# REUTILIZACIÓN DE CÓDIGO FUENTE ENTRE LENGUAJES DE PROGRAMACIÓN

Enrique Flores Sáez

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS
Y COMPUTACIÓN

---

Dirigido por:
Paolo Rosso y Lidia Moreno


UNIVERSIDAD
POLITECNICA
DE VALENCIA

Trabajo Final de Máster desarrollado dentro del Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Valencia, Febrero 2012

*La mentira más común es aquella con la que un hombre se engaña a sí mismo. Engañar a los demás es un defecto relativamente vano.*

Friedrich Nietzsche

# Resumen

El uso de material ajeno sin reconocimiento al autor se considera plagio. Cuando se cita la fuente del material usado o simplemente éste proviene de una fuente de distribución libre se considera que se está realizando un proceso de reutilización del mismo. En la actualidad, dentro de la era digital, dónde casi cualquier contenido está disponible en Internet existe una gran tentación por reutilizar. El material reutilizado puede ser una pista musical, una imagen, un texto e incluso un código fuente. Esta facilidad de acceso a los recursos impone la creación de herramientas de detección de reutilización para preservar la propiedad intelectual de los mismo.

En el mundo de la industria de desarrollo de software existe gran interés por mantener la autoría de códigos fuente, y preservar el uso legítimo que se establece a través de licencias o contratos de uso. En el ámbito académico, también es interesante la detección de reutilización con el fin de disuadir al alumnado de la creciente práctica de "copiar y pegar" o presentar el trabajo de un compañero como propio. Reutilizando material escolar, no se cumple con la obligación de demostrar sus conocimientos frente a la figura del profesor.

En este trabajo presentamos una descripción del estado del arte en materia de detección de reutilización tanto a nivel de textos como a nivel código fuente centrándose en este última. Además, proponemos dos modelos de detección de reutilización en códigos fuente y sus prestaciones a nivel monolingüe y translingüe. Los resultados obtenidos hasta ahora con los modelos planteados: a nivel de documento y a nivel de fragmento han generado resultados prometedores. Con el fin de continuar los experimentos realizados, se definen una serie de líneas de investigación futuras.

# Agradecimientos

Este trabajo de fin de máster, no se habría podido realizar sin la ayuda y apoyo directo e indirecto de varias personas a las que me gustaría agradecer en este apartado.

Primeramente, quisiera agradecer a Paolo Rosso y Lidia Moreno por haber confiado en mi persona, por la paciencia y su constante participación en la realización este trabajo. Sin ninguna duda van a ser muy importantes en mi etapa doctoral.

En segundo lugar, también darle las gracias a Alberto Barrón-Cedeño por sus siempre acertados consejos y valoraciones, siempre estar ahí ante cualquier problema y todas sus sugerencias durante la realización de este trabajo. Aunque no la necesitará, le deseo la mayor de las suertes con su tesis doctoral.

Tampoco cabe olvidar a compañeros del grupo de investigación, por sus comentarios y apoyo durante este periodo de máster. A amigos de la universidad, por permitir entrar en sus vidas y motivarme para seguir adelante y no decaer.

A mi família que me acompañó en esta aventura y intentar entender el mundo de la investigación sin formar parte de el. A María que, de forma incondicional, me mostró su apoyo y comprensión, siempre estando ahí cuando la he necesitado.

Finalmente, gracias a todos aquellos a los que no haya podido mencionar en este apartado y que aunque no lo sepan han influido indirectamente en este trabajo.

# Índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

## 1.1. Descripción del problema, motivación y objetivos

La extensiva accesibilidad a Internet ha posibilitado la existencia de enormes cantidades de información al alcance de cualquier usuario. Este hecho ha generado nuevas formas de trabajar partiendo de obras desarrolladas previamente por otros autores, conociéndose estas prácticas como reutilización (en inglés *reuse*). Pero también ha dado pie a un incremento considerable de problemas de autoría y plagio, que aunque en algunos casos están estrechamente relacionados se trata de problemas distintos.

Se entiende el término plagiar (plagiarize) como "la acción de copiar en lo sustancial obras ajenas, dándolas como propias" [11]. En el caso de texto escrito, puede producirse la reutilización de fragmentos de otros textos sin que necesariamente se considere plagio, citando las fuentes originales. Es obvio que las obras publicadas en cualquier medio digital están expuestas a copias y reutilización de todo o parte de ellas aunque tampoco esta práctica deba suponer plagio necesariamente ya que puede tratarse de una copia de obras de libre disposición y que por lo tanto dicha copia esté autorizada.

Por otra parte, el problema de autoría consiste en identificar al autor de una obra, sea ésta una imagen, un texto o un programa. En muchos casos no está asociado a temas de plagio sino más bien a resolver delitos de estafa con el fin de conocer la autenticidad

de una obra de arte, o de lingüística forense analizando notas de suicidio sospechosas o mensajes de texto anónimos, o bien identificar el autor de un programa malicioso. En esta línea de investigación se encuentran trabajos como el de Stein et al. [33], Stamatatos et al. [32], Peng et al. [23] o Georgia Frantzeskou et al. [16].

En cualquier caso, se evidencia un gran interés en la identificación del plagio entre las obras publicadas. En abril de 2011, apareció en prensa una noticia que supuso un escándalo a nivel internacional; el ex ministro alemán de Defensa Karl Theodor zu Guttenberg fue condenado por la vía penal y ha tenido que dimitir de todos sus cargos por plagiar gran parte del texto de su tesis doctoral[1]. Y aún podría agravarse más si los plagiados, en su mayoría profesores universitarios, ejercieran su derecho a denunciar a Guttenberg por atentado a la propiedad intelectual. Este es un claro ejemplo de la relevancia de la detección del plagio y sus consecuencias.

En la literatura correspondiente al plagio entre textos escritos, concretamente entre artículos científicos, se plantea otro nuevo término, auto-plagio (self-plagiarism) consistente en la reutilización de una parte significativa, idéntica o casi idéntica de un trabajo propio sin citar el trabajo original [19].

Sin embargo, no es trivial la localización y comparación de toda la información relevante para asegurar un plagio, o al menos cierto grado de similitud entre obras. Recientemente se encuentra disponible en la red una nueva herramienta, *churnalistm*[2], que viene del término inglés *churn* (remover) y *journalism* (periodismo). Esta herramienta detecta el plagio entre artículos periodísticos en inglés publicados en la BBC, en SKY News o en periódicos del Reino Unido. Muestra el artículo copiado y el original, la cantidad de caracteres que coinciden en ambos, porcentaje de texto cortado y porcentaje que ha sido pegado, además de mostrar el autor original y dónde y cuándo el artículo se publicó. Más detalles sobre reutilización de texto escrito se verán en la sección 2.1 de esta memoria.

[1] El 23 de febrero de 2011 Guttenberg ha sido desposeído de su título de doctorado por la Universidad de Bayreuth tras las pruebas de plagio detectadas en su tesis doctoral: `http://www.elpais.com/articulo/internacional/Dimite/ministro/Defensa/aleman/plagiar/tesis/doctoral/elpepuint/20110301elpepuint_6/Tes`

[2] `http://www.churnalism.com`

Un caso particular se produce cuando estas obras publicadas consisten en códigos fuente. En la actualidad, un amplio colectivo de programadores pone a disposición pública códigos fuente bajo licencias que protegen la propiedad de sus creaciones[3]. Esta situación propicia la reutilización de una parte o de la totalidad de cualquier código fuente disponible en la red.

El plagio de código fuente, presentación de un código de otro autor como propio, es un problema serio tanto para el negocio de desarrollo de software como para el mundo académico.

En el ámbito académico, la copia de las tareas de programación asignadas a los alumnos (ya sea desde la Web o desde otro código desarrollado por un compañero) se observa con mucha frecuencia. Esto pone en duda la honradez de algunos alumnos, afectando seriamente a la credibilidad en lo que concierne a la evaluación de sus trabajos. Este problema ya se empezó a tratar a finales de los años 70. Por un lado está el trabajo de Halstead [20] que aunque no estaba interesado en el problema de plagio, éste definió unas métricas orientadas a medir la complejidad o esfuerzo en la creación de código. Otro artículo pionero es el de Faidhi et al. [13] cuya aportación más significativa consiste en el estudio y caracterización en seis niveles de las principales modificaciones que tienden a realizar los estudiantes con el fin de ofuscar[4]. A partir de los dos trabajos referenciados, numerosos investigadores han desarrollado nuevas métricas y algoritmos con la idea de que probablemente se haya producido plagio si dos programas producen valores parecidos para estas métricas con un cierto umbral predeterminado. JPLAG[5] es la herramienta más difundida para la detección de copia de las tareas de programación monolingüe, es decir, en un mismo lenguaje de programación, asignadas a los alumnos, permitiendo sólo su uso a profesores.

En lo que respecta a la reutilización del software comercial, no necesariamente tiene que considerarse plagio si se trata de software

[3] A menudo se permite la reutilización de código fuente gracias a licencias como la de Creative Commons http://creativecommons.org/

[4] La ofuscación en plagio se considera como la inserción de ruido con la finalidad de esconder el material plagiado.

[5] `https://www.ipd.uni-karlsruhe.de/jplag/`

de código abierto o con autorización de su desarrollador. En numerosas ocasiones, empresas y organismos se ven envueltos en litigios y acuerdos millonarios debido a la reutilización de código fuente no autorizado como en el acuerdo firmado entre Microsoft y Novell, donde esta última le pagaba 40 millones de dólares a Microsoft a cambio de no demandar a los clientes de Novell que utilizan los sistemas operativos Linux que parten de tecnología patentada propiedad de Microsoft[6]. La posible demanda se basa en que Microsoft asegura que son propietarios de las patentes en la que se basa parte de la tecnología que utilizan los sistemas operativos Linux. También Microsoft, que ya demandó a Motorola por sus teléfonos con el sistema Android, ha presentado recientemente una denuncia por violación de patente contra la librería Barnes & Noble por su dispositivo electrónico de lectura NooK, ampliando el ataque legal contra aparatos que cuentan con el sistema operativo Android de Google[7].

Generalmente, este tipo de litigios suelen producirse en los Estados Unidos de Norte América ya que vienen provocados por violación de patentes de software. En la Unión Europea (UE) no existe esa guerra de patentes. Aunque ha habido intentos por patentar el "software", en la legislación actual de los estados no se consideran patentables los programas, es decir, la implementación de un algoritmo. Por otra parte, en la UE existe una directiva, de obligado cumplimiento, sobre patentes de "invenciones implementadas por ordenador" que permite patentar invenciones que se realizan mediante un programa informático considerando que pertenecen al campo tecnológico. No obstante, los programas están protegidos por los derechos de autor, propiedad intelectual. La diferencia más significativa entre propiedad intelectual y propiedad industrial consiste en que en caso de litigio por plagio, según la legislación sobre propiedad intelectual ha de ser el plagiado el que tenga que demostrar que es el auténtico autor para poder interponer una demanda, mientras que según la legislación de propiedad industrial sería suficiente con mostrar la patente.

[6] http://www.zdnet.com/blog/btl/novell-and-microsoft-enter-into-late-night-spitting-match-over-40m-ip-payoff/3988

[7] http://www.microsoft.com/presspass/press/2011/mar11/03-21corpnewspr.mspx

Concluyendo, éstas son algunas de las razones del creciente interés en el desarrollo de mecanismos dirigidos hacia la detección de códigos similares y la posible reutilización de los mismos; incluso, detectando casos en los que el plagio ha consistido en la traducción de código fuente de un lenguaje de programación a otro distinto. Sobre estos temas de reutilización de código fuente entre códigos correspondientes a un mismo lenguaje de programación o distinto, es decir, monolingüe o translingüe, se verán más detalles respectivamente en los apartados 2.2 y 2.3.

La ingeniería lingüística, más conocida en el área de la Inteligencia Artificial como Procesamiento del Lenguaje Natural (PLN), facilita el tratamiento automático de la documentación textual. Se han aplicado recursos y técnicas de PLN con el fin de detectar similitud, incluso para texto traducido entre distintas lenguas [25] consiguiendo resultados de cobertura bastante altos entre inglés y otros 5 idiomas.

Frantzeskou et al. afirma en [16] que "los lenguajes de programación se parecen a los lenguajes naturales en tanto que ambos, códigos fuente y textos escritos en lenguaje natural, se pueden representar como cadenas de símbolos (caracteres, palabras, frases, etc.)". Esta apreciación y la escasez de estudios dirigidos a este tipo de reutilización de código fuente traducido entre diferentes lenguajes de programación han motivado la realización de este trabajo. El autor pretende demostrar a través de los experimentos realizados que es posible detectar la similitud de código fuente monolingüe o translingüe aplicando técnicas de PLN.

En este trabajo se proponen dos modelos basados en $n$-gramas de caracteres para detectar la similitud y posible reutilización de código fuente, incluso tratándose entre códigos escritos en distintos lenguajes de programación. Ambos modelos abordan el problema de reutilización, uno trabaja a nivel de documento, y el otro trabaja comparando fragmentos de código con el fin de detectar solo partes del código. Este último representa mejor las situaciones reales de reutilización. Estos modelos se exponen detalladamente en el Capítulo 1.2 de esta memoria.

## 1.2. Estructura de la tesis

Adicionalmente a este capítulo introductorio, el presente trabajo consta de cuatro capítulos más, los cuales se describen a continuación:

- **Capítulo 2** Estado del arte.

  En este capítulo se describen los tipos de reutilización existentes en textos escritos y los dos principales tipos de enfoques para su detección: analisis intrínseco y extrínseco. Por otra parte, se describen las dos principales tendencias en la detección de reutilización en código fuente a nivel monolingüe: basadas en características y de tipo estructural. Finalmente, se describen los dos trabajos que presentan un modelo de detección translingüe.

- **Capítulo 3** Modelos propuestos.

  En este capítulo se describen los dos modelos propuestos en este trabajo de investigación: a nivel de documento y a nivel de framento. En ambos modelos los términos se representan como $n$-gramas de caracteres sin que el orden de éstos sea relevante. Para comparar los códigos fuente, se usa como medida de similitud la función del coseno.

- **Capítulo 4** Evaluación de los modelos.

  En esta parte se describen los dos corpus utilizados para ajustar los modelos y evaluarlos: SPADE y CL-AT++. Además, se describe una nueva métrica aplicable cuando las medidas estándar no aportan suficiente información. Ambos modelos se ajustan utilizando el corpus SPADE y se validan con el corpus CL-AT++ comparándose con la herramienta JPlag.

- **Capítulo 5** Conclusiones y trabajos futuros.

  En este capítulo se retoman las conclusiones extraídas a lo largo de los experimentos. Además se proponen las líneas a seguir con este trabajo de investigación como son: la creación de nuevos corpus, detección de fragmentos sospechos, aplicar otras

técnicas de lenguaje natural, combinar con técnicas utilizadas en otros estudios previos. También se plantea como objetivo final del trabajo de investigación desarrollar una plataforma abierta que permita detectar reutilización en grandes colecciones de códigos fuente, idependientemente de si está escrito en un mismo lenguaje de programación o en diferentes. Dada su complejidad, estas tareas serán abordadas con mayor profundidad dentro de la investigación doctoral.

# Capítulo 2

# Estado del Arte

## 2.1. Reutilización y detección automática de plagio en textos

Existe cierta ambigüedad a la hora de hablar de reutilización y plagio entre obras, ya sea a nivel de lenguaje natural como a nivel de lenguaje de programación que es necesario precisar.

Al hablar de reutilización de texto, se entiende como el proceso de uso de una fuente dada o conocida, mientras que cuando se habla de plagio, se refiere al proceso de reutilización donde no se le da reconocimiento a la fuente. Cabe destacar que el primer término se considera hiperónimo del segundo y dado que a lo largo de esta tesis se va a utilizar casos conocidos y reconocidos de reutilización no se va a emplear el término plagio al hablar de detección de similitud entre códigos fuente.

Un ejemplo conocido de reutilización es el caso de la prensa, donde una agencia recoge una noticia en un lugar y lo vende a distintas redacciones de periódicos. Estos periódicos, tienen dicha información de la misma fuente, la cual citan, y tienen permiso para realizar una copia exacta de dicha fuente, o bien, reescribir la noticia a partir de dicha fuente. Por el contrario, en el ámbito académico, un estudiante que ha reutilizado el trabajo de otro compañero, presenta al profesor un trabajo reutilizado como si fuera propio, provocando una situación de plagio. Una vez distinguidos ambos conceptos, también es necesario comentar que la detección de reutilización está fuerte-

mente ligada a otros campos de PLN como puede ser la atribución de autoría [18, 33], siendo su diferencia en el propósito, uno intenta encontrar partes escritas por otro autor, y el otro intenta identificar a que autor pertenece un texto.

Cuando hablamos de reutilización de texto debemos identificar los distintos tipos existentes: (*i*) La copia exacta es el más fácil de identificar, se considera un problema resuelto con algoritmos de identificación de subcadenas más largas; (*ii*) reutilización consistente en añadir o quitar palabras del texto; (*iii*) paráfrasis o resumen, donde se produce una reformulación de oraciones y, en el caso de resumen, además se produce una síntesis del contenido del texto mucho más difícil de detectar que los casos anteriores; (*iv*) traducción de texto, su detección es compleja debido a que es necesario conocer ambos idiomas y las traducciones muchas veces precisan de cambios por formas que no existen en un lenguaje y en otro si.

A la hora de abordar la detección de reutilización es necesario distinguir diferentes enfoques en los que se va a centrar el proceso de detección, como por ejemplo si va a abordar un problema monolingüe o translingüe. Va a ser más complicado detectar reutilización si ha habido un proceso previo de traducción, para lo cual habrá que añadir al detector mecanismos de traducción o alguna otra técnica específica [4].

También se pueden clasificar los tipos de detección según la naturaleza de la aproximación empleada. Se llama análisis *intrínseco*, al análisis que sin conocer la fuente de la reutilización trata de detectar que partes del texto pertenecen a un autor distinto del original. Por otra parte, el análisis *extrínseco* consiste en identificar las fuentes reutilizadas y los pares de fragmentos reutilizados dado un conjunto de fuentes potencialmente sospechosas de reutilización.

En la detección intrínseca, al no existir un conjunto de posibles fuentes contra las que comparar partes del texto, es necesario comparar con el propio texto. Para ello se trata de determinar las características propias del autor como son el rango del vocabulario, longitud de las oraciones y de palabras. Se determina el estilo del

autor del texto y es comparado con el estilo de cada fragmento de texto [9]. Si existe una variación notoria del estilo en un fragmento este es marcado para una revisión manual para que un experto humano tenga la decisión final. Un ejemplo de herramienta que realiza de forma gráfica el estudio utilizando distintas medidas de estilo es Stylysis[1], con medidas como la edad necesaria para entender un texto, longitud media de las oraciones, longitud media de las palabras y la variedad del vocabulario.

Para la detección extrínseca, sí que se dispone de una colección contra la que comparar el documento fuente. Para ello se propone en Stein et al. [33] un esquema básico. Inicialmente se hace una selección de aquellos documentos que casen con la temática del documento fuente para reducir el subconjunto de posibles sospechosos. En segundo lugar, sobre este conjunto reducido se realiza una comparación más minuciosa que la anterior con el fin de detectar los documentos reutilizados. Finalmente se hace un post-proceso para descartar los casos que no son verdaderamente plagio. La primera etapa de este esquema, generalmente, tiene que trabajar con una gran cantidad de documentos por lo que es necesario tener una comparación menos exhaustiva [5].

Respecto a la etapa de comparación, existen distintas aproximaciones pero en general en todas ellas se distinguen tres etapas: preproceso, representación de la información y aplicación de una medida de similitud.

Para poder comparar textos se han realizado distintos tipos de preproceso. Uno de ellos y el más extendido es el de eliminar los espacios en blanco y los símbolos de puntuación para darle importancia al contenido del texto. También es muy común eliminar la capitalización de las letras, convirtiendo todos los caracteres en minúsculas. Otro tratamiento que proporciona buenos resultados es la aplicación de listas de palabras de detención, eliminando palabras muy comunes, que no son relevantes y no aportan información (en inglés a este tipo de palabras se les denomina *stopwords*). Por último, comentar otro preproceso que consiste en sustituir todas las palabras que sean

[1] `http://memex2.dsic.upv.es:8080/StylisticAnalysis`

derivadas de una por la raíz (en inglés llamado *stemming*). Con esta técnica se detectan mejor las reformulaciones como por ejemplo cambiar el tiempo verbal de una oración.

Para representar la información después de realizar el preproceso se han utilizado técnicas desde bolsas de palabras que contienen las palabras del texto junto a su frecuencia, hasta representaciones en "huellas digitales". El concepto de bolsa de palabra (en inglés, *bag of words*) consiste en generar una lista desordenada de términos que aparecen en un texto, perdiendo tanto el contexto como el orden. Además, en la bolsa de palabras se almacena la frecuencia de aparición de cada término. Estos términos pueden ser palabras o agrupaciones de caracteres ($n$-gramas). Cuando se habla de $n$-gramas se refiere a una secuencia de caracteres contiguos de tamaño $n$. El concepto de huella digital (en inglés *fingerprint*) consiste en generar a partir de fragmentos del texto una representación a modo de resumen de lo que contiene el texto. Un ejemplo consiste en utilizar funciones *hash* para esta tarea, la cual a partir de una secuencia de caracteres, obtiene un número único, y que al realizar la mínima modificación de la secuencia, la función devuelve un número completamente distinto.

Estas informaciones se comparan para estimar su similitud aplicando distintas medidas. En el trabajo de Barrón-Cedeño et al. [2] se describen las medidas agrupadas en tres modelos: ($i$) modelos de espacio vectorial (en inglés *Vector Space Model*, VSM), ($ii$) modelos basados en la huella digital y ($iii$) modelos probabilísticos.

Dentro de los modelos de espacio vectorial encontramos el coeficiente de Jaccard que consiste en dividir el tamaño de la intersección de dos conjuntos de muestras entre el tamaño de su unión como muestra la ecuación 2.1.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.1}$$

En el mejor de los casos cuando el conjunto $A$ es igual al conjunto $B$, la intersección y la unión tienen el mismo valor y la división de ambos tiene valor 1. En el peor de los casos la intersección devuelve

el valor nulo y la división vale 0, por lo que se tiene una medida de similitud acotada en el rango 0-1. Otra de las medidas más utilizadas es la de la similitud del coseno, que consiste en calcular el ángulo que forman dos conjuntos de muestras dentro de un espacio vectorial. La explicación de esta fórmula se encuentra más detallada en la subsección 3.1.2 por haber sido utilizada en los experimentos de este trabajo de investigación.

Entre los modelos de fingerprint se utiliza la técnica de *winnowing* [29] y SPEX [7]. La técnica de winnowing consiste en generar un valor hash al texto cada $k$-gramas obteniendo una secuencia de *hashes*. Estos $k$-gramas pueden ser de caracteres, de palabras, etc. A continuación, mediante una ventana deslizante se guarda el menor valor de hash dentro de la ventana. Todos los hashes seleccionados de esta secuencia, compondrán la huella del documento. Finalmente se comparará los valores de las huellas para detectar similitud. Por otra arte, la técnica SPEX consiste en identificar secuencias comunes más largas entre dos documentos. Se parte de la idea que si una subcadena es única, la cadena será única. Para ello genera una secuencia de hashes de los documentos. Sobre estas secuencias genera bigramas, y selecciona aquellos que estén en más de un documento. Estos valores de hash se almacenan y se itera el mismo proceso hasta 8-gramas. La huella final es el conjunto de los hashes seleccionados.

$$KL(A||B) = \sum_{x \in X} (A(x) - B(x)) \log \frac{A(x)}{B(x)} \tag{2.2}$$

El tercer grupo de medidas se basan en un modelo estadístico. Todos ellos calculan la similitud a partir del vector de características de un documento como de cerca está la distribución de probabilidad sobre otro documento. Un ejemplo de estos métodos es la divergencia de Kullback-Leibler. La divergencia de Kullback-Leibler es un indicador de la similitud entre dos funciones de distribución. Consiste en medir como de cerca están dos distribuciones de probabilidad $A$ y $B$ como se muestra en la ecuación 2.2.

Como se ha comentado anteriormente, existe un tipo de reutilización adicional: la reutilización translingüe. Este tipo de reutilización ha sido tratado recientemente en el estudio Potthast et al. [25] proponiendo 3 sistemas distintos para abordarlo. El primero consiste en la técnica del análisis explícito translingüe (*Cross-Language Explicit Semantic Analysis*, CL-ESA); donde se compara el documento fuente y el sospechoso con un corpus translingüe y paralelo, por lo que cada elemento comparado con el documento fuente, tendrá uno en el otro idioma que se comparará con el documento sospechoso. La comparación del fuente y el sospechoso generará dos vectores de valores de similitud de la misma longitud. Si estos vectores tienen valores parecidos se entenderá que el contenido de uno es similar al del otro. El segundo consiste en el alineamiento entre lenguajes basado en los principios estadísticos de la traducción automática (*Cross-Language Alignment-based Similarity Analysis*, CL-ASA). Se genera a partir de traducciones un diccionario estadístico automático de todas las posibles traducciones de una palabra al otro idioma. Finalmente se comprueba si el documento fuente es una traducción del documento sospechoso. Finalmente el método más sencillo de los tres y que obtiene buenos resultados si las lenguas guardan cierta relación. Este método (*Cross-Language Character N-Grams*, CL-CNG) divide el texto en $n$-gramas y obtiene la similitud entre los dos textos con funciones como la similitud del coseno o el coeficiente de Jaccard.

## 2.2. Detección automática de plagio en código fuente monolingüe

Esta sección va a describir la detección de la reutilización en código fuente, porqué genera interés su detección y cómo se ha abordado el problema hasta el momento. El principal objetivo de la reutilización es el de usar un material externo como propio con o sin reconocimiento del autor.

En este contexto existen escenarios en los que se produce reutilización: la industria del software y el mundo académico. Dentro de la industria del software se persigue preservar la propiedad intelectual. Las empresas invierten grandes sumas de dinero desarrollando software, y lo protegen mediante patentes. Por lo tanto es de gran

valor para las empresas de software que sus productos no sean reutilizados por otras empresas o por la comunidad de programadores. En el caso contrario, cuando un programador independiente publica un código fuente bajo una licencia con propósitos no comerciales[2], interesa tener un mecanismo de detección.

En el mundo académico se evalúa a los alumnos con el fin de que estos muestren los conocimientos adecuados a través de la implementación de programas, y en el caso de reutilizar código fuente, estos conocimientos no se demuestran realmente. Es por ello el interés en disponer de herramientas que ayuden al evaluador a detectar la reutilización efectuada por el alumno.

Uno de los principales problemas para la detección de reutilización de código fuente es que el espacio de búsqueda es muy amplio. Por ejemplo en un repositorio pueden existir muchas versiones de distintos proyectos, y esto puede suceder en todos los repositorios con códigos fuente funcionales disponibles en la red. Otro problema es que las empresas no muestran el código fuente que utilizan y esto dificulta la tarea de detectar si están utilizando código fuente ajeno o no. Esto centra la detección de código fuente en repositorios de código fuente locales o repositorios públicos de la red.

Otro gran problema existente es la ofuscación del propio código fuente. El concepto de ofuscación consiste en transformar un código fuente en otro menos comprensible para un ser humano pero con el mismo comportamiento de entrada/salida y funcionalidades que el original. Con la ofuscación se consigue dificultar la comprensión y la reutilización de un código para otras aplicaciones. Además de dificultar la reutilización también esconde si un código ha sido reutilizado dificultando su detección.

Actualmente, se dispone de herramientas que realizan estas tareas automáticamente como son SandMark o BCEL[3] de la Apache Software Foundation. La ofuscación en un principio fue diseñada

[2] Un ejemplo de este tipo de licencias son las Creative Commons `http://creativecommons.org/` o las licencias Apache de la Apache Software Foundation. `http://www.apache.org/`

[3] *SandMark* ofusca código Java a nivel de bytecode (código intermedio Java) `http://sandmark.cs.arizona.edu/`, *Byte Code Engineering Library* BCEL también trabaja con bytecode de Java `http://commons.apache.org/bcel/`

para evitar que se obtuviera información o diseños de un código fuente, con el fin de dificultar su reutilización. Obviamente si un código fuente ha sido reutilizado, aplicando técnicas de ofuscación será mucho más difícil su detección.

Se producen multitud de casos donde sin disponer del código fuente se descubre su funcionalidad interna utilizando tests unitarios. A esta ciencia se le llama ingeniería inversa. La ingeniería inversa consiste en obtener información o diseños a partir de un producto accesible al público, para determinar de que está hecho, qué lo hace funcionar o cómo fue fabricado. Uno de los casos más conocidos donde se ha aplicado la ingeniería inversa es el sistema de compartición de archivos entre equipos de Microsoft Windows implementado en el programa Samba o la aplicación Wine[4] para ejecutar aplicaciones nativa de Microsoft Windows bajo la plataforma Linux/UNIX.

Whale [36] hizo un estudio sobre las técnicas más empleadas por los estudiantes para dificultar el proceso de detección por parte del profesor. Algunos de los cambios más frecuentes realizados por los estudiantes son: cambios en comentarios, cambios de identificadores, cambios de orden de operandos, cambio de tipos de datos, reemplazar expresiones por equivalentes, añadir sentencias redundantes, cambios de sentencias independientes del orden de ejecución, etc.

Faidhi y Robinson [13] realizaron una clasificación de los tipos de modificaciones según su dificultad. Estos niveles van desde la copia exacta hasta cambios de estructura general del código fuente. En la sección 3.1 se explican más detalladamente estos niveles y como se han abordado en esta tesis. Se considera que falta un nivel más que no ha sido comentado en estos estudios, y que es el principal objeto de estudio de esta tesis, la traducción de código entre diferentes lenguajes de programación.

Al igual que se realiza en lenguaje natural, el tipo de detección se puede dividir en si el análisis es monolingüe, sobre el mismo lenguaje de programación, o translingüe, entre diferentes lenguajes de programación. Dentro del análisis de detección de reutilización en

[4]Página Web del proyecto Samba `http://www.samba.org/`, página Web del proyecto WINE HQ `http://www.winehq.org/`

código fuente a nivel monolingüe se ha encontrado dos tendencias principales en la comunidad científica. La primera de ellas fue la de utilizar comparación de características del propio código fuente. Como características se consideran por ejemplo número de líneas de un código fuente, número de palabras y caracteres, número de saltos de línea, número de líneas indentadas, cantidad y tipos de tokens.

Por otra parte, la segunda tendencia, y la más utilizada para la detección de reutilización de código fuente ha sido la de comparar las estructuras del código. Por estructura de código se refiere a la estructura sintáctica que representa en código, generalmente en forma de árbol, empezando de un nodo raíz que supone el inicio de ejecución, y seguido de las posibles bifurcaciones que el orden de ejecución pueda tener. Estos árboles sintácticos se pueden representar de distintas formas para realizar comparaciones, como pueden ser en forma de grafo, recorridos en inorden del árbol o secuencias de los nodos importantes (las bifurcaciones).

Los métodos basado en comparación de características fueron los pioneros en la detección de reutilización. Se ha comprobado que son más efectivos cuando los códigos a comparar son copias muy cercanas, y que han sufrido ligeras modificaciones. En el momento que un programador intente evadir su detección, una de las técnicas para disfrazar la reutilización es añadir o modificar código. Los sistemas basados en comparación de características son sensibles a este tipo de cambios en el código por lo que hay pocos trabajos que trabajen bajo este paradigma.

El primer trabajo conocido en el ámbito de la detección de reutilización en código fuente es el propuesto por Halstead [20], en el que trata de encontrar a través de métricas del código, como el número de ocurrencias de operadores y operandos ($n_1$ y $n_2$), número de operadores y operandos distintos ($N_1$ y $N_2$). Halstead trata de encontrar una fórmula basándose en $n_1$ y $n_2$, que aplicándola a algoritmos que realizan la misma tarea, se obtengan resultados similares para los algoritmos similares sin importar el lenguaje de programación. Para ello a través del *ratio* de ocurrencia de operandos entre los operandos distintos encuentra la equivalencia $N_1 = n_1 \log_2 n_1$.

Los resultados obtenidos por Halstead muestran que los algoritmos en diferentes lenguajes que utiliza (ADA, Pilot, Procedure), verifican la fórmula que ha definido, pero tan solo se utilizan 14 muestras positivas, y ninguna muestras negativas. Además, estos algoritmos no han sufrido ningún tipo de modificación para evitar la detección.

Otro trabajo en la línea de comparación de características es el de Selby [30]. Selby propone distintos tipos de características a los utilizados por Halstead, en particular propone medir aspectos como el número de veces que se llama a una función, el tiempo de ejecución de una parte del código, etc. Se ha demostrado que estos sistemas son incapaces de detectar códigos fuente similares pero que han sufrido cambios estructurales, y es una de las principales razones por las que a partir de este estudio se opta por utilizar métodos basados en la estructura del código.

Utilizando la comparación de estructuras, los sistemas de detección son más robustos a la adición o extracción de sentencias redundantes. Para evitar ser detectado por estos sistemas, se necesitaría realizar modificaciones significativas de gran parte del código para reducir el emparejamiento de los segmentos reutilizados.

Siguiendo un esquema básico similar al presentado para comparar textos, el proceso de comparación de código se desarrolla en 3 pasos: ($i$) un preproceso inicial para ignorar comentarios, espacios en blanco, nombres de variables, signos de puntuación, etc. ; ($ii$) representar los códigos fuente en una estructura específica para la comparación y ($iii$) comparar estas representaciones de los códigos.

Los sistemas basados en la estructura del código fuente tienen una serie de características comunes muy marcadas. Una de las principales características es que son dependientes del lenguaje de programación. En la mayoría de los casos, los sistemas que deben conocer de qué forma se pueden generar las bifurcaciones en el árbol, y entender sintácticamente su contenido. Esto impide desarrollar un método general que se pueda aplicar a cualquier lenguaje sin una adaptación previa. Otra característica consiste en que no utilizan todo el código fuente, sino que descartan información que no consideran relevante

como por ejemplo todos los tokens, o bien, que no pertenecen al léxico del lenguaje de programación,y funciones, transforman el código en otro equivalente para simplificar la estructura.

Jankowitz [21] es uno de los primeros autores en proponer utilizar la estructura del código para realizar la detección. Su estrategia se basa en crear el árbol de ejecución, realizar un recorrido en postorden, es decir, representando los nodos terminales hojas con 0 y los nodos de ramas internas con 1. Este recorrido genera una cadena binaria que representa un perfil del árbol de ejecución, y que con algoritmos de búsqueda de subcadenas comunes más largas se pueden identificar rápidamente partes en común entre dos árboles de ejecución representados de esta forma.

El primer trabajo de investigación que genera una herramienta, YAP3, utilizando estructuras de códigos es el de Wise [37]. YAP3 en una fase inicial genera una secuencia de tokens correspondiente a la estructura descartando comentarios, constantes, identificadores, convirtiendo a minúsculas el texto, expandiendo llamadas recursivas y convirtiendo funciones a sus equivalentes más básicas (como en el caso de un bucle *for* por un bucle *while*). En una segunda fase se obtiene la secuencia de tokens común más larga no solapada de dos códigos usando el algoritmo *Karp-Rabin Greedy-String-Tiling* (KR-GST) [22]. Este algoritmo sirve para buscar la subcadena de longitud maximal común entre dos cadenas.

La herramienta más popular es JPlag, comentada en el trabajo de Prechelt et al. [26] llamada JPlag[5]. También realiza un preproceso ignorando los comentarios, identificadores y espacios en blanco. Analiza el código fuente sintácticamente y lo representa por secuencias de tokens. Utiliza una versión optimizada del algoritmo KR-GST para reducir el número de comparaciones, utilizando tablas hash para búsquedas de coste unitario. La similitud entre dos códigos fuente se estima con el porcentaje de caracteres comunes sobre el total de caracteres de los códigos. Esta herramienta ha sido y es un referente con el que se han comparado numerosas herramientas posteriores. Es capaz de procesar diferentes lenguajes de programación como

[5] `https://www.ipd.uni-karlsruhe.de/jplag/`

Java, C#, C, C++ y Scheme además de texto plano en diferentes idiomas (inglés, alemán, francés, español portugués). La comparación de códigos aunque soporte distintos lenguajes se realiza a nivel monolingüe.

Otro trabajo en la línea de comparar estructuras sintácticas del código fuente es el de Xiong et al. [38] donde se transforma código escrito en el lenguaje de programación C a un lenguaje intermedio CIL[6] que contiene pocas estructuras diferentes para construir un árbol y sin redundancias. Se genera un árbol sintáctico abstracto con el plugin CDT de la plataforma de desarrollo Eclipse[7]. Utilizando la técnica de dividir en $n$-gramas la representación sintáctica del árbol, junto a la ventana deslizante, se comparan dos representaciones en árbol de dos códigos utilizando el coeficiente de Jaccard. Este trabajo da lugar a la herramienta BUAA_Antiplagiarism, que utilizando 4-gramas con un conjunto de 40 códigos fuente, obtiene mejores resultados que la herramienta JPlag.

Un trabajo a remarcar por ser capaz de trabajar en lenguajes tan distintos como el lenguaje de alto nivel C, o los lenguajes de bajo nivel (lenguaje ensamblador) para los chips Motorola MC88110 e Intel P8080E es el realizado por Rosales et al. [28]. Se presenta una herramienta , llamada PK2, que permite detectar plagio entre estudiantes en diferentes asignaturas. Para ello, a partir de la estructura del código, obviando identificadores y comentarios, y sustituyendo las palabras reservadas por símbolos internos, crean una firma de los códigos. Estos códigos se comparan bajo cuatro criterios: ($i$) búsqueda subcadenas contiguas comunes más largas; ($ii$) longitud acumulada, localizar las subcadenas comunes más largas; su longitud se acumula; y el primer carácter de cada subcadena se descarta en la siguiente comparación, repitiendo esto hasta que no queden más subcadenas comunes; ($iii$) el resultado del criterio ($ii$) normalizado sobre cien; y ($iv$) medición el porcentaje de palabras reservadas comunes entre ambos códigos. Se considera la intersección de los histogramas. Se aplicó esta herramienta durante 14 años, concluyendo que el $4,6\,\%$ de los alumnos había copiado código.

[6] `http://sourceforge.net/projects/cil`
[7] `http://www.eclipse.org`

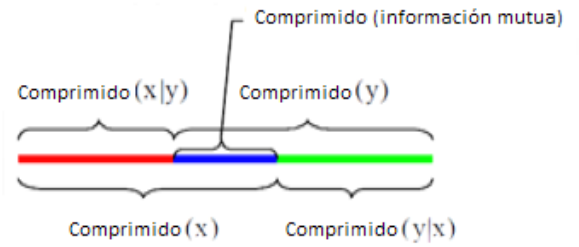


Figura 2.1: Representación de la información mutua dentro del espacio de compresión: dos cadenas (x e y) son representadas por dos longitudes (comprimido(x) y comprimido(y)). Las cadenas comparten información mutua representada por comprimido(información mutua). Comprimido(x—y) es más pequeña que comprimido(x) porque existe una contribución de y.

La herramienta MOSS, es la propuesta en el trabajo de Schleimer et al. [29] que trabaja con distintos lenguajes de programación, pero a nivel monolingüe. MOSS procesa documentos conteniendo Java, C, C++, Pascal, ADA o texto plano. Para detectar similitudes en partes del código usa el algoritmo de winnowing, seleccionando y comparando las huellas o fingerprints de los documentos que los contienen.

La propuesta de Zhang et al. [39] tiene un enfoque distinto a las aproximaciones expuestas anteriormente, está basada en la información mutua. A partir de la estructura del código, se genera una tokenización donde se han eliminado los comentarios, las palabras reservadas del lenguaje se han traducido a símbolos internos y los tipos de datos, variables y constantes se traducen al mismo token. Con dicha tokenización se obtiene una secuencia de caracteres que representa el código fuente. Para estimar la similitud, se utiliza el algoritmo de compresión Lempel-Ziv (LZ77) [40]. como una aproximación a la complejidad de Kolmorov, como refleja el estudio de Yang y Kieffer [12]. De ambos diccionarios de compresión se obtiene la información que tienen en común como se muestra en la figura 2.1. Se realiza una comparación de este algoritmo con JPlag y MOSS con casos de plagio reales. Estos casos reales a posteriori han sido manipulados artificialmente para ocultar la copia en cinco niveles de modificación según los niveles propuestos por Faidhi

y Robinson. Los resultados obtenidos reflejan que JPlag y MOSS funcionan mejor que su algoritmo en casos que han sufrido un nivel de modificación bajo, y para casos de alto nivel de modificación su sistema resulta más efectivo.

## 2.3. Detección automática de plagio en código fuente translingüe

En la sección anterior se ha comprobado que existen diferentes tipos de aproximaciones para la detección de reutilización en código fuente a nivel monolingüe. Pero al igual que ocurre con textos escritos en lenguaje natural cabe la posibilidad que la fuente de la reutilización esté escrita en un lenguaje distinto del documento sospechoso.

Además, los lenguajes de programación tienen un vocabulario más corto, y es frecuente que tengan elementos en común o equivalentes. La facilidad de un programador para utilizar y entender un nuevo lenguaje posibilita la reutilización entre lenguajes de programación. Además, la existencia de repositorios de proyectos en la Web favorece al proceso de "copiar y pegar" un código fuente.

Por otra parte también existe software capaz de traducir código funcional de un lenguaje de programación a otro, como es el caso de Tangible Software Solutions Inc[8]. Éste es capaz de traducir código entre los lenguajes: Visual Basic, C#, C++ y Java. Otro ejemplo de traductor entre distintos lenguajes de programación se encuentra en developerFusion Ltd.[9] que es capaz de traducir de C# y Visual Basic .NET a Python y Ruby. También traduce entre el par C# y Visual Basic .NET. La proliferación de herramientas que permiten traducir código fuente, facilita aún más la reutilización de códigos desarrollados e incluso evita al usuario tener que aprender nuevos lenguajes de programación.

El primer trabajo conocido que detecta similitudes entre códigos fuente escritos en diferentes lenguajes de programación es el de

[8] http://tangiblesoftwaresolutions.com/
[9] http://www.developerfusion.com/tools/

Halstead [20]. Este trabajo, que ya ha sido comentado en la sección anterior, fue uno de los pioneros en la detección en código fuente. Halstead ya trataba de encontrar una fórmula común para calcular la similitud entre las distintas implementaciones de un algoritmo en varios lenguajes de programación, en concreto ADA, Pilot y Procedure.

La segunda aproximación conocida es la de Arwin y Tahaghoghi [1]. Este trabajo genera la herramienta XPlag que permite detectar reutilización entre múltiples lenguajes utilizando código intermedio generado por un compilador. Para ello necesitan un compilador que permita más de un lenguaje. En este caso se utiliza el compilador *GNU Compiler Collection* (GCC)[10] que soporta los lenguajes C, C++, Java, Fortran y Objective C, aunque en este estudio sólo se utilizan los lenguajes C y Java, traduciéndose al lenguaje intermedio *Register Transfer Language* (RTL). Por otro lado utilizan un corpus monolingüe con dos partes, una en el lenguaje C y otra en el lenguaje Java. Generan 10 casos simulados de reutilización translingüe utilizando el traductor Jazillian de código fuente de C a Java, el cual no se encuentra ya disponible. Para el proceso de detección realizan un mapeo de los tokens del texto a símbolos internos y descartan constantes, nombres de variables y tipos de datos. Después de este preproceso, realizan el pesado de términos de los $n$-gramas junto con la bolsa de palabras y miden la similitud entre dos códigos intermedios con la función de similitud Okapi BM25 [27]. Esta función de similitud es altamente efectiva en búsqueda general de texto. Okapi BM25 extiende la aproximación *idf* y *tf*, que se explicaran en la sección 3.1. La ecuación 2.3 muestra la definición formal de la fórmula, siendo $k_1 = 1,2$, $b = 0,75$, $k_3 = 2$, $|d|$ representa la longitud del documento y $L_{avg}$ la longitud media de los documentos de la colección.

Por el hecho de no conocer ningún sistema anterior que permita detectar reutilización de código entre diferentes lenguajes de programación, realizan un estudio para obtener la mejor combinación de $n$-gramas, siendo trigramas la mejor. La principal desventaja de XPlag es que es dependiente de un compilador común a todos los

[10] `http://gcc.gnu.org/`

lenguajes. El corpus utilizado en este estudio no se ha conservado en su totalidad, tan solo los corpus monolingües y sin etiquetar los casos de reutilización por lo que imposibilita la comparación de otros sistemas con los resultados publicados en este trabajo.

$$\sum_{t \in d_q} idf_t \cdot \alpha_{t,d} \cdot \beta_{t,d_q}$$

$$\alpha_{t,d} = \frac{(k_1+1)tf_{t,d}}{k_1((1-b)+b \cdot \frac{|d|}{L_{avg}})+tf_{t,d}} \qquad \beta_{t,d_q} = \frac{(k_3+1)tf_{t,d_q}}{k_3+tf_{t,d_q}} \quad (2.3)$$

La falta de estudios dentro del campo de la detección de reutilización de código translingüe, es la principal motivación de la realización de este trabajo de investigación. En el siguiente capítulo se detallará la aproximación utilizada en este estudio para abordar reutilización en general, ya sea a nivel monolingüe o a nivel translingüe. Se trata de una aproximación que no depende de ningún lenguaje de programación ni de ningún compilador por lo que es aplicable a cualquier lenguaje de programación.

# Capítulo 3

# Modelos propuestos

En este capítulo se describen los distintos modelos utilizados para detectar reutilización de código fuente para este trabajo de investigación. Para ello se ha hecho uso de técnicas efectivas aplicadas para detectar reutilización de textos escritos en lenguaje natural [25]. En la sección 3.1 se describe un primer modelo utilizado para detectar reutilización de código fuente, comparando códigos a nivel de documento. La sección 3.2 contiene los detalles de un segundo modelo que compara códigos a nivel de fragmento basándose en la estructura del modelo anterior.

## 3.1. Modelo a nivel de documento

En la sección 2.1 se han descrito distintos modelos para abordar el problema de reutilización de textos ya sea a nivel monolingüe [7, 2, 29] como a nivel translingüe [25]. El propósito de este trabajo de investigación es comprobar si las herramientas que han dado buenos resultados aplicadas sobre textos escritos en lenguaje natural son también efectivas sobre códigos fuente. En este trabajo se va a considerar al texto escrito y al código fuente, como semejantes, ya que ambos estan escritos en un lenguaje, natural o de programación, poseen un código semiótico estructurado, para el que existe un contexto de uso y tienen ciertos principios combinatorios formales.

Considerados los lenguajes de programación al mismo nivel que el lenguaje natural, se procede a explicar el enfoque utilizado para

la detección de reutilización en el marco de este trabajo de investigación basado en modelos aplicados a textos.

A la hora de crear un sistema que detecte reutilización de código fuente es necesario plantear qué tipos de estrategias utiliza un programador para evadir ser detectado y cómo poder evidenciar la reutilización a pesar de estas estrategias. Varios autores ya se plantearon las diferentes estrategias como por ejemplo Whale [36] o Faidhi y Robinson [13]. Particularmente, este último trabajo realiza una clasificación según la dificultad para el programador para ocultar que está reutilizando un código fuente. Estos niveles van desde la copia íntegra sin realizar ninguna modificación sobre el código, hasta cambios de estructura general del código fuente como se puede ver en la figura 3.1.

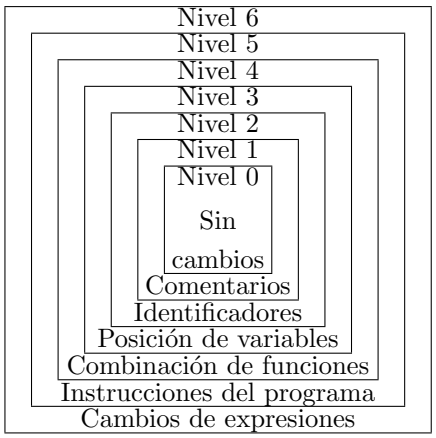


Figura 3.1: Niveles de modificación de códigos fuente para ocultar el plagio.

El nivel 0 consiste en la copia verbatim, es decir, sin realizar ningún cambio al código fuente. Los cambios de comentarios como inserción, borrado, o modificación junto con cambios de indentación (cambios en la alineación del código fuente y que no suponen ninguna variación en la compilación ni ejecución del código fuente), como añadir o quitar espaciados, tabulaciones o saltos de línea constituyen el nivel 1. En el nivel 2 se considera el cambio de identificadores, ya sea cambiar los nombres o convertir a mayúsculas el identificador.

El nivel 3 consiste en cambiar la posición de las declaraciones, ya sea variables o funciones de una parte del código a otra, y también añadir variables o funciones que no se van a utilizar.

A partir de este nivel, los cambios van a ser más complejos y van a afectar a la forma de ejecutarse del código fuente internamente aunque las modificaciones son realizadas sin intención de cambiar el resultado final, sólo la apariencia del código fuente. La división en funciones más pequeñas y combinación de funciones forman parte del nivel 4 propuesto por Faidhi y Robinson. En el nivel 5 se incluye el cambio de las instrucciones de un código fuente, por ejemplo cambiar la instrucción *while* por una que puede realizar la misma tarea como la instrucción *for*. Finalmente, en el último nivel se incluye cambiar partes de la estructura del código fuente, como por ejemplo reorganizar secciones de código que no tengan dependencia y no sea necesario ejecutarlas en un cierto orden.

Hay que considerar que cada nivel representado en la figura 3.1 contiene todas las modificaciones de cada nivel inferior además de las descritas en ese nivel, por lo que cada nivel es más complicado que el anterior tanto para la parte que trata de ocultar la reutilización de código fuente como para la parte que trata de detectarla.

### 3.1.1. Preproceso

Inicialmente, se considera el código fuente como un vector de caracteres al igual que se hace en lenguaje natural. De este vector se eliminan los saltos de línea, los espacios en blanco y se convierten todos los caracteres a minúsculas. Así se evita que nuestro sistema se vea afectado por recursos habituales para evadir la detección de copia como son modificar las indentaciones del código fuente, añadir espacios en blanco (nivel 1) o cambiar el nombre a las variables (nivel 2). Una vez realizado este preproceso se obtiene una secuencia de caracteres en minúsculas, sin espacios, ni saltos de línea.

Para que el sistema no sea sensible al nivel 3 de modificación del código, cambios de posición de las variables declaradas o añadir variables adicionales, el código preprocesado se divide en $n$-gramas

de caracteres. Estos $n$-gramas se almacenan en un bolsa de palabras, que en este caso seran $n$-gramas. Además, en esta bolsa de palabras se va a guardar la frecuencia de aparición de cada $n$-grama.

Al perder la localización espacial de los $n$-gramas no importará si un programador sitúa una función que estaba al principio en el código fuente original, al final o en cualquier otra parte del mismo porque con la división en $n$-gramas no se tiene en cuenta la situación espacial de cada parte y el conjunto de $n$-gramas finalmente será el mismo. Este tipo de modificación, cambiar de posición de una función o una variable, también está descrito en el nivel 3 de modificaciones para ocultar la copia manualmente. Será necesario realizar un proceso de normalización porque al utilizar la frecuencia de aparición creamos una bolsa de palabras que sólo permitirá comparar de manera equitativa los términos dentro de dicha bolsa. En el momento que se desee comparar dos bolsas de palabras correspondientes a dos códigos fuente distintos, la comparación entre ambas va a ser desigual porque en la mayoría de situaciones va a haber un código fuente más extenso que otro.

Para este trabajo de investigación se ha utilizado tf y tf-idf por ser las más utilizadas y con mayor rendimiento en aplicaciones tales como minería de datos y recuperación de información (acrónimo en inglés *IR*). Tf consiste en calcular la frecuencia de los términos respecto al documento. Esto se consigue dividiendo la frecuencia de aparición de cada término por el total de términos que ha aparecido en un documento o en nuestro caso, en el código fuente. En oposición a este método de pesado surge idf (en inglés *inverse document frequency*), que consigue dar mayor importancia a los elementos menos frecuentes, términos que están en un documento y no en los otros y por tanto podrán caracterizarlos mejor. Dado que tf destaca las palabras relevantes dentro del documento y idf destaca las palabras relevantes entre documentos, el método de pesado tf-idf, tiene ambas propiedades y consiste en el simple producto de ambas medidas. En los experimentos, además de utilizar tf-idf por ser el más completo de los tres, también se ha utilizado el método tf, porque al añadir un nuevo código no es necesario recalcular sus valores para todo el conjunto de documentos.

### 3.1.2. Estimación de similitud

Una vez estimadas las frecuencias de dos códigos fuente utilizando un sistema de pesado normalizado como son tf o tf-idf, nos interesará conocer la similitud existente entre ambos códigos. Si nuestro sistema estima que es alta la similitud de dos códigos fuente, probablemente cuando un revisor analice manualmente los dos códigos encontrará mucha similitud entre estos códigos fuente. Siguiendo la tendencia de aplicar técnicas y modelos ya contrastados en lenguaje natural, se ha elegido utilizar la medida de similitud del coseno. Destaca por su simplicidad de cálculo y buen rendimiento en el campo de la recuperación de información.

Para aplicar esta medida de similitud es necesario disponer de dos vectores $\vec{a}$ y $\vec{b}$ dentro de un mismo espacio vectorial $\mathcal{D}$. En nuestro caso, disponemos de dos bolsas de palabras, y cada $n$-grama de la bolsa de palabras se considerará como una característica de $\mathcal{D}$, por lo que el espacio vectorial estará compuesto por la unión de todos los $n$-gramas contenidos en ambas bolsas. El valor obtenido con el sistema de pesado será el valor de la característica en $\vec{a}$ y $\vec{b}$. En una bolsa de palabras puede haber $n$-gramas no contenidos en la otra, esta ausencia se representará con un valor 0.

Después de tener caracterizadas las bolsas de palabras como vectores dentro de un espacio vectorial, la función de similitud del coseno consiste en medir el coseno del ángulo que forman los dos vectores en el espacio vectorial. La fórmula del coseno se puede deducir a partir de la ecuación del producto euclídeo de dos vectores (Ecuación 3.1):

$$\mathrm{A} \cdot \mathrm{B} = \|\mathrm{A}\| \, \|\mathrm{B}\| \cos(\theta) \tag{3.1}$$

Por lo que finalmente el cálculo del coseno equivale a calcular el producto escalar de dos vectores de documentos $A$ y $B$ y dividirlo por la raíz cuadrada del sumatorio de los componentes del vector $A$

multiplicada por la raíz cuadrada del sumatorio de los componentes del vector $B$ como se desglosa en la ecuación 3.2.

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}} \tag{3.2}$$

Los vectores correspondientes a dos códigos fuente cuyas bolsas de palabras sean idénticas, tendrán las mismas características y formarán un ángulo de 0°. Al realizar el coseno del ángulo obtendremos como valor de la máxima similitud de 1, $\cos(0°) = 1$. En el caso contrario, cuando no comparten ningún $n$-grama en las bolsas de palabras, y por lo tanto ninguna característica en el espacio vectorial, los ángulos de los vectores va a ser perpendicular, o sea 90°, y su valor de coseno va a ser 0, $\cos(90°) = 0$.

Utilizando la función del coseno disponemos de una medida de similitud sencilla, normalizada en el rango 0-1 y que va permitir comparar numéricamente como de similares son dos códigos fuente.

## 3.2. Modelo a nivel de fragmento

En la sección 3.1, se han detallado las características del modelo a nivel de documento. Como se ha comentado también en dicha sección, es necesario detectar todos los posibles modos de reutilizar código fuente. Una de las ideas que no se ha propuesto en trabajos como el de Whale [36] o Faidhi y Robinson [13] es si el programador en el momento de reutilizar un código, va a reutilizar todo el código, o parte de él; si su código va a ser sólo código reutilizado o una mezcla de código propio con el reutilizado. Es por ello que surge la necesidad de la idea de fragmentos reutilizados. Es posible que tan solo se reutilice una parte muy pequeña de un código, como un algoritmo, una función, etc. y el resto del código no tenga nada que ver con la fuente de la reutilización.

### 3.2.1. Preproceso

Partiendo de esta idea, inicialmente se aplica el mismo preproceso que en el modelo a nivel de documento eliminando los saltos de línea, los espacios en blanco y convirtiendo todo el texto que esté en mayúsculas a minúsculas. Como resultado obtenemos un vector de caracteres que representa el código. Para abordar el problema, tras el preproceso se divide el vector en fragmentos que consistirán en secuencias de caracteres de longitud $l$ (a partir de ahora se usará el término ventana).

Una ventana en el contexto de minería de datos significa un bloque de caracteres contiguos de una cierta longitud. En el modelo a nivel de fragmento se pretende comparar fragmentos: detectar aquellos fragmentos que son muy similares, rechazar los que no sean tan similares para eliminar los que puedan generar ruido, y a partir de los fragmentos similares establecer una similitud global.

Partiendo del primer carácter, se toman los primeros $l$ caracteres en la primera ventana; a continuación para la siguiente ventana se desplaza la posición inicial un número $d$ de caracteres (desplazamiento). En el caso de la siguiente ventana, su inicio se situa en la posición de la ventana anterior más $d$ posiciones. Se considera que la ventana se ha desplazado $d$ caracteres, tal que $d$ puede tomar valores menores o iguales que $l$. Si el desplazamiento toma valores menores que $l$ se produce el fenómeno de solapamiento (en inglés *overlapping*), es decir, las ventanas están compartiendo caracteres al no desplazarse tanto como la longitud de la ventana anterior.

Una vez divididos los códigos en ventanas se realiza el proceso de división en $n$-gramas, se guardan los $n$-gramas en bolsas de palabras y se realiza el pesado con métodos como tf o tf-idf como en el sistema a nivel de documento. Se ha de incluir un proceso de normalización de las bolsas de palabras, dado que la última ventana de un código fuente puede ser menor que el resto de ventanas.

En el caso del modelo a nivel de documento se disponía de dos bolsas de palabras para comparar dos códigos, mientras que en el

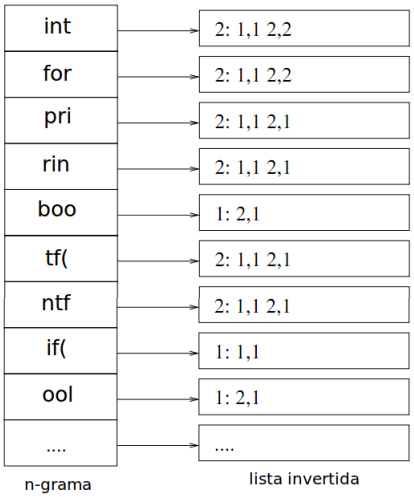


Figura 3.2: Lista de $n$-gramas invertida con los 9 primeros trigramas de códigos fuentes. Cada entrada muestra la cantidad de ventanas que contienen el $n$-grama, el identificador individual de la ventana y las ocurrencias del $n$-grama en cada ventana.

modelo a nivel de fragmento se disponen de tantas bolsas de palabras como se haya dividido los dos códigos fuente. Supongamos que disponemos de un código fuente $A$ y otro $B$, de los cuales se ha extraído $N$ y $M$ ventanas respectivamente ($A_1 \dots A_N$ y $B_1 \dots B_M$). En este modelo se va a comparar las $N$ ventanas de $A$ contra las $M$ de $B$ que darán como resultado $N \times M$ valores de similitud.

Por el hecho de que se va a comparar cada ventana de $A$ contra el resto de ventanas de $B$, con el sistema de guardar una bolsa de palabras por cada ventana se va a repetir multitud de cálculos al aplicar la fórmula del coseno 3.2 como por ejemplo: $\sum_{i=1}^{n} (A_i)^2$ ó $\sum_{i=1}^{n} (B_i)^2$ para cada par de ventanas.

Por ello se ha realizado el cálculo de forma diferente y más eficiente que no almacena la información en bolsas de palabra, sino que está basado en los $n$-gramas; el índice invertido de $n$-gramas se muestra en la figura 3.2.

El índice invertido de $n$-gramas consiste en generar un diccionario con todos los $n$-gramas que hay en todas las ventanas que se van

a comparar. Cada entrada del diccionario enlazará con una lista de tuplas de números. Estas tuplas estarán constituidas por el identificador de la ventana y el número de ocurrencias del $n$-grama en dicha ventana. El uso del índice invertido permite que además de comparar dos códigos fuente a la vez, se puedan comparar más de dos, añadiendo tantas ventanas como se desee.

### 3.2.2. Estimación de la similitud

Para estimar la similitud en el modelo de ventanas se va a utilizar la misma base matemática que para el modelo a nivel de documento, es decir, en la fórmula del coseno. La información de las ventanas no se almacena como una bolsa de palabras, sino como una lista invertida con lo cual para el cálculo de la similitud se desglosa la fórmula de la siguiente forma:

($i$) Para realizar el cálculo del numerador ($\sum_{i=1}^{n} A_i \times B_i$) se almacenarán datos en una matriz de tamaño *número_total_de_ventanas* $\times$ *número_total_de_ventanas*. Dado que la función del coseno cumple la propiedad de simetría (los dos vectores van a formar el mismo ángulo en el rango 0°-90°), el espacio final necesario es la mitad por la propiedad conmutativa del producto que va a devolver el mismo valor para el producto del para ($i$,$j$) como para el par ($j$,$i$), por lo tanto se puede almacenar en una matriz triangular.

($ii$) Para almacenar $\sum_{i=1}^{n} (B_i)^2$, se utilizará un vector de tamaño igual al *número_total_de_ventanas*. Una vez calculados los resultados de esta estructura se calculará la raíz cuadrada de los valores.

Es necesario identificar qué procesos se van a poder realizar recorriendo el diccionario de $n$-gramas una sola vez. Finalmente, para obtener la similitud entre todas las ventanas, para cada par de ventanas se divide el resultado en el paso ($i$) por el producto del valor de las dos ventanas calculado en ($ii$) .

Llegado a este punto, se dispone de tantos valores de similitud entre ventanas, como pares de ventanas se han comparado. El propósito del modelo a nivel de ventana es el de extraer una similitud entre pares de documentos partiendo de comparaciones entre ventanas.

Una primera aproximación para obtener la similitud de dos códigos fuente consiste en calcular la media de las similitudes obtenidas entre sus ventanas. Sin embargo, en este trabajo de investigación se ha optado por establecer un umbral $t$ para descartar los falsos positivos siendo la similitud final entre dos códigos fuente el porcentaje de pares de ventanas entre los dos códigos que han superado un valor de similitud $t$. Obtendrán un valor 0 aquellos pares de códigos fuente cuyas comparaciones, ningún par de ventanas supere el umbral y un valor 1 si todos los pares de ventanas superan dicho umbral.

# Capítulo 4

# Evaluación de los modelos

En el capítulo anterior se han propuesto dos aproximaciones para la detección de código fuente translingüe basada en el sistema de pesado con $n$-gramas y el modelo de representación de la información en bolsas de palabras. En la primera aproximación se genera una bolsa de palabras para todo el código fuente, y en la segunda aproximación se genera una por cada ventana en las que dividimos el código. Estas bolsas de palabras son comparadas utilizando la medida de similitud del coseno, dando un valor que permite clasificar la similitud de dos códigos fuente.

En este capítulo se van a detallar los corpus utilizados para evaluar las aproximaciones propuestas anteriormente, las medidas para evaluar estas aproximaciones y los resultados obtenidos en los diferentes experimentos. Utilizando el corpus de la subsección 4.1.1 se han ajustado los parámetros de las aproximaciones, y con el corpus de la subsección 4.1.2 se han comparado los resultados de las aproximaciones con el sistema JPlag.

## 4.1. Corpus utilizados en los experimentos

Este trabajo de investigación nace en la asignatura Aplicaciones de la Lingüística Computacional del Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital. El principal inconveniente encontrado es el de la obtención de un corpus. Para solventar el problema se ha optado por la creación de dos recursos: un primer recurso a partir de un proyecto con implementaciones en

tres lenguajes de programación y un segundo recurso que contiene códigos fuente monolingüe y multilingüe facilitado por los autores Arwin y Tahaghoghi.

### 4.1.1. Corpus SPADE

Ante la falta de recursos para probar las aproximaciones descritas, se ha creado el corpus llamado SPADE[1]. La plataforma SPADE sirve para desarrollar sistemas multiagente u organizaciones basada en la tecnología XMPP/Jabber y escrito en el lenguaje Python. Este corpus ha sido creado a partir del código fuente de la Interfaz de programación de aplicaciones o API (del inglés *Application Programming Interface*). Como parte de la asignatura Sistemas Multiagente en el marco del máster, se propuso desarrollar la parte de la API en los lenguajes Java y C++ basándose en la versión existente en el lenguaje Python. Existe una reutilización parcial entre los códigos C++ a Java; ambos lenguajes realizan la misma tarea que los códigos escritos en Python, aunque sólo existe reutilización de Python a C++. El caso Python→C++ representa un ejemplo real de reutilización translingüe, y el caso Python→Java representa un caso simulado. Sin embargo, C++−Java representa la reutilización triangular entre los tres lenguajes teniendo Python como pivote. En la tabla 4.1 se muestran algunas estadísticas del corpus.

El conjunto de códigos escritos en el lenguaje Python consiste en sólo 4 códigos fuente con 10.503 tokens. Estos códigos escritos por los colaboradores del proyecto SPADE, mantienen funcionalidad similar a los escritos en los lenguajes C++ y Java. Los códigos escritos en el lenguaje C++ consisten en 5 códigos fuente con un total de 1.318 tokens. La tercera parte del corpus escrita en Java consiste en 4 códigos fuente con 1.100 tokens. De los códigos fuente escritos en el lenguaje Python Se ha reutilizado exactamente la misma funcionalidad que tienen los códigos escritos en el lenguaje Java. Los códigos escritos en C++ y Java tienen la misma funcionalidad y ambos han reutilizado partes del código escrito en Python, que además posee otras funcionalidades. Esto explica la diferencia de tamaño entre los códigos fuente escritos en Python y los escritos

[1] `http://code.google.com/p/spade2/`

| Lenguaje | C++ | Java | Python |
|---|---|---|---|
| Tokens | 1.318 | 1.100 | 10.503 |
| Longitud media de los tokens | 3,46 | 4,52 | 3,24 |
| Tipos | 144 | 190 | 671 |
| Códigos fuente | 5 | 4 | 4 |
| Tipos por código fuente | 28,8 | 47,5 | 167,75 |

Tabla 4.1: Estadísticas del corpus SPADE.

en el resto de lenguajes.

### 4.1.2. Corpus CL-AT++

Para este trabajo de investigación se ha utilizado además del descrito en la subsección 4.1.1, una versión adaptada y enriquecida (*Cross-Language-Arwin-Tahaghoghi++* o CL-AT++ del corpus utilizado por Arwin y Tahaghoghi. CL-AT++ es una versión adaptada por problemas de confidencialidad, se ha procesado el corpus original para anonimizar la colección de códigos fuente. Se considera como versión enriquecida porque no fue posible la obtención de las muestras que se consideraban reutilización translingüe y el conjunto de códigos fuente en el lenguaje Java es más amplio.

El corpus original del estudio de Arwin y Tahaghoghi, disponía de 79 códigos fuente escritos en el lenguaje C, 107 en el lenguaje Java y 20 códigos en Java resultado de la traducción de 10 escritos en el lenguaje C. Para este proceso de traducción se han utilizado la herramienta Jazillian que actualmente no se encuentra disponible. En cambio en CL-AT++ se dispone de los mismos 79 códigos fuente escritos en C, 259 códigos escritos en Java y como muestras para experimentar a nivel translingüe se han traducido los 79 códigos escritos en C a Java utilizando el traductor de código fuente C++ to Java Converter version 2.7 tool from Tangible Software Solutions Inc.[2]. Una vez establecidas las diferencias con el corpus original, tal que el conjunto de códigos fuente escritos en Java y el conjunto de casos de reutilización son distintos del original, es necesario definir los *gold standard*, es decir los pares de códigos fuente

[2] `http://tangiblesoftwaresolutions.com/Product_Details/CPlusPlus_to_Java_Converter_Details.html`

que se considerarán como reutilización para poder evaluar nuestras aproximaciones.

Arwin y Tahaghoghi proporcionan un gold standard determinado manualmente sólo para el conjunto de códigos fuente escritos en C. No obstante, se ha detectado que algunos pares de este gold standard no se pueden considerar como casos de reutilización. Por esta razón, se ha optado por realizar una revisión manual de todos los pares de códigos fuente y establecer un gold standard con el fin de tener un criterio fiable y unificado para futuros experimentos. Para llevar a cabo este proceso, dos revisores que desconocían el gold standard proporcionado por Arwin y Tahaghoghi, han marcado de forma independiente los casos de reutilización. Así pues, el conjunto final de casos de reutilización viene dado por aquellos pares de códigos fuente que hayan sido marcados por dos o más revisores, considerando como revisores a: los autores del corpus original como un revisor, y los dos nuevos revisores independientes. El gold standard final lo constituyen 27 pares de códigos fuente que se consideran reutilizados porque al menos dos de tres revisores así lo han determinado.

Del conjunto Java no se dispone de gold standard, dado que no coincide con el conjunto del corpus original y se desconoce el gold standard de este conjunto. Dado el elevado número de pares de códigos fuente que existen en un conjunto de 259 códigos fuente, se ha optado por determinar el gold standard haciendo uso de una herramienta monolingüe contrastada y que la mayoría de estudios utilizan como referencia como es JPlag, situando el umbral de similitud en el 40 %. JPlag descarta los pares de códigos fuente por debajo de ese umbral. Así pues, el gold standard está constituido por todos los pares de códigos fuente que tengan una similitud de 40 % o superior. En este caso son 98 pares de códigos fuente que cumplen esta condición. Finalmente, para el conjunto de códigos fuente traducidos, se tomará como gold standard los pares formados por el código escrito en C y su correspondiente traducción.

## 4.2. Medidas de evaluación

En la sección anterior se han presentado dos corpus multilingüe describiendo tanto el tamaño como el ámbito de los mismos. Para determinar la calidad de un sistema informático se necesita medir las bondades de éste respecto al gold standard. Para ello existen métricas muy extendidas en el área de la recuperación de información como son la precisión y la cobertura (en inglés *precision* y *recall*). La precisión en IR sirve para medir el número de documentos relevantes recuperados en función del número de documentos recuperados. En el caso de detección de reutilización de código fuente, la precisión es el número de pares de códigos fuente reutilizados detectados entre el número de pares de códigos fuente detectados. La relevancia de los pares de códigos vendrá dada por la estimación de similitud entre éstos.

Por otro lado, la cobertura en IR se define como el porcentaje de documentos relevantes recuperados del total de los relevantes. En el problema de la detección de reutilización en código fuente, la cobertura consiste en el total de pares de códigos reutilizados detectados entre el total de los pares de códigos reutilizados del corpus. La precisión y la cobertura están interrelacionadas entre sí tal como se muestra en la figura 4.1. Tanto precisión como cobertura dependen del total de pares de códigos reutilizados detectados (en IR documentos relevantes recuperados).

El cálculo de la precisión y la cobertura del sistema sobre el corpus SPADE no aporta la suficiente información para poder analizar los resultados debido al tamaño de éste corpus, por lo que se ha optado por utilizar una métrica distinta. Dado que la intención de utilizar el corpus pequeño no es la de comparar con aproximaciones correspondientes a otros trabajos de investigación sino la de establecer los mejores parámetros con la mayor información se optó por utilizar una medida basada en la posición del ranking de similitud entre un código fuente y los el resto de códigos de la colección. Para ello un código fuente $c_a$ escrito en un lenguaje de programación $A$, se compara contra todos los códigos escritos en un lenguaje $B$, siendo $A \neq B$. Se ordenan de mayor a menor los pares similares según


Figura 4.1: Representación del conjunto de pares de códigos fuente según si son detectados y si han sido recuperados. La precisión analiza la relación entre los dos conjuntos de la parte izquierda, mientras que la cobertura analiza la relación entre los conjuntos de la parte superior.

el valor de similitud obtenido. En el corpus SPADE es sabido que cada código fuente en un lenguaje $A$ tiene uno correspondiente en $B$, por lo que el valor de esta similitud mide la posición de $c_b$ dentro del ranking. Este valor se obtiene para cada código fuente, por lo que se mostrará el valor de esta medida en función a la media de la posición media junto con la desviación típica.

## 4.3. Experimentos

En las siguientes subsecciones se exponen los experimentos realizados utilizando los modelos a nivel de documento y a nivel de fragmento. Con el fin de explorar las características entre códigos fuente se han aplicado los dos modelos propuestos sobre los corpus descritos en la sección 4.1. Se ha explorado las distintas combinaciones de parámetros que mejor se ajustan al problema.

Además se han analizado las distintas partes del código, como pueden ser los comentarios, las palabras reservadas del propio lenguaje de programación que ha utilizado el programador, los nombres de variables, funciones y texto de las cadenas que puedan aportar información sobre la forma de programar.

### 4.3.1. Ajustes en la aproximación a nivel de documento

Para conseguir los mejores resultados, aplicando la aproximación de detección automática de reutilización de código a nivel de documento, se debe ajustar el tamaño de los $n$-gramas, el pesado de éstos y las partes relevantes del código que debemos considerar. Este es el objetivo principal de este experimento inicial. Para ello se ha utilizado el corpus SPADE descrito en el apartado 4.1.1. Debido al tamaño del corpus, la métrica de evaluación más apropiada es la del ranking descrita en la sección 4.1.2.

En estudios previos sobre detección de reutilización de código han apuntado que considerar comentarios e identificadores empeoran la calidad de sus resultados. Se han realizado una serie de experimentos con el fin de determinar y demostrar las partes del código que se deben considerar para conseguir los mejores resultados. En concreto, se ha ejecutado el sistema propuesto bajo los siguientes supuestos, tomando: el código fuente entero sin modificar (*full code*); el código fuente sin los comentarios (*fc-without comments*); considerando sólo las palabras reservadas del lenguaje (*fc-reserved words only*); sólo comentarios contenidos en el código fuente (*comments only*); el código fuente sin las palabras reservadas (*fc-without rw*); y el código fuente sin comentarios ni palabras reservadas (*fc-wc-wrw*).

| Experimentos/$n$-gramas | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| full code | 2,89 ± 1,10 | **1,00 ± 0,00** | **1,00 ± 0,00** | **1,00 ± 0,00** | **1,00 ± 0,00** |
| fc-without comments | 2,89 ± 1,10 | **1,00 ± 0,00** | **1,00 ± 0,00** | **1,00 ± 0,00** | **1,00 ± 0,00** |
| only comments | 3,43 ± 1,17 | 2,29 ± 1,57 | 2,29 ± 1,57 | 2,57 ± 1,49 | 2,71 ± 1,48 |
| fc-only reserved words | 2,56 ± 1,16 | 1,33 ± 0,66 | 1,56 ± 0,83 | 1,67 ± 0,82 | 1,67 ± 0,82 |
| fc-without rw | 2,67 ± 1,05 | 1,78 ± 1,22 | 1,44 ± 0,83 | 1,44 ± 0,83 | 1,56 ± 1,06 |
| fc-wc-wrw | 2,67 ± 1,05 | 1,89 ± 1,28 | 1,44 ± 0,83 | 1,44 ± 0,83 | 1,44 ± 0,83 |

Tabla 4.2: Resultados obtenidos comparando los documentos del corpus SPADE escritos en C++ y en Java a nivel de documento, dónde cada documento es pesado utilizando tf. La tabla refleja la media y desviación típica de las posiciones de los documentos considerados reutilizados respecto del documento que está siendo analizado.

En las siguientes tablas se muestran los resultados de los experimentos. De las seis tablas disponibles, las tres primeras (tablas 4.2, 4.3 y 4.4) corresponden al análisis utilizando tf para el

pesado de términos, y las tres siguientes (tablas 4.5, 4.6 y 4.7) corresponden al análisis aplicando la técnica de pesado tf-idf. Cada tabla resume los resultados obtenidos entre un par de lenguajes siendo C++−Java, Python→C++ y Python→Java el orden de las tablas tanto en el conjunto tf y tf-idf. Los resultados se muestran en términos de la media y desviación típica para un valor $n$ respecto a los distintos tipos de modificaciones comentados en el párrafo anterior.

| Experimentos/$n$-gramas | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| full code | 2,78 ± 1,31 | 1,67 ± 1,33 | **1,44 ± 0,83** | 1,78 ± 1,13 | 1,78 ± 1,13 |
| fc-without comments | 2,67 ± 1,33 | 1,67 ± 1,33 | **1,44 ± 0,83** | 1,78 ± 1,34 | 1,78 ± 1,34 |
| only comments | 3,17 ± 1,06 | 1,50 ± 1,11 | 2,83 ± 1,34 | 2,50 ± 1,25 | 3,17 ± 1,21 |
| fc-only reserved words | 2,33 ± 1,24 | 2,22 ± 1,13 | 1,78 ± 1,02 | 1,67 ± 1,05 | 2,00 ± 0,94 |
| fc-without rw | 3,11 ± 1,19 | 2,11 ± 1,44 | 1,78 ± 1,13 | 1,67 ± 1,05 | 1,67 ± 1,05 |
| fc-wc-wrw | 2,89 ± 0,69 | 2,11 ± 1,44 | 1,67 ± 0,94 | 1,78 ± 1,06 | 1,89 ± 1,37 |

Tabla 4.3: Resultados obtenidos comparando los documentos del corpus SPADE escritos en Python y en C++ a nivel de documento, dónde cada documento es pesado utilizando tf. La tabla refleja la media y desviación típica de las posiciones de los documentos considerados reutilizados respecto del documento que está siendo analizado.

En la tabla 4.2 se reflejan los resultados para las distintas partes del código fuente que se han considerado en los experimentos para el par de lenguajes C++−Java. Cada experimento se ha repetido cambiando el valor de $n$ en el rango $[1, \ldots, 5]$. En la mayoría de experimentos se refleja como a partir de trigramas se encuentran los mejores resultados, y en varios casos; considerando solamente los comentarios o sólo las palabras reservadas del lenguaje; empeoran al subir el valor a tetragramas o 5-gramas. Por otra parte, los mejores resultados se han obtenido considerando el código fuente entero y considerándolo sin los comentarios. Desde bigramas hasta 5-gramas la aproximación sitúa siempre a su par reutilizado como el más sospechoso entre los escritos en el otro lenguaje.

Los resultados del estudio de reutilización en el corpus SPADE a nivel de documento entre los lenguajes Python→C++ se muestra en la tabla 4.3. En comparación con el par C++−Java, los resultados son ligeramente superiores. Esto es debido a que entre el

par C++−Java gran parte de la sintaxis de ambos lenguajes es la misma, y el resto es muy similar. El hecho de tener sintaxis tan cercanas permite reutilizaciones de código más parecidas. En el caso Python→C++, existe una necesidad de adaptar más el código. Este proceso de adaptación hace que exista menos similitud y los resultados no sean tan buenos como en el otro caso. Como ocurre en el caso C++−Java, considerando el código fuente entero y considerando el código sin los comentarios, esta vez sólo con trigramas.

| Experimentos/$n$-gramas | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| full code | 2,62 ± 1,10 | 1,75 ± 0,96 | 1,62 ± 1,10 | 1,62 ± 1,10 | 1,62 ± 1,10 |
| fc-without comments | 2,62 ± 1,10 | 1,75 ± 0,96 | 1,62 ± 1,10 | 1,62 ± 1,10 | 1,62 ± 1,10 |
| only comments | 2,33 ± 1,56 | 2,33 ± 1,56 | 3,00 ± 0,67 | 2,33 ± 1,56 | 3,33 ± 0,89 |
| fc-only reserved words | 2,50 ± 0,86 | 1,88 ± 1,05 | 1,75 ± 0,83 | 1,75 ± 1,09 | 1,75 ± 1,09 |
| fc-without rw | 2,50 ± 1,00 | 1,62 ± 1,10 | 2,00 ± 1,32 | 1,88 ± 1,16 | 1,75 ± 1,09 |
| fc-wc-wrw | 2,50 ± 1,50 | 1,67 ± 1,78 | **1,44 ± 0,69** | 1,78 ± 1,28 | 1,78 ± 1,28 |

Tabla 4.4: Resultados obtenidos comparando los documentos del corpus SPADE escritos en Python y en Java a nivel de documento, dónde cada documento es pesado utilizando tf. La tabla refleja la media y desviación típica de las posiciones de los documentos considerados reutilizados respecto del documento que está siendo analizado.

Entre el par de lenguajes Python→Java, el mejor resultado obtenido ha sido considerando el código fuente sin comentarios y sin palabras reservadas como se muestra en la tabla 4.4. Esto se debe a que el proceso de reutilización entre ambos lenguajes ha requerido utilizar librerías diferentes de la estándar y al eliminar las palabras pertenecientes al lenguaje y librerías han sido eliminadas. Esto ha causado que la similitud estimada sea de la información que aporta el programador como nombres de identificadores, mensajes en cadenas, etc. Sin embargo, al igual que en los otros pares de lenguajes, el código fuente entero y el código fuente sin comentarios también han generado valores buenos, siendo bastante similares a los mejores.

Como se ha comentado al inicio del experimento, también se ha aplicado el método de pesado tf-idf sobre los tres pares de lenguajes de programación. En la tabla 4.5 se muestran los resultados obtenidos al utilizar el método tf-idf sobre el par de lenguajes C++−Java y

considerando los mismos experimentos que para tf. Como en el caso de tf entre este par de lenguajes, con considerando el código fuente entero y considerando el código sin los comentarios, sitúa siempre a su par reutilizado como el más sospechoso entre los escritos en el otro lenguaje. El hecho que comparten sintaxis, facilita al programador la reutilización del código, siendo ésta más similar y por lo tanto más fácil de detectar por aproximaciones como la del trabajo actual.

| Experimentos/$n$-gramas | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| full code | 2,56 ± 0,69 | **1,00 ± 0,00** | **1,00 ± 0,00** | **1,00 ± 0,00** | **1,00 ± 0,00** |
| fc-without comments | 2,56 ± 0,69 | **1,00 ± 0,00** | **1,00 ± 0,00** | **1,00 ± 0,00** | **1,00 ± 0,00** |
| only comments | 2,43 ± 0,82 | 2,29 ± 1,92 | 2,14 ± 2,41 | 2,43 ± 2,24 | 2,57 ± 2,24 |
| fc-only reserved words | 2,56 ± 0,69 | 1,56 ± 0,47 | 1,56 ± 0,25 | 1,22 ± 0,17 | 1,44 ± 0,47 |
| fc-without rw | 2,56 ± 0,69 | 2,11 ± 2,54 | 2,89 ± 1,88 | 3,11 ± 2,32 | 1,89 ± 0,99 |
| fc-wc-wrw | 2,56 ± 0,69 | 2,11 ± 2,54 | 2,11 ± 2,54 | 2,11 ± 2,54 | 2,11 ± 2,54 |

Tabla 4.5: Resultados obtenidos comparando los documentos del corpus SPADE escritos en C++ y en Java a nivel de documento, dónde cada documento es pesado utilizando tf-idf. La tabla refleja la media y desviación típica de las posiciones de los documentos considerados reutilizados respecto del documento que está siendo analizado.

También entre el par de lenguajes Python→C++, el mejor resultado se obtiene utilizando el código fuente entero y el código fuente sin los comentarios como se muestra en la tabla 4.6. Se ha estudiado manualmente porqué se obtiene valores similares utilizando el código fuente con y sin los comentarios. Se ha apreciado que la cantidad de comentarios es de un 2 % del total del texto, y en muchos casos el propio programador añade sus propios comentarios después de la reutilización de código. La baja cantidad de comentarios hace que éstos no influyan en el resultado de la detección. A diferencia de utilizar tf entre este par de lenguajes, utilizando idf se obtienen los mejores resultados además de con trigramas con 4-gramas y 5-gramas.

El estudio realizado entre el par de lenguajes Python→Java utilizando como método de pesado tf-idf se muestra en la tabla 4.7. El

| Experimentos/$n$-gramas | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| full code | 2,56 ± 0,69 | 1,56 ± 1,14 | **1,44 ± 0,69** | **1,44 ± 0,69** | **1,44 ± 0,69** |
| fc-without comments | 2,56 ± 0,69 | 1,78 ± 1,28 | **1,44 ± 0,69** | **1,44 ± 0,69** | **1,44 ± 0,69** |
| only comments | 2,50 ± 0,92 | 2,83 ± 2,47 | 2,83 ± 2,14 | 3,17 ± 1,81 | 3,00 ± 1,67 |
| fc-only reserved words | 2,56 ± 0,69 | 2,89 ± 1,65 | 2,56 ± 1,80 | 2,89 ± 1,88 | 2,67 ± 1,78 |
| fc-without rw | 2,56 ± 0,69 | 2,78 ± 0,84 | 2,67 ± 2,00 | 3,67 ± 1,11 | 1,89 ± 0,99 |
| fc-wc-wrw | 2,56 ± 0,69 | 2,33 ± 2,44 | 2,89 ± 1,88 | 3,56 ± 1,58 | 2,00 ± 1,11 |

Tabla 4.6: Resultados obtenidos comparando los documentos del corpus SPADE escritos en Python y en C++ a nivel de documento, dónde cada documento es pesado utilizando tf-idf. La tabla refleja la media y desviación típica de las posiciones de los documentos considerados reutilizados respecto del documento que está siendo analizado.

mejor resultado obtenido ha sido utilizando 5-gramas considerando el código fuente entero y el código sin los comentarios. A pesar de que la posición media en el ranking para trigramas y 4-gramas es ligeramente superior, no tienen diferencias significativas con el mejor resultado. Sin embargo, comparando los resultados con los obtenidos con tf, son ligeramente peores. Ninguno de los tres pares de lenguajes ha mostrado que funcione mejor utilizando tf-idf. Esto ocurre porque tf-idf necesita un conjunto más grande que los disponibles en este trabajo de investigación para determinar la relevancia de cada $n$-grama.

| Experimentos/$n$-gramas | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| full code | 2,50 ± 1,50 | 1,62 ± 0,98 | 1,62 ± 1,23 | 1,62 ± 1,23 | **1,50 ± 0,75** |
| fc-without comments | 2,62 ± 1,23 | 1,62 ± 0,98 | 1,62 ± 1,23 | 1,62 ± 1,23 | **1,50 ± 0,75** |
| only comments | 2,00 ± 2,00 | 2,33 ± 1,56 | 3,00 ± 0,67 | 2,33 ± 1,56 | 4,00 ± 0,00 |
| fc-only reserved words | 2,50 ± 1,50 | 2,62 ± 0,98 | 2,62 ± 0,98 | 2,88 ± 1,11 | 2,88 ± 1,11 |
| fc-without rw | 2,50 ± 1,50 | 2,00 ± 1,50 | 2,50 ± 1,25 | 1,75 ± 0,19 | 3,50 ± 0,75 |
| fc-wc-wrw | 2,50 ± 1,50 | 2,00 ± 1,50 | 2,50 ± 1,25 | 1,75 ± 0,19 | 4,00 ± 0,00 |

Tabla 4.7: Resultados obtenidos comparando los documentos del corpus SPADE escritos en Python y en Java a nivel de documento, dónde cada documento es pesado utilizando tf-idf. La tabla refleja la media y desviación típica de las posiciones de los documentos considerados reutilizados respecto del documento que está siendo analizado.

Finalmente, tras analizar individualmente todos los resultados, realizamos un análisis en conjunto. Respecto al parámetro $n$ que determina el tamaño del $n$-grama, se ha apreciado una tendencia

común entre todos los pares de lenguajes. A partir del valor $n = 3$, los resultados ya no mejoran y se estabilizan siendo los mejores resultados. Este mismo comportamiento se puede observar en la detección de reutilización en textos [25]. Por otra parte, como se puede apreciar en la tabla 4.8, los mejores resultados se obtienen utilizando el código entero o utilizando el código sin los comentarios. Se debe al pequeño impacto de los comentarios que no son más del 2 % del total del código fuente. Eliminar los comentarios supone incluir un preproceso dependiente de los lenguajes de programación. Por estas razones, se optará por considerar el código fuente entero sin eliminar los comentarios, ni palabras reservadas, ni identificadores. Así conseguimos tener una aproximación translingüe a nivel de documento sin necesidad de adaptarse al lenguaje, es decir, con independencia de los lenguajes de programación con los que estén escritos los códigos a comparar. También se ha llegado a la conclusión que con los corpus que disponemos actualmente no se puede aplicar tf debido a que es necesario un corpus mucho más grande para identificar las relevancias de los $n$-gramas.

| Experimentos | C++−Java | Python→C++ | Python→Java |
|---|---|---|---|
| full code | 1,00 ± 0,00 | 1,44 ± 0,83 | 1,62 ± 1,10 |
| fc-without comments | 1,00 ± 0,00 | 1,44 ± 0,83 | 1,62 ± 1,10 |
| comments only | 2,29 ± 1,57 | 2,83 ± 1,34 | 3,00 ± 0,67 |
| fc-reserved words only | 1,56 ± 0,83 | 1,78 ± 1,02 | 1,75 ± 0,83 |
| fc-without rw | 1,44 ± 0,83 | 1,78 ± 1,13 | 2,00 ± 1,32 |
| fc-wc-wrw | 1,44 ± 0,83 | 1,67 ± 0,94 | 1,44 ± 0,69 |

Tabla 4.8: Resultados obtenidos con trigramas de caracteres. Cada valor representa la media y la desviación típica de la posición del código fuente en el raking.

En conclusión, se ha determinado tras los resultados que para detectar reutilización a nivel de documento, la mejor solución con esta aproximación es: (*i*) Utilizar trigramas de caracteres para segmentar el código fuente porque es más efectivo, como ocurre en textos escritos; (*ii*) utilizar un método de pesado tf porque con los corpus actuales tf-idf no aporta una mejora sustancial que compense su

mayor coste computacional; y (*iii*) considerar el código fuente entero sin eliminar comentarios, ni palabras reservadas del lenguaje ni identificadores porque, en general, para los tres pares de lenguajes ha obtenido buenos resultados.

### 4.3.2. Estimación de los parámetros del modelo a nivel de fragmento

El objetivo de este experimento se centra en encontrar la combinación de valores de los parámetros que permiten detectar mejor la reutilización de código fuente translingüe a nivel de fragmento. Con el análisis a nivel de fragmento se pretende evitar que si la reutilización ha sido de una parte pequeña y que el resto del código reduzca la similitud global del documento. Para ello hemos utilizado el corpus SPADE. Los rangos de valores considerados para ajustar los tres parámetros son los siguientes:

- tamaño de la ventana: $s$={10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300}
- desplazamiento de la ventana: $d$={10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300}
- umbral: $t$=$\{0,1,\ 0,2,\ \ldots,\ 0,9\}$

Se ha elegido el valor 300 como máximo valor del tamaño y desplazamiento de la ventana por ser éste el tamaño del código fuente más pequeño tras el preproceso. En todos los cálculos realizados siempre se ha elegido un desplazamiento de la ventana menor o igual al tamaño de la ventana porque eligiendo un desplazamiento mayor, no se compararía todo el código. Eligiendo un desplazamiento menor que el tamaño de la ventana se produce el fenómeno de solapamiento donde cada ventana comparte información de la anterior.

En la figura 4.2 se muestra un ejemplo de la variación del desplazamiento ($d$) manteniendo constante el tamaño de la ventana ($l$) para varios valores del umbral ($t$) entre códigos correspondientes al par de lenguajes Python→Java. Independientemente del valor del parámetro $t$, se comprueba que durante el solapamiento el valor de

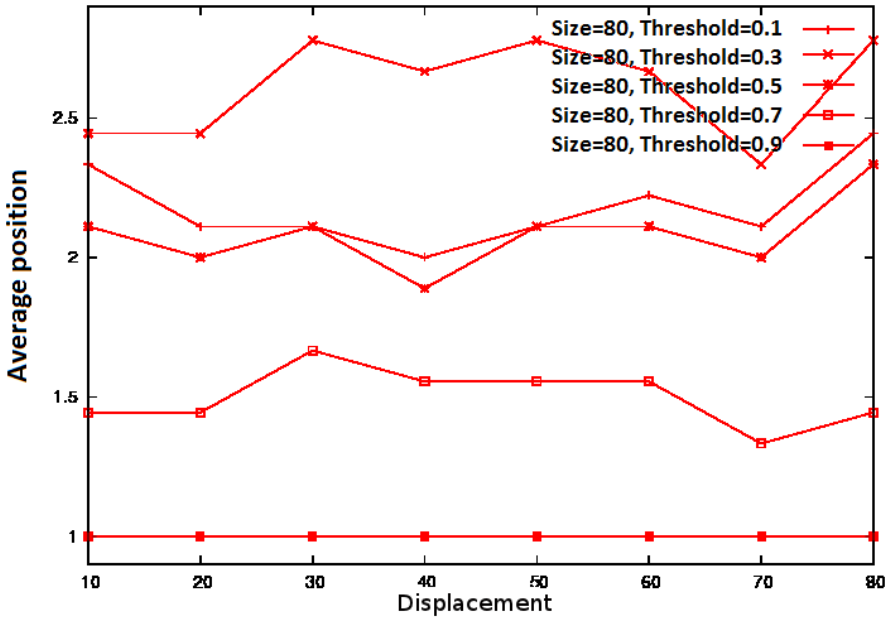


Figura 4.2: Resultados del modelo a nivel de fragmento sobre el corpus SPADE muestran que al variar el solapamiento no se produce ninguna mejora relevante. El resultado para cada umbral y el mismo tamaño de ventana variando el desplazamiento se mantiene estable.

la posición media en el ranking se mantiene alrededor de un valor sin obtener mejoras significativas para todos los casos. Es por ello que se desestima el solapamiento de ventanas en el proceso de cálculo de similitud. Esto comportará una reducción notable del número de comparaciones a realizar porque se reducirá el número de ventanas de cada código fuente que se vaya a comparar. Para los tres pares de lenguajes del corpus se ha observado la misma tendencia.

Una vez establecido que el valor del desplazamiento que se va a utilizar va a ser el mismo que el valor del tamaño de la ventana, se procede a delimitar el valor del umbral entre los pares de lenguajes. Se ha comprobado que entre lenguajes que tienen una sintaxis muy próxima como el caso de C++ y Java, e incluso comparten gran parte de vocabulario, tienen una similitud alta entre ventanas que no son reutilizadas. Es por ello que se necesita un umbral elevado para discriminar las ventanas reutilizadas de las que no como se muestra en la figura 4.3.

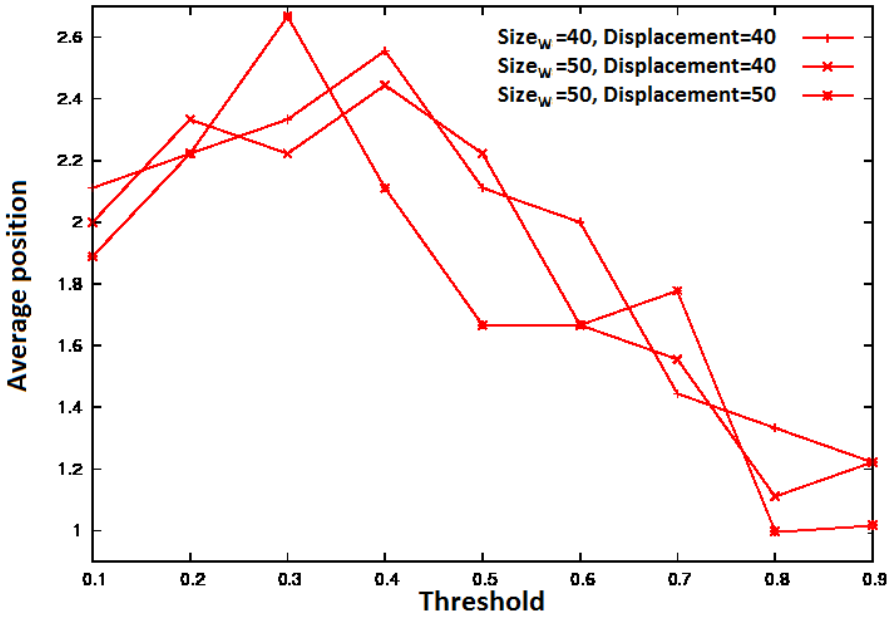


Figura 4.3: Resultados del modelo a nivel de fragmento sobre el corpus SPADE muestran que incrementar el umbral se consiguen mejores resultados. Para diferentes configuraciones de $s$ y $d$, en torno al valor $0,8$ de umbral se consiguen los mejores resultados entre C++ y Java.

El total de experimentos que se puede llevar a cabo con la combinación de los 3 parámetros es de 1.764: 14 valores de desplazamientos ($l$) * 9 valores de umbrales ($t$) * 14 valores de tamaño de ventana ($s$). Como sólo se han tenido en cuenta las combinaciones que generan solapamiento y las combinaciones que el desplazamiento es igual al tamaño de la ventana, en total 945 experimentos se han realizado en este estudio. En la tabla 4.9 se muestra la mejor configuración obtenida combinando distintos valores para los tres parámetros para los tres pares de lenguajes. Basándose en que más de una combinación de parámetros obtenía el mismo valor de posición media en el ranking, se ha optado por: ($i$) descartar aquellas opciones que requerían de solapamiento por el motivo que no mejora los resultados y aumenta el coste computacional; ($ii$) descartar las opciones que tuvieran el umbral más restrictivo, permitiendo ser más flexible manteniendo la calidad del sistema y ($iii$) elegir el menor tamaño de ventana posible para considerarlo como el tamaño mínimo que

se puede reutilizar del código fuente. En el caso de que una ventana tuviese un tamaño mayor que el del fragmento reutilizado, se estaría utilizando para la comparación parte no reutilizada por lo que disminuiría el valor de similitud entre dos ventanas.

| Pares de lenguajes | Tam. vent. ($s$) | Despl. ($d$) | Umbr.($t$) | Pos. med. |
|---|---|---|---|---|
| C++−Java | 50 | 50 | 0,8 | 1,000 ± 0,00 |
| Python→C++ | 50 | 50 | 0,1 | 1,375 ± 0,69 |
| Python→Java | 30 | 30 | 0,2 | 1,444 ± 0,71 |

Tabla 4.9: Resumen de la mejor combinación de parámetros que permite obtener los mejores resultados sobre el corpus SPADE.

Las mejor combinación de parámetros entre el par C++−Java corresponde a un tamaño de 50 caracteres para la ventana y desplazamiento y con un umbral de $0,8$. Es necesario un umbral tan alto debido a que ambos lenguajes comparten sintaxis y esto provoca que la similitud entre ventanas que no han sido reutilizadas sea más elevada que entre dos lenguajes que no comparten sintaxis. Respecto a los otros dos pares de lenguajes (Python→C++ y Python→Java) se ha calculado como tamaño de ventana y desplazamiento unos valores de 50 y 30 caracteres respectivamente. Como se ha comentado anteriormente, entre los lenguajes que no tienen sintaxis similar es necesario utilizar un umbral menor ($0,1$ y $0,2$ respectivamente). En el caso entre Python→Java, al utilizar una ventana menor, hay menos variedad de caracteres por lo tanto se necesita un umbral más alto que con Python→C++. En general los resultados obtenidos utilizando el modelo a nivel de fragmento son ligeramente mejores que los obtenidos a nivel de documento. Se espera que el modelo a nivel de fragmento se comporte mejor en escenarios donde la reutilización ha sido sólo una parte, y no como es el caso del corpus SPADE donde todo el código fuente ha sido reutilizado.

### 4.3.3. Reutilización monolingüe sobre corpus CL-AT++

Para comprobar la efectividad de estos parámetros sobre un corpus más grande, se ha realizado otros experimentos con el corpus CL-AT++ que contiene casos reales de reutilización entre alumnos como se indica en [1]. El primer experimento se ha realizado a nivel monolingüe, sobre código escrito en el lenguaje C. Se ha comparado el sistema a nivel de documento, con el sistema a nivel de fragmento y la herramienta JPlag. Dado que la cantidad de códigos fuente en el corpus ya es un número considerable, esta comparación se puede realizar con métricas estándar como son la precisión y la cobertura. Se ha detectado que los parámetros estimados con el corpus SPADE entre los lenguajes C++−Java, no funcionan tan bien a nivel monolingüe sobre el corpus CL-AT++ en el lenguaje C, por lo que se ha reestimado estos parámetros.

Para estimar los parámetros se han probado todos los valores de umbral $t$=[0, 1, ...,0, 9] y con los siguientes valores de tamaño de ventana y desplazamiento $s$=$d$={50, 100, 150, 200, 250, 300, 350, 400}, es decir, sin solapamiento y tomando el valor máximo de 400 caracteres por ser éste el tamaño del código fuente más pequeño en este corpus. La mejor combinación de parámetros encontrada para el lenguaje Java a nivel monolingüe ha sido $s$=$d$=300 y $t$=0, 9. A diferencia del corpus SPADE, en el corpus CL-AT++ se han apreciado modificaciones como cambios de nombres de identificadores o cambios en el orden de la declaración de las variables. Para detectar fragmentos con cambios para evitar ser detectados será necesario utilizar un tamaño de ventana mayor. Por otra parte, el umbral tiene un valor alto como era de esperar porque la similitud aumenta cuando se trata de códigos escritos en lenguajes de programación con sintaxis parecidas.

En la figura 4.4 se muestra la comparación entre el sistema a nivel de documento utilizando tf y trigramas, el sistema a nivel de fragmento con los parámetros estimados y el sistema JPlag. Mientras que JPlag mantiene la precisión con valor 1, 00 hasta el 0, 05 de la cobertura, el sistema basado en fragmento y el basado en documento lo mantiene hasta 0, 24 y 0, 33 respectivamente. Con el sistema a

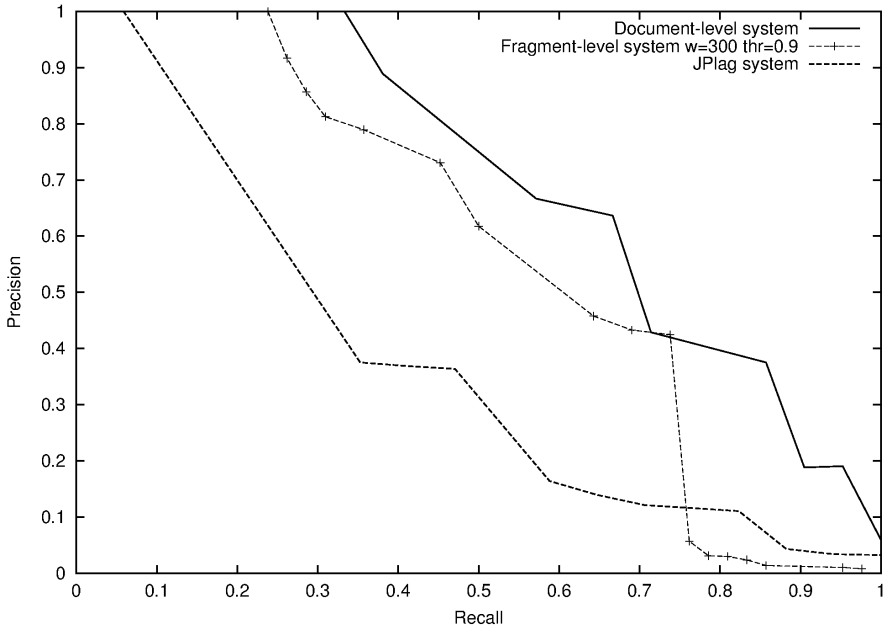


Figura 4.4: Resultados de precisión frente a cobertura comparando el modelo a nivel de documento, a nivel de fragmento y JPlaga nivel monolingüe con el corpus CL-AT++. El modelo a nivel de documento mantiene la curva por encima de el resto de modelos.

nivel de documento se obtienen mayor precisión que con los otros dos. El sistema propuesto basando en el modelo de fragmento es notablemente más preciso que JPlag hasta alcanzar una cobertura de $0,75$. Las dos aproximaciones de este trabajo de investigación presentan mejores prestaciones que una herramienta contrastada a nivel monolingüe. Comparando las dos aproximaciones con un caso real de reutilización como es CL-AT++, la aproximación a nivel de documento ha resultado obtener mejores resultados. Sin embargo, si interesa localizar los fragmentos de código que han sido reutilizados, es más sencillo adaptar el modelo a nivel de fragmento.

### 4.3.4. Reutilización translingüe sobre corpus CL-AT++

Se trata un experimento translingüe con el corpus CL-AT++. Para ello se han comparado los dos sistemas propuestos con los mismos parámetros (trigramas y tf para nivel de documento, y $s$=$d$=300 con

$t$=0, 9 para nivel de fragmento). Además, se comparará con el sistema JPlag del estudio anterior a nivel monolingüe para comparar los resultados con un sistema contrastado trabajando en una tarea más sencilla como es el caso monolingüe frente al caso translingüe.

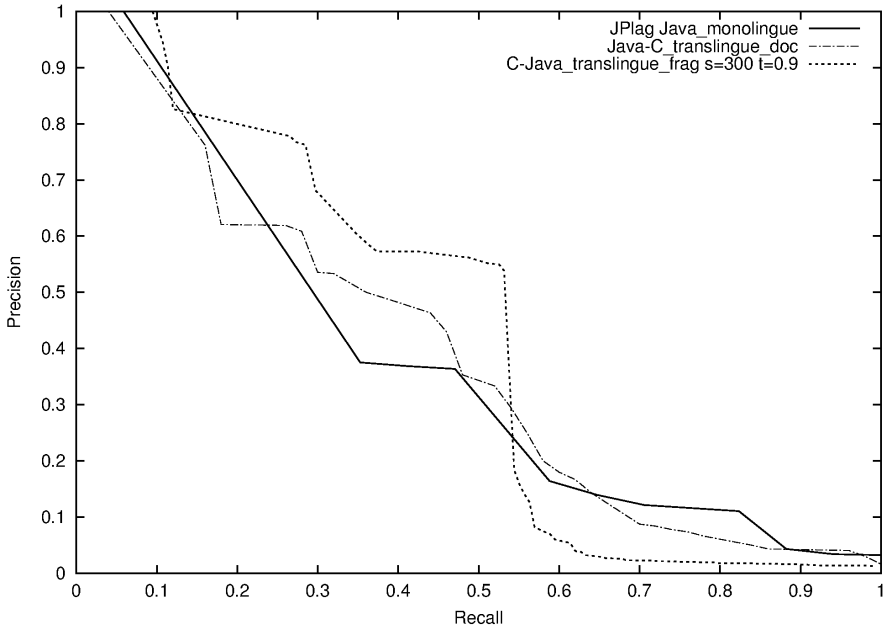


Figura 4.5: Resultados de modelos a nivel de documento y fragmento a nivel translingüe comparados con JPlag a nivel monolingüe con el corpus CL-AT++. Incluso a nivel translingüe se muestran ligeramente superiores.

En la figura 4.5 se muestra como los tres sistemas, tanto los dos translingüe de este trabajo de investigación como JPlag con un problema monolingüe mantienen la precisión con valor 1 para valores de recall similares, por debajo de 0, 1. Tanto el modelo a nivel de fragmento como el modelo a nivel de documento han visto reducidas sus prestaciones al trabajar en un contexto más complejo como es el translingüe. Como ocurría con el corpus SPADE, las prestaciones del modelo a nivel de fragmento en el contexto translingüe también son ligeramente mejores que las del modelo a nivel de documento. Además, las dos aproximaciones de este trabajo de investigación demuestran tener prestaciones similares o superiores al de la herramienta JPlag, incluso trabajando en un contexto con más complejidad que este último.

# Capítulo 5

# Conclusiones y Trabajos Futuros

## 5.1. Conclusiones

La detección automática de reutilización de código fuente en un contexto translingüe es un campo muy poco explorado, sólo se ha encontrado un trabajo de investigación que aborde el problema en código fuente a nivel translingüe [1]. Su complejidad aumenta al no disponer de suficientes recursos ni herramientas con los que contrastar los resultados obtenidos[1].

Este trabajo aborda la detección de reutilización de código fuente entre diferentes lenguajes de programación. Las dos aproximaciones propuestas, comparación a nivel de documento y comparación a nivel de fragmento, se basan en la medición de similitud a través de la división en $n$-gramas a nivel de carácter. Se ha analizado el impacto que generan los comentarios, nombres de variables y las palabras reservadas de los diferentes lenguajes de programación sobre el cálculo de similitud entre un par de códigos. Las estrategias de ocultación más comunes son: renombrar variables y funciones, modificar el orden de partes del código no dependientes como pueden ser las funciones, *cases* en un *switch*, etc. tal como sugieren Faidhi y Robinson [13].

[1]Se contactó con los autores del estudio a nivel translingüe y no disponían completamente del corpus utilizado, habiéndose facilitado parte del corpus.

Con la aproximación a nivel de documento, los mejores resultados se han obtenido utilizando el código fuente entero, o bien el código sin los comentarios. Analizando el corpus, se ha observado que la existencia de comentarios es muy baja y en su mayoría el programador que ha reutilizado código ha desestimado los comentarios, o bien los ha reescrito. Como la cantidad de comentarios en un código fuente es una cantidad reducida del código fuente, un alumno no insertará comentarios si pretende evadir la detección automática de código fuente reutilizado. El código fuente que implemente un alumno se pretende que sea un código usable y entendible para el profesor, por lo que se considera muy poco probable el caso de introducir ruido a través de comentarios. Por otra parte, incluir un preprocesado a las aproximaciones para eliminar los comentarios supondría tener un sistema dependiente del los lenguaje de programación. Por estas razones, se considera que la mejor opción es utilizar el código fuente entero.

En la estimación del valor $n$ de los $n$-gramas de caracteres los mejores resultados se han obtenido utilizando un valor de $n$=3. Los trigramas de caracteres son capaces de representar el estilo de programación como sucede en los textos escritos en lenguaje natural como se discutió en [25]. Entre los métodos tf y tf-idf para el pesado de trigramas no se ha encontrado una gran diferencia. Este hecho es debido a que idf sería relevante si dispusiéramos de un corpus mucho más grande que los disponibles actualmente. Se experimentará con tf-idf cuando se disponga de una cantidad elevada de códigos fuente del cual se pueda extraer un conjunto de $n$-gramas que represente al conjunto posible de $n$-gramas que se puede utilizar en los lenguajes de programación que se desee analizar. Así pues, en la situación actual, dado que tf-idf no supone una mejora respecto a tf se considera utilizar el último dado que no es necesario ningún preproceso y su cálculo es menos costoso computacionalmente.

Para el cálculo de la similitud a nivel de fragmento, se han utilizado los mismos términos usados a nivel de documento, es decir, trigramas y el método de pesado tf. El resto de parámetros que son propios de este sistema - como son el tamaño de la ventana $s$, el desplazamiento de la ventana $d$ y el umbral de confianza ($t$) - se han ob-

tenido estimando la mejor combinación entre todos ellos. Utilizando el corpus SPADE se ha detectado que los lenguajes de programación que comparten sintaxis como las propias palabras reservadas del lenguaje necesitan un umbral con un valor elevado $(0,8)$, a diferencia de los lenguajes que no los comparten donde el umbral es relativamente bajo $(0,1-0,2)$. Esto se debe a que los lenguajes que comparten sintaxis obtienen una similitud mayor respecto a los que no la comparten. Por otra parte, se ha observado que el solapamiento entre ventanas no mejora los resultados. Por ello, se opta por desestimar el solapamiento cuando se calcula la similitud con el método de la ventana deslizante. Es decir, el tamaño de la ventana $s$ será igual al desplazamiento $l$. Además, se ha comprobado que se identifican mejor las partes del código reutilizadas cuando el tamaño de la ventana es bajo, por lo que se ha elegido el menor valor de $s$ entre los que generan los mejores resultados.

Con el corpus CL-AT++, que dispone de más casos de reutilización y más reales, se ha comprobado que los valores de los parámetros $s$ y $d$ estimados con el corpus SPADE no funcionan correctamente. Esto se debe a que la reutilización en corpus SPADE no consta de modificaciones para evitar la detección, sólo son traducciones del código fuente. Sin embargo, en el corpus CL-AT++ los casos de reutilización consisten en modificaciones para dificultar la detección. Por este motivo, el tamaño de la ventana y el desplazamiento de la ventana se incrementan para poder detectar estos casos correctamente. En concreto para la parte del corpus escrita en lenguaje C a nivel monolingüe se utiliza $s$=$d$=300 y $t$=$0,9$.

En el caso monolingüe el sistema que compara a nivel de documento funciona ligeramente mejor que el sistema a nivel de fragmento. Ambos sistemas se muestran más efectivos que la herramienta JPlag. Finalmente, se han comparado con JPlag los resultados de evaluación de las dos aproximaciones propuestas sobre el corpus multilingüe CL-AT++ en ambos contextos, monolingüe y translingüe. En el caso de translingüe, del mismo modo que ocurre con el corpus SPADE, resulta más eficaz el modelo a nivel de fragmento que el modelo a nivel de documento, del mismo modo que ocurría con el corpus SPADE. Concluyendo, tanto en el caso monolingüe como en

el translingüe, ambas aproximaciones se comportan igual o mejor que el sistema contrastado JPlag que funciona sólo en un entorno monolingüe.

## 5.2. Líneas de investigación abiertas

Aunque los resultados preliminares obtenidos en la detección de reutilización de código fuente entre lenguajes de programación son satisfactorios, éste es un problema que está en sus inicios, a diferencia de la reutilización de código fuente a nivel monolingüe donde ya existen sistemas disponibles para ámbitos académicos como JPlag [26].Existen multitud de opciones por explorar antes de poder considerarlo un problema resuelto. Algunas líneas interesantes en futuras investigaciones:

1. Obtener recursos más amplios con códigos correspondientes a nuevas tareas y nuevos lenguajes de programación.
2. Centrar el estudio a nivel de fragmento con más casos reales de fragmentos reutilizados con el objeto de localizar las partes reutilizadas del código fuente.
3. Explorar nuevos modelos utilizados sobre textos para adaptarlos a códigos fuente.
4. Desarrollar un sistema combinando distintas.
5. Desarrollar una plataforma Web para diseminación y uso del sistema a nivel académico y comercial.

Actualmente se ha comenzado a trabajar en varias de las líneas propuestas. La dificultad de las tareas anteriores, incluso de manera individual, obliga a planificar su investigación dentro de la continuación de esta investigación: en la etapa de doctorado.

### 5.2.1. Creación de nuevos recursos

Como se ha comentado a lo largo de la tesis, el tamaño del corpus no es el ideal para poder extraer conclusiones definitivas. Deberemos ampliar el corpus con nuevas muestras que nos permitan reforzar la validez de los parámetros y técnicas utilizadas en este estudio contrastando los resultados con los obtenidos en este trabajo de investigación.

El objetivo reside en disponer de códigos fuente de distintos concursos (*Southwestern Europe Regional Contest* o *Google Code Jam*) de programación donde se ha detectado manualmente casos de reutilización de código, tanto parcial como completa[2]. Actualmente, se está recopilando nuevos códigos desarrollados por los alumnos de la asignatura de Programación y de Ingeniería del Lenguaje Natural dónde existen indicios de casos de reutilización detectados manualmente.

La idea es generar un corpus multilingüe con códigos fuente de cualquier ámbito a partir de repositorios. Este corpus deberá contemplar el máximo número de lenguajes de programación. Existen repositorios que contienen grandes cantidades de problemas o algoritmos implementados en distintos lenguajes de programación[3].

### 5.2.2. Detección de fragmentos

En el corpus SPADE todos los casos son de reutilización del código entero, mientras que en el corpus CL-AT++ la mayoría de casos tienen reutilización parcial y se han añadido partes de código propias. Es por ello, que al obtener resultados satisfactorios con el estudio a nivel de fragmento, se considera interesante adaptarlo para detectar y señalar que fragmentos de un código fuente que han sido reutilizados por otros.

[2] `http://swerc.eu/` o `http://code.google.com/codejam/`
[3] `http://rosettacode.org/wiki/Rosetta_Code`

### 5.2.3. Explorar nuevas técnicas

Este trabajo de investigación se considera un primer acercamiento a la detección de reutilización translingüe en código fuente utilizando técnicas de PLN. Como primer paso, se ha utilizado el sistema basado en $n$-gramas debido a que es el menos restrictivo de los utilizados en otros trabajos dado que no requiere de ningún tipo de corpus inicial con el que entrenar.

Una de las técnicas del lenguaje natural que es interesante aplicar en código fuente es el análisis semántico explícito (ESA, por su acrónimo en inglés) propuesto por Gabrilovich y Markovitch [17]. ESA consiste en comparar un documento con un corpus monolingüe. Estas comparaciones generan un valor de similitud por cada documento del corpus. Si dos documentos comparados con el mismo corpus generan unos valores de similitud cercanos, ambos documentos se considerarán similares. Dado que el corpus es fijo, entonces ambos vectores son de la misma longitud. La técnica anterior se ha extendido con un modelo translingüe (CL-ESA) en el trabajo de Potthast et al. [25]. CL-ESA utiliza un corpus multilingüe comparable (con documentos similares en distintos idiomas). El uso de corpus comparables permite comparar dos documentos escritos en distintos lenguajes con la parte del corpus escrita en su idioma y que es similar a la de los otros idiomas. Wikipedia[4] es un recurso muy utilizado para este tipo de tareas.

También se tratará de abordar el problema translingüe desde la perspectiva de la traducción automática. Para ello se utiliza la técnica CL-ASA propuesta por Barrón-Cedeño et al. [6]. CL-ASA [24] consiste en detectar reutilización de textos con un método probabilístico que calcula un diccionario estadístico bilingüe basado en el modelo de alineamiento IBM-1. CL-ASA calcula la probabilidad de asociación probabilística entre dos términos en dos idiomas diferentes. Se pretende utilizar este método utilizando un corpus paralelo (traducciones de código fuente entre lenguajes) para estimar los diccionarios bilingües.

[4] http://www.wikipedia.org/

### 5.2.4. Combinación de técnicas

Una vez probadas todas las técnicas descritas anteriormente, la idea es combinarlas con las utilizadas a nivel monolingüe descritas en el capítulo 1.2 basadas en la estructura del código. El objetivo principal es analizar las bondades y debilidades de los diferentes sistemas ya sean basados en tecnologías de lenguaje natural como basados en la estructura del código. Con ello se identificará el sistema más idóneo a aplicar en cada caso. Es decir, la finalidad será diseñar un sistema de detección de reutilización mixto que sea viable.

### 5.2.5. Plataforma Web

La idea consiste en proporcionar una plataforma independiente del tipo de máquina, del lenguaje de programación, capaz de detectar reutilización entre colecciones de códigos fuente escritos en diferentes lenguajes. Esta plataforma proporcionará aquellos conjuntos que tengan una similitud elevada y por lo tanto sean sospechosos de ser casos de reutilización. También deberá ser capaz de mostrar a nivel de código fuente los fragmentos que han sido reutilizados entre los pares de códigos. Se ha desarrollado un prototipo de esta segunda parte[5]. El propósito de esta plataforma es que sea utilizada tanto en el ámbito académico como en el comercial.

[5] `http://memex2.dsic.upv.es:8080/DeSoCoRe/`

# Bibliografía


[1] C. Arwin and S. Tahaghoghi. Plagiarism detection across programming languages. *Proceedings of the 29th Australian Computer Science Conference, Australian Computer Society*, 48:277–286, 2006.

[2] A. Barrón-Cedeño, A. Eiselt, and P. Rosso. Monolingual text similarity measures: A comparison of models over wikipedia articles revisions. *Proceedings of 7th International Conference on Natural Language Processing, ICON-2009, Hyderabad, India*, pages 29–38, Dec. 2009.

[3] A. Barrón-Cedeño and P. Rosso. On Automatic Plagiarism Detection based on n-grams Comparison. *Proceedings of European Conference on Information Retrieval, ECIR-2009, Springer-Verlag, LNCS(5478)*, pages 696–700, 2009.

[4] A. Barrón-Cedeño, P. Rosso, E. Agirre, and G. Labaka. Plagiarism Detection across Distant Language Pairs. *Proceedings of the 23rd International Conference on Computational Linguistics,COLING-2010, Beijing, China* , pages 37–45, Aug. 2010.

[5] A. Barrón-Cedeño, P. Rosso, and J. Benedí. Reducing the Plagiarism Detection Search Space on the basis of the Kullback-Leibler Distance. *Proceedings 10th International Conference on Computational Linguistics and Intelligent Text Processing, CICLing-2009, Springer-Verlag, LNCS(5449)*, pages 523–534, 2009.

[6] A. Barrón-Cedeño, P. Rosso, D. Pinto, and A. Juan. On Cross-lingual Plagiarism Analysis Using a Statistical Model. In *Pro-*

*ceedings of 2nd Workshop on Uncovering Plagiarism, Authorship, and Social Software Misuse, PAN-2008*, pages 9–13, Patras, Greece, 2008.

[7] Y. Bernstein and J. Zobel. A scalable system for identifying coderivative documents. *Proceedings of the Symposium on String Processing and Information Retrieval*, 2004.

[8] S. Burrows, S. Tahaghoghi, and J. Zobel. Efficient plagiarism detection for large code repositories. *Software Practice and Experience*, 37:151–175, Sept. 2006.

[9] P. Clough and M. Stevenson. Developing A Corpus of Plagiarised Short Answers. *Language Resources and Evaluation: Special Issue on Plagiarism and Authorship Analysis*, 45(1):5–24, 2010.

[10] P. Cunningham and A. Mikoyan. Using CBR techniques to detect plagiarism in computing assignments. *Department of Computer Science, Trinity College, Dublin (Internal Report)*, Sept. 1993.

[11] R. A. Española. Real academia española. diccionario de la lengua española. vigésima segunda edición, 2008.

[12] E.Yang and J. Kieffer. On the performance of data compression algorithms based upon string matching. *IEEE Transactions on Information Theory*, 44:47–65, 1998.

[13] J. Faidhi and S. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming enviroment. *Computers and Education*, 11:11–19, 1987.

[14] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno. Detecting Source Code Reuse across Programming Languages. *27th Conference of the Spanish Society for Natural Language Processing (SEPLN 11)*, Sept. 2011.

[15] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno. Towards the detection of cross-language source code reuse. *Proceedings of 16th International Conference on Applications of Natural Language to Information Systems, NLDB-2011, Springer-Verlag, LNCS(6716)*, pages 250–253, 2011.

[16] G. Frantzeskou, S. MacDonell, E. Stamatatos, and S. Gritzalis. Examining the significance of high-level programming features in source code author classfication. *The Journal of Systems and Software*, 81(3):447–460, 2008.

[17] E. Gabrilovich and S. Markovitch. Computing semantic relatedness using wikipedia-based explicit semantic analysis. In *In Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1606–1611, 2007.

[18] Y. HaCohen-Kerner, A. Tayeb, and N. Ben-Dror. The state of authorship attribution studies: Some problems and solutions. *Computers and the Humanities*, 31(4):351–365, 1997.

[19] Y. HaCohen-Kerner, A. Tayeb, and N. Ben-Dror. Detection of simple plagiarism in computer science papers. *Huang and Jurafsky*, pages 421–429, 2010.

[20] M. H. Halstead. Naturals laws controlling algorithm structure? *SIGPLAN Noticies*, 7(2), Feb. 1972.

[21] H. T. Jankowitz. Detecting plagiarism in student pascal programs. *The Computer Journal*, 31(1), 1988.

[22] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM ournal of Research and Development*, 31(2):249–260, 1987.

[23] F. Peng, D. Schuurmans, V. Keselj, and S. Wang. Automated authorship attribution with character level language models. *Proceedings of 10th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2003)*, pages 267–274, 2003.

[24] D. Pinto, J. Civera, A. Barrón-Cedeño, A. Juan, and P. Rosso. A statistical approach to crosslingual natural language tasks. *Journal of Algorithms*, 64(1):51–60, 2009.

[25] M. Potthast, A. Barrón-Cedeño, B. Stein, and P. Rosso. Cross-language plagiarism detection. *Language Resources and Evaluation, Special Issue on Plagiarism and Authorship Analysis*, 45(1):45–62, 2011.

[26] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[27] S. Robertson and S. Walker. Okapi/keenbow at trec-8. *The Eighth Text Retrieval Conference (TREC-8)*, pages 151–162, 1999.

[28] F. Rosales, A. García, S. Rodríguez, J. L. Pedraza, R. Méndez, and M. M. Nieto. Detection of plagiarism in programming assignments. *IEEE Transactions on Education*, 51(2):174–183, 2008.

[29] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. *ACM SIGMOD Conference*, pages 76–85, June 2003.

[30] R. W. Selby. Quantitative studies of software reuse. *Software Reusability*, 2:213–233, 1989.

[31] E. Stamatatos. Intrinsic plagiarism detection using character n-gram profiles. *Proceedings of SEPLN'09, Donostia, Spain*, pages 38–46, 2009.

[32] E. Stamatatos, N. Fakotakis, and G. Kokkinakis. Computer-based authorship attribution without lexical measures. *Computers and the Humanities*, 35(2):193–214, 2001.

[33] B. Stein, M. Koppel, and E. Stamatatos. Plagiarism analysis, authorship identification, and near-duplicate detection. *SIGIR Forum (PAN 2007)*, 41(2):68–71, 2007.

[34] G. Whale. Detection of plagiarism in student programs. *Journal of Systems and Software*, 13:131–138, 1990.

[35] G. Whale. Identification of program similarity in large populations. *The Computer Journal*, 33(2), 1990.

[36] G. Whale. Software metrics and plagiarism detection. *Proceedings of the Ninth Australian Computer Science Conference, Canberra*, pages 231–241, 1990.

[37] M. J. Wise. Detection of similarities in student programs: Yaping may be preferable to plagueing. *Proceedings of the 23th SIGCSE Technical Symposium*, 1992.

[38] H. Xiong, H. Yan, Z. Li, and H. Li. Buaa_antiplagiarism: A system to detect plagiarism for c source code. *Computational Intelligence and Software Engineering, CiSE 2009*, pages 1–5, 2009.

[39] L. Zhang, Y. Zhuang, and Z. Yuan. A program plagiarism detection model based on information distance and clustering. *Internacional Conference on Intelligent Pervasive Computing*, pages 431–436, 2007.

[40] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

# Apéndice A

# Publicaciones y desarrollo de la herramienta DeSoCoRe

Las investigaciones descritas en esta tesis han permitido la publicación de los siguientes artículos:

- E. Flores, A. Barrón-Cedeño, P. Rosso and L. Moreno. Detección de reutilización de código fuente entre lenguajes de programación en base a la frecuencia de términos. In: Proceedings IV Jornadas PLN-TIMM, Torres, Jaén, Spain, Abril 7-8, pp.21-26. ISBN 978-84-15364-00-9.
- E. Flores, A. Barrón-Cedeño, P. Rosso and L. Moreno. Towards the Detection of Cross-Language Source Code Reuse. In: Proceedings of 16th International Conference on Applications of Natural Language to Information Systems, NLDB-2011, Springer-Verlag, LNCS(6716), pp. 250-253 `http://dx.doi.org/10.1007/978-3-642-22327-3_31` **CORE C**.
- E. Flores, A. Barrón-Cedeño, P. Rosso and L. Moreno. Detecting source code reuse across programming languages. Poster at Conference of Sociedad Española para el Procesamiento del Lenguaje Natural (SEPLN), Huelva, Spain, 5-7 September.

Además de las publicaciones de este trabajo de investigación, también se ha llevado a cabo las siguientes acciones:

- Desarrollo de un prototipo de la plataforma de detección de reutilización de código fuente llamado DeSoCoRe (nombre del inglés *Detecting Source Code Re-use*). Este prototipo compara dos códigos fuente a nivel de fragmento. Los resultados se muestran gráficamente indicando las partes similares entre dos códigos fuente estando escritos en el mismo o distinto lenguaje de programación.
- Presentación de la herramienta DeSoCoRe en el III Encuentro Inter-Estudiantes TICnología que tuvo lugar en Valencia el 5 de Enero de 2012.