sitios fue almacenada en la base de datos. Además se clasificó cada uno de los documentos bajo tres categorías.

- *Tipo 1*: Documentos bibliográficos y ensayos escolares.
- *Tipo 2*: Entradas de Blog, o páginas personales.
- *Tipo 3*: Noticias.

Luego los párrafos fueron usados en el sistema como entrada. Los mejores 15 resultados entregados por el sistema para cada párrafo fueron clasificados manualmente como relevantes o no relevantes por medio de la interfaz. El criterio de asignar un resultado como relevante, se definió bajo la condición de que contenga dentro de su texto exactamente el contenido del párrafo entregado[1].

La tabla 6.1 muestra el número de párrafos, de consultas generadas y documentos recuperados por tipo.

Los motores de búsqueda utilizados fueron Google, Yahoo! y Bing. Los parámetros utilizados para cada buscador tanto en el proceso de generación de consultas como de rankeo de resultados se presentan en la tabla 6.1, cuyos valores fueron asignados por inspección.

6.2. Criterio de Evaluación

El objetivo de este experimento fue medir la efectividad del modelo resolviendo el problema RDS. El criterio usado para medir la calidad del modelo fue el número de *documentos encontrados que contentan exactamente el párrafo* (DEPs).

[1]El corpus puede ser descargado en `http://dcc.uchile.cl/~fbravo/docode/corpus.xml`

-	Tipo 1	Tipo 2	Tipo 3	Todos
Párrafos	77	53	30	160
Consultas generadas	539	371	210	1120
Documentos recuperados	1155	795	450	2400

Cuadro 6.1: Número de párrafos, consultas y documentos recuperados por tipo usados en el proceso de evaluación.

-	Google	Yahoo!	Bing
Consultas por párrafo	3	2	2
minLenght por consulta	12	12	12
maxLength por consulta	15	15	15
c_s	0.95	0.93	0.93
β_s	0.5	0.5	0.5

Cuadro 6.2: Valores de los parámetros usados en el experimento.

Las métricas de evaluación seleccionadas fueron *precision at k* y el puntaje promedio asignado para los resultados relevantes y no relevantes agrupados por su ranking de aparición. Donde, la métrica *precision at k* se define como:

$$\text{precision at } (k) = \frac{\text{DEPs recuperados en los top } k \text{ resultados}}{\text{Documentos recuperados en los top } k \text{ resultados}} \tag{6.1}$$

Los puntajes promedio de los resultados relevantes y no relevantes buscan verificar la existencia de una relación entre los puntajes asignados por el modelo con la relevancia real.

6.3. Resultados y Discusiones

Una vez ejecutado el modelo sobre la colección de párrafos seleccionados y evaluado manualmente los resultados encontrados, se calcularon las métricas previamente descritas por medio de consultas *SQL* a la base de datos. En la tabla 6.3 se pueden ver el puntaje promedio asignado por el modelo a todos los DEPs y no DEPS. Se puede apreciar una fuerte dependencia entre el puntaje promedio de los resultados rankeados más arriba y la relevancia. La dependencia disminuye al aumentar k. Los primeros resultados que tienen altos puntajes tienen una alta probabilidad de ser relevantes. Esto se debe a que los resultados que recibieron un alto puntaje, fueron encontrados en varias instancias de búsqueda dentro los primeros lugares. Para resultados de ranking más bajo, el puntaje asignado por el modelo no nos es útil para realizar distinciones entre los resultados relevantes y no relevantes.

En la tabla 6.3 se muestra la *precision at k* para los resultados recuperados asociados al tipo de documento del párrafo de entrada. Se puede ver claramente que la *precision at k* varía con el tipo de documento.

Para el caso de los tipo 1, se ve que los primeros resultados tienen una mejor precisión que para los otros tipos. Esto se debe a que los documentos bibliográficos se encuentran generalmente en sitios muy populares como Wikipedia, los cuales son generalmente indexados por todos los motores de búsqueda y sus resultados tienden a aparecer entre los primeros resultados con mucha frecuencia.

k	DEPs	Puntaje promedio DEPs.	No DEPs	Puntaje promedio no DEPs
1	139	0,534	21	0,221
2	88	0,271	72	0,166
3	64	0,133	96	0,145
4	48	0,171	112	0,135
5	36	0,133	124	0,124
6	31	0,103	129	0,115
7	25	0,136	135	0,107
8	27	0,096	133	0,097
9	19	0,096	141	0,094
10	18	0,096	142	0,090
11	20	0,094	140	0,085
12	16	0,094	144	0,083
13	15	0,077	145	0,078
14	10	0,078	150	0,076
15	14	0,094	146	0,073

Cuadro 6.3: Puntaje promedio para DEPs y no DEPs por ranking

k	Prec. Todos	Prec. Tipo 1	Prec. Tipo 2	Prec. Tipo 3
1	0,869	0,922	0,849	0,767
2	0,709	0,727	0,689	0,700
3	0,606	0,602	0,591	0,644
4	0,530	0,526	0,509	0,575
5	0,469	0,465	0,457	0,500
6	0,423	0,416	0,415	0,456
7	0,385	0,375	0,375	0,429
8	0,358	0,344	0,356	0,396
9	0,331	0,317	0,335	0,359
10	0,309	0,295	0,313	0,340
11	0,293	0,277	0,298	0,321
12	0,277	0,259	0,283	0,311
13	0,263	0,247	0,269	0,292
14	0,248	0,233	0,256	0,274
15	0,238	0,224	0,244	0,260

Cuadro 6.4: Precision at k por tipo de documento de entrada

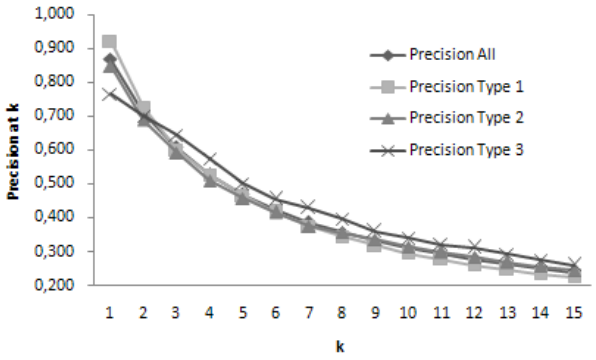

Figura 6.1: Precision at k para distintos tipos de documentos de entrada.

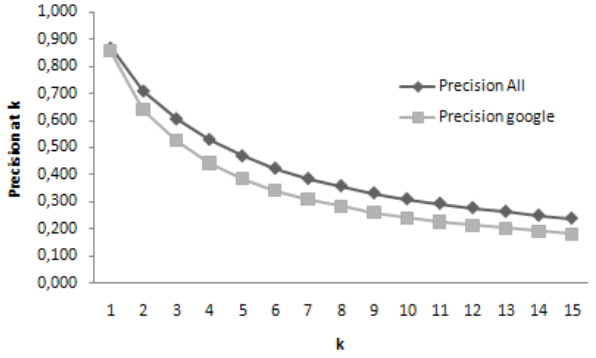

Figura 6.2: Precisión de los resultados entregados por un único motor de búsqueda

Luego, los documentos tipo 2 no son tan populares en la Web como los tipo 1. En este caso, entradas de Blog o páginas personales son difícilmente indexados por todos los motores de búsqueda.

Finalmente, podemos observar precisiones más bajas para los primeros resultados de los documentos tipo 3. Sin embargo, la precisión decrece más lento cuando usamos noticias como entrada. Esto se debe a que las noticias se repiten en muchas páginas Web diferentes, puesto que existen muchos sitios agregadores de contenido que hacen que una misma publicación de noticia sea encontrada en varias páginas diferentes. Lo anterior, posibilita a que se encuentren resultados relevantes para documentos que obtuvieron menos puntaje y por ende quedaron rankeados más abajo.

En la figura 6.3 se muestra claramente que la unión de los motores de búsqueda brinda una mayor cobertura de la Web, mejorando la precisión del modelo.

Capítulo 7

Evaluación del Software

En este capítulo se evalúan el diseño y la implementación de la aplicación bajo la perspectiva de indicadores de calidad de diseño orientado a objetos y pruebas de funcionalidad y eficiencia.

7.1. Evaluación con Métricas de Orientación a Objetos

En [6] se proponen las siguientes métricas para evaluar el diseño orientado a objetos de una aplicación:

- **LCOM**: Porcentaje de falta de cohesión. Representa el grado de similaridad entre los métodos de una clase. Se mide con la cantidad de variables de instancia compartidas. Un método es considerado como cohesivo, cuando ejecuta a una única tarea. Por lo general, a mayor valor de este indicador, menor es la calidad de la clase.
- **DIN**: Profundidad de herencia en la jerarquía de herencia. Por lo general, mientras más profunda sea la herencia, mayor será la dificultad para entender la clase, además aumenta el esfuerzo de desarrollo y mantención. Por otro lado, a mayor profundidad de herencia, mayor es la reusabilidad de código.
- **IFANIN**: Cantidad de clases base inmediatas. Considera la implementación de interfaces.
- **CBO**: Acoplamiento entre objetos. Mide la relación existente con otros objetos, contando la cantidad de objetos con los que relaciona sin considerar relaciones de herencia. Mientras mayor sea su valor, menor será la modularidad y reusabilidad de la clase. Generalmente, las clases con altos niveles de acoplamiento entre objetos, requieren un mayor esfuerzo de mantención y son dificultan la extensibilidad.
- **NOC**: Número de hijos. Representa el número de subclases directas. Por lo general, mientras mayor sea el valor de este indicador, mayor es la reusabilidad de la clase y mayor es el impacto de la clase en el desarrollo de la aplicación.
- **RFC**: Respuesta de una clase. Mide la complejidad de comunicación entre los los componentes de la clase. Incluye todos los métodos que son llamados en la clase, considerando los métodos heredados. A mayor valor de esta métrica, mayor será el tiempo destinado a la depuración y prueba del código.
- **NIM**: Conteo de instancias de métodos.

- **NIV**: Conteo de instancias de las variables.
- **WMC**: Peso de los métodos por clase. Suma los métodos de la clase ponderado por sus complejidades. En este caso se considera la complejidad de cada método como un número unitario. Por lo tanto, entrega valores equivalente a *NIM*. Por lo general, mientras mayor sea valor, mayor será el esfuerzo requerido para desarrollar y mantener la clase. Por otro lado, mientras menor sea su valor, mayores serán las facilidades para reutilizar la clase.

A continuación se levantas los indicadores previamente definidos. En esta evaluación sólo se considera el código programado para el servicio de recuperación de documentos similares, tanto el código de la interfaz y de las transacciones con la base de datos fueron excluidos

7.1.1. Resultados de la evaluación

Para construir automáticamente las métricas señaladas, se utilizó la herramienta *Understand 2.5*[1]. La herramienta entregó los siguientes indicadores generales.

- Clases: 25
- Líneas: 1609
- Líneas en blanco: 389
- Lineas de código: 992
- Lineas comentadas: 303
- Declaraciones: 433
- Ratio Comentario/Código: 0.31

Los valores obtenidos para las métricas de orientación a objetos, se aprecian en la tabla 7.1.

7.1.2. Discusión de valores obtenidos

En esta sección se discuten los valores obtenidos para las principales métricas de orientación a objetos previamente descritas.

Respecto a la falta de cohesión **LCOM**, se tienen valores superiores al 50 % en todas las clases abstractas, principalmente por no tener métodos implementados. Lo mismo ocurre con clases que representan entidades de información como *MetaAnswer*, *QueryAnswer*, *ServiceParameter*, *SearchParameter* al no tener más métodos que los *getters* y *setters* de las variables de instancia.

Luego para la profundidad de herencia **DIN**, ésta no sobrepasa en ninguna oportunidad una profundidad de 2. Las clases que tienen una profundidad de 2, son todas las clases relacionadas por herencia con *Search*, *ScoreFunction*, *QueryGenerator* y *DocProductFactory*. En todos estos casos se usó herencia bajo algún patrón de diseño conocido, por que lo que no se llegó a profundidades de herencia tan grandes, lo que debiese hacerlas simples de comprender.

[1] `http://www.scitools.com/`

Clase	LCOM	DIN	IFANIN	CBO	NOC	RFC	NIM	NIV	WMC
BingSearch	33	2	1	3	0	7	3	1	3
DocProductFactory	71	1	1	0	1	7	7	1	7
GoogleSearch	33	2	1	2	0	7	3	1	3
HyperGeoLMQueryGenerator	0	2	1	1	0	4	2	2	2
Main	0	1	1	3	0	1	0	0	1
MetaAnswer	68	1	2	1	0	12	12	6	12
MetaSearch	57	1	1	6	0	7	7	5	7
QueryAnswer	62	1	1	1	0	9	9	5	9
QueryGenerator	50	1	1	1	2	2	2	2	2
QueryGeneratorContext	41	1	1	2	0	3	3	4	3
RetrieveService	33	1	1	7	0	3	3	6	3
ScoreContext	50	1	1	2	0	4	4	6	4
ScoreFunction	75	1	1	1	4	8	8	3	8
Search	62	1	1	1	3	4	4	2	4
SearchContext	56	2	1	2	0	4	4	4	4
SearchParameter	56	1	1	0	0	8	8	4	8
SentenceQueryGenerator	0	2	1	1	0	4	2	0	2
ServiceParameter	52	1	1	0	0	8	8	5	8
SnippetSimilarityScore	16	2	1	4	0	10	2	3	2
SpanishDocProductFactory	61	2	1	0	0	14	7	6	7
VectorOperations	0	1	1	0	0	2	0	0	2
WeightedBorda	0	2	1	3	0	9	1	0	1
YahooSearch	33	2	1	3	0	7	3	1	3
ZipfLikeScore	0	2	1	3	0	9	1	0	1
ZipfLikeSnippetScore	0	2	1	4	0	9	1	1	1

Cuadro 7.1: Métricas de orientación a objetos de clases

Para el caso de la métrica **NOC** que representa el número de hijos de las clases, se tiene que *ScoreFunction* tiene un valor de 4. En estos casos, donde una clase posee varios hijos, se puede haber caído en el error de diseño de estar modelando como hermanos clases que debiesen tener relaciones de herencia. Pero en este caso particular, se esta usando un patrón *strategy* donde cada hijo representa una estrategia diferente para asignar puntaje a las respuestas, lo cual justifica que puedan desarrollarse varias estrategias diferentes para resolver un mismo problema.

La métrica de acoplamiento entre objetos **CBO** entregó valores elevados para clases que tienen responsabilidades de control y que deben interactuar con varios objetos, como *RetrieveService* y *MetaSearch*. Estas clases se encuentran por lo tanto, fuertemente acopladas a las clases que controlan. Lo cual las hace poco modulares y reusables. Esto es esperable, para clases de control que deben crear y manejar distintos objetos.

Los valores obtenidos para la métrica de respuesta de una clase **RFC**, entregó valores elevados para la clase *SpanishDocProductFactory*, *MetaAnswer* y en general para las clases que fueron pasadas como párametros en varios métodos de otras. Para el caso de *SpanishDocProductFactory* esto se debe a que la fábrica de productos del texto de entrada es usada tanto por funciones generadoras de consultas como por estrategias de scoring de resultados, por lo tanto va pasando como parámetro por varios objetos. Luego para *MetaAnswer* también es un objeto que debe ser trabajado en distintas fases del proceso global. Como estas clases tienen un alto valor de respuesta, destinará alto tiempo de depuración y prueba.

Finalmente la métrica de los pesos de métodos por clase **WMC** en general no se perciben valores elevados, salvo clases que tienen varias variables de instancia como *MetaAnswer*, *QueryAnswer*, *ServiceParameter* y *SearchParameter* que al tener métodos *get* y *set* implementados incrementan el valor de la métrica de manera considerable.

7.2. Pruebas

7.2.1. Prueba de Funcionalidad

Para probar el correcto funcionamiento del sistema se elaboraron funciones en *JUnit* para la clase *RetrieveService* que contiene el servicio principal que se comunica con el cliente. Principalmente se probaron combinaciones de parámetros de generación de consultas, de scoring y de motores de búsqueda usados. Como párrafo de entrada se entregó un párrafo extraído del sitio *www.dcc.uchile.cl*, el cual se verificó que se encuentre indexado por los motores de búsqueda utilizados. Se verificó entonces, que el sistema retorne en esos casos por lo menos un documento. Se probaron varios combinaciones de parámetros del modelo, obteniendo en todos los casos resultados positivos.

A continuación se muestra un ejemplo de las pruebas realizadas:

```
public void testExecuteService() {
System.out.println("executeService");

Collection<ServiceParameter> serPara=new Vector<ServiceParameter>();

serPara.add(new ServiceParameter("Yahoo",0.95,0.5,"SentenceQueryGenerator",2));
serPara.add(new ServiceParameter("Google",0.95,0.5,"HyperGeoLMQueryGenerator",2));
String text="El Departamento de Ciencias de la Computación (DCC), en
concordancia con la Universidad de Chile y en particular con la Facultad
de Ciencias Físicas y Matemáticas, tiene por misión ser un centro de excelencia
en docencia, investigación y extensión para Chile y el extranjero en diversas
áreas de las Ciencias de la Computación. Dicha Misión en particular contempla";
RetrieveService instance = new RetrieveService(serPara,"ZipfLikeScore",text,10);
instance.executeService();
assert !instance.getMetaAnswers().isEmpty();
}
```

Queries	Avg.Time[ms]
2	4226
4	4717
6	5026
8	5982
10	5566
12	6116
14	6335
16	6729

Cuadro 7.2: Tiempos de ejecución

7.2.2. Prueba de Tiempo

Para medir el tiempo de respuesta del sistema, se midió el tiempo promedio de ejecución para distintos párrafos de entrada al ir aumentando la cantidad de consultas generadas.

El experimento se ejecutó en un laptop de procesador `Intel Centrino Duo 1.6 Ghz` 2.0 GiB de RAM en un sistema operativo `Ubuntu Linux 9.10 Kernel 2.6.31-21`.

Se ejecutó 5 veces el servicio con diferentes párrafos de entrada para cada número de consultas generadas. Se usó el MLH para la generación de consultas y Zip-Like para el scoring de resultados. Las consultas fueron distribuidas uniformemente entre el motor de búsqueda Google y Yahoo! usando distintos párrafos para cada instancia de ejecución del servicio. Esto se hizo, porque los motores de búsqueda poseen estrategias de caching de resultados para las consultas repetidas. Entonces, al usar un mismo párrafo en distintos experimentos, se generarían consultas repetidas que mejorarían los tiempos de respuesta de cada motor de búsqueda, distorsionando los resultados del experimento.

Los resultados obtenidos se muestran en el cuadro 7.2.2, donde se ve que para dos consultas se obtiene un tiempo promedio de $4,2$ segundos, y para 16 consultas de $6,7$ segundos. De estos resultados, se puede identificar un leve incremento del tiempo de ejecución al aumentar la cantidad de consultas generadas. Esto se debe a dos razones. La primera se basa en el caching de resultados por parte del motor de búsqueda externo, donde al realizar muchas consultas sobre un mismo modelo de lenguaje aumenta la probabilidad de que éstas sean muy similares entre sí, de esta manera el motor de búsqueda externo recupera los resultados desde su cache de resultados en vez de buscar dentro de su índice invertido. La segunda, es que las peticiones de consultas son ejecutadas en paralelo, lo que permite tener a los motores de búsqueda trabajando más de una consulta en simultáneo, además el algoritmo de scoring Zipf-like realiza un recorrido lineal sobre los resultados recuperados, donde las únicas operaciones que se realizan son inserción y búsqueda dentro del diccionario. En este caso, la implementación de diccionario usada fue la clase *HashMap* de *Java*. Dode las operaciones de búsqueda e inserción tienen una complejidad de $O(1)$ para el caso esperado y $O(n)$ en el peor caso, con n el número total de resultados recuperados[2]. Luego el ordenamiento de objetos *MetaAnwer* mediante *mergeSort* toma $O(nlog(n))$ [5] sobre el conjunto de objetos *MetaAnswer*. De lo anterior, se comprueba la eficiencia de la estrategia planteada.

[2] `http://stackoverflow.com/questions/222658/multiset-map-and-hash-map-complexity`

Capítulo 8

Conclusiones

De la revisión bibliográfica realizada, se desprende que no existían metodologías previas para solucionar el problema de recuperación de documentos similares basadas en la generación probabilística de consultas y metabuscadores con funciones de score que usen solamente los ranking locales de cada buscador y la confianza en éstos. Los resultados experimentales obtenidos en la evaluación del modelo (cap. 6) muestran que el modelo propuesto es capaz de resolver el problema, afirmándose de este modo la hipótesis de investigación planteada en el capítulo 1.

Respecto al modelo propuesto, el modelo de lenguaje hipergeométrico permite la extracción de términos relevantes a partir de un párrafo, donde su flexibilidad de parametrizar el enfoque de asignación de pesos lo convierten en una metodología ligera para la extracción de *key terms* a partir de un documento. De esta forma, esta metodología intenta generalizar los enfoques aleatorizados de *fingerprinting*. Por el lado de la función de scoring propuesta, se demuestra que la función Zipf-like puede ser utilizada como estimador de relevancia de un resultado de consulta retornado por un buscador. Además sus parámetros permiten modelar la confianza en los motores de búsqueda en función de la relevancia promedio de su mejor respuesta y el decaimiento de calidad de sus resultados. Lo anterior, entrega una mayor flexibilidad en el trato que se le da a los motores de búsqueda usados con respecto a técnicas como *Weighted Borda* que sólo consideran un parámetro de confianza.

El modelo propuesto, puede ser aplicado como detector de plagio, analizador de impacto de documentos y recuperador de ideas similares. Por ejemplo, en la detección de plagio el modelo recuperará documentos similares al documento sospechoso, permitiendo al usuario determinar la autenticidad del trabajo. Además, un trabajo plagiado usa generalmente las mismas palabras que el documento original, cambiando tal vez el orden de algunas palabras, el modelo propuesto lo convertiría en un conjunto de consultas que permitirían recuperar el documento original.

Por otro lado, en el análisis de impacto, el número de documentos recuperados podría ser aumentado para analizar la frecuencia de aparición de alguna cita o párrafo, de manera de saber cuantos documentos en la Web lo están citando.

La arquitectura de software propuesta donde un cliente consume un servicio de metabúsquedas entregando un párrafo de entrada, fue probada con éxito. Lo cual muestra que la arquitectura de *DOCODE-lite* orientada a servicios soporta de buena manera el modelo propuesto para resolver el problema de recuperación de documentos similares. Además, el diseño orientado a objetos basado en patrones de diseño conocidos, junto a su positiva evaluación mediante métricas realizada en el capítulo 7 permitirán la extensibilidad del modelo a futuras investigaciones sobre la recuperación de documentos similares abiertas a nuevas estrategias de generación de consultas y ranking de resultados.

Es importante mencionar que en la interfaz de usuario, el cuadro de texto de entrada fue implementado mediante un cuadro de texto sin restricciones de largo. Por lo tanto el usuario, podría ingresar documentos completos como entrada. Sin embargo, en caso de ingresar muchos párrafos, es muy probable que se aumente el número de resultados no relevantes, puesto que el proceso de generación de consultas combinaría términos de párrafos distintos que podrían haber sido plagiados de diferentes fuentes. Por eso el modelo se adapta mejor a los párrafos. Queda abierta una investigación que permita definir el tamaño óptimo de un párrafo para una generación adecuada de consultas.

Como trabajo futuro, se espera que el sistema permita al usuario subir un documento completo en múltiples formatos (*pdf, doc, docx*). Para realizarlo será necesario utilizar estrategias de minería de texto capaces de identificar los párrafos con mayor información y de esta manera construir las consultas. Es probable que se requiera un mayor número de consultas, lo cual dificultaría obtener respuestas en línea para el usuario. Una solución podría ser entregar el reporte vía correo electrónico al usuario una vez computada la solución.

La arquitectura propuesta podría ser extendida a una arquitectura distribuida con procesamiento concurrente, de manera tal de no caer en cuellos de botella en las ejecuciones del proceso. Otro punto a considerar sería permitir que la herramienta se personalice a los usuarios mediante el feedback que ellos entregan. De manera que los parámetros del modelo se adapten al usuario, entregándoles mejores resultados. Algoritmos de clasificación podrían ser utilizados para determinar la relevancia de un resultado, de manera tal de mostrarle al usuario sólo los resultados potencialmente relevantes para éste. Donde, una categorización previa automática del texto de entrada podría ayudar a definir parámetros adecuados para cada tipo de documento.

Finalmente, es importante considerar, que DOCODE-lite es una primera aproximación de solución al problema de recuperación de documentos similares en el contexto del plagio. La integración entre DOCODE-lite con el proyecto DOCODE es fundamental para poder extraer los principales documentos sospechosos de la Web. Luego para poder identificar la autenticidad del documento de entrada otras estrategias de análisis de lenguaje natural y de comparación de texto deberán ser desarrolladas.

Bibliografía

[1] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[2] Giambattista Amati. Information theoretic approach to information extraction. In Henrik Larsen, Gabriella Pasi, Daniel Ortiz-Arroyo, Troels Andreasen, and Henning Christiansen, editors, *Flexible Query Answering Systems*, volume 4027 of *Lecture Notes in Computer Science*, pages 519–529. Springer Berlin / Heidelberg, 2006.

[3] Javed A. Aslam and Mark Montague. Models for metasearch. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 276–284, New York, NY, USA, 2001. ACM.

[4] Ricardo Baeza-yates and Berthier Ribeiro-Neto. Modern information retrieval, 1999.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[6] Anton Eliens. *Principles of Object-Oriented Software Development with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[7] Caio Kinzel Filho. Factory method + reflection: Achieving better extensibility in applications. Technical report, ThoughtWorks, August 2008.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] Claudio Gutiérrez, Gonzalo Navarro, Ricardo Baeza-Yates, Carlos Hurtado, Marcelo Arenas, Mauricio Marín, José M. Piquer, M.Andrea Rodríguez, Javier Ruiz del Solar, and Javier Velasco. *Cómo funciona la Web*. Autoeditada, June 2008.

[10] W.L. Hakerness. Properties of the extended hypergeometric distribution. *Ann. Math. Statist.*, 36(3):938–945, 1965.

[11] Daniel C. Halbert and Patrick D. O'Brien. Using types and inheritance in object-oriented languages. In *European conference on object-oriented programming on ECOOP '87*, pages 20–31, London, UK, 1987. Springer-Verlag.

[12] Álvaro R. Pereira Jr. and Nivio Ziviani. Retrieving similar documents from the web. *J. Web Eng.*, 2(4):247–261, 2004.

[13] Giridhar Kumaran and Vitor R. Carvalho. Reducing long queries using query quality predictors. In *SIGIR '09: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 564–571, New York, NY, USA, 2009. ACM.

[14] James Lewis, Stephan Ossowski, Justin Hicks, Mounir Errami, and Harold R. Garner. Text similarity: an alternative way to search medline. *Bioinformatics*, 22(18):2298–2304, 2006.

[15] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[16] S. V. Nagaraj. *Web Caching And Its Applications (Kluwer International Series in Engineering and Computer Science)*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[17] Yves Rasolofo, Faïza Abbaci, and Jacques Savoy. Approaches to collection selection and results merging for distributed information retrieval. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 191–198, New York, NY, USA, 2001. ACM.

[18] Ricardo Seguel. Seguridad en web services. *NeoSecure*, June 2005.

[19] Erik Selberg and Oren Etzioni. The metacrawler architecture for resource aggregation on the web. *IEEE Expert*, 12:8–14, 1997.

[20] Gabriel L. Somlo and Adele E. Howe. Using web helper agent profiles in query generation. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 812–818, New York, NY, USA, 2003. ACM.

[21] J. D. Velasquez and V. Palade. *Adaptive Web Sites: A Knowledge Extraction from Web Data Approach*. 2008.

[22] Zonghuan Wu, Weiyi Meng, Clement Yu, and Zhuogang Li. Towards a highly-scalable and effective metasearch engine. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 386–395, New York, NY, USA, 2001. ACM.

[23] Zonghuan Wu, Vijay Raghavan, Hua Qian, Vuyyuru Rama, Weiyi Meng, Hai He, and Clement Yu. Towards automatic incorporation of search engines into a large-scale metasearch engine. In *WI '03: Proceedings of the 2003 IEEE/WIC International Conference on Web Intelligence*, page 658, Washington, DC, USA, 2003. IEEE Computer Society.

[24] Bilal Zaka. Empowering plagiarism detection with a web services enabled collaborative network. *Journal of Information Science and Engineering*, 25(5):1391–1403, 2009.

[25] ChengXiang Zhai. Statistical language models for information retrieval a critical review. *Found. Trends Inf. Retr.*, 2(3):137–213, 2008.

[26] George K. Zipf. *The Psychobiology of Language*. Houghton-Mifflin, New York, NY, USA, 1935.

[27] George K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.

Anexos

De este trabajo, se realizaron las siguientes publicaciones en conferencias internacionales:

- F. Bravo-Marquez, G. L'Huillier, S. Rios, J.D. Velasquez, and L. Guerrero *DOCODE-lite: A Meta-Search Engine for Document Similarity Retrieval*, In *KES '10: 14th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems*. Cardiff, Wales, 2010. Springer-Verlag.
- F. Bravo-Marquez, G. L'Huillier, S. Rios, and J.D. Velasquez *Hypergeometric Language Model and Zipf-like Scoring Function for Web Document Similarity Retrieval*, In *SPIRE '10: 17th International Symposium on String Processing and Information Retrieval*. Los Cabos, Mexico, 2010. Springer-Verlag.

Ambos trabajos se muestran en las siguientes páginas.