UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Detección de reutilización de código fuente monolingüe y translingüe

30 de Mayo de 2016

Enrique Flores Sáez

Tesis depositada en cumplimiento de los requerimientos para obtener el título de Doctor por la Universitat Politècnica de València

Directores: Dra. Lidia Moreno y Dr. Paolo Rosso

Cheater Cheater Pumpkin Eater
American children's folk rhyme

Agradecimientos

Durante el tiempo en que ha estado gestándose esta tesis han sido muchas las personas que han participado y colaborado, y a quienes quiero expresar mi gratitud por el apoyo y la confianza que han depositado en mí.

En primer lugar quiero agradecer a mis directores de tesis, Lidia y Paolo que, con sus consejos, dedicación, guía, enfoque, críticas (siempre constructivas) e incluso amistad, han conseguido que esto llegue desde el máster hasta terminar en lo que tenemos aquí. Del mismo modo, agradecer a Alberto por su participación, colaboración y ayuda que ha contribuido durante gran parte de esta tesis. Agradecer a Esaú su ayuda en la organización, preparación y defensa de los resultados de las competiciones que hemos organizado. También me gustaría agradecer a Rosa y Vicente por su ayuda y predisposición en la parte estadística de estas competiciones.

Como no, agradecer a mis compañeros de laboratorio durante estos años por la ayuda prestada y hacer el día a día un poco más ameno. También a los amigos de universidad, con especial énfasis del doctorado por estos años divirtiéndonos, compartiendo las penas y alegrías, sufriéndonos, bromeándonos e incluso celebrando alguna boda y tesis ya escritas. GRACIAS *GH-team*: Miquel (el samurái), Pastor (el káiser), Jaume (el holandés errante), Joan (el chino *unreachable*) y Mario (el patriarca *sloth*).

Así como he agradecido a gente de la UPV, también me gustaría agradecer a aquellos que durante mis estancias en México me han aportado mucho tanto académica como personalmente. Primero, quiero agradecer a Manuel Montes y Grigori Sidorov, mis responsables durante las estancias, y a sus respectivas instituciones (INAOE e IPN) por acogerme como un miembro más, aportando su conocimiento en mi doctorado y ayudarme a conocer y vivir (en) México. También agradecer a Marc y Martín su colaboración, ayuda y amistad tanto en el doctorado como en las estancias. Mi mayor agradecimiento a aquellas personas que durante la(s) estancia(s) formaron parte de mi día a día y espero conservar su amistad durante muchos años pese a la distancia: Alicia, Miguel y Carolina, Nicté, Liliana, Samuel, Alexia, Fani, Uri, Erick, Chris, Noé y Sabino entre muchos otros.

Parte fundamental de conseguir este objetivo lo tiene la familia y amigos, tanto los que están como los que ya no, con su apoyo incondicional e interés en saber como iba y funcionaba el doctorado, las estancias y el mundo de la investigación en general. 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.

Resumen

La detección automática de reutilización en códigos fuente consiste en determinar si un (fragmento de) código ha sido creado considerando otro como fuente. El plagio y las ramificaciones en proyectos software son dos ejemplos de tipos de reutilización en códigos fuente. Con la llegada de la Web y los medios electrónicos ha crecido enormemente la facilidad de acceso a códigos fuente para ser leídos, copiados o modificados. Esto supone una gran tentación para programadores que, con propósitos de reducir costes (temporales o económicos), deciden utilizar códigos fuente previamente depurados y probados. Este fenómeno ha causado que expertos en lenguajes de programación estudien el problema.

La gran cantidad de recursos disponibles en la Web hace imposible un análisis manual de códigos fuente sospechosos de haber sido reutilizados. Por ello, existe una necesidad urgente de desarrollar herramientas automáticas capaces de detectar con precisión los casos de reutilización. Basándose en técnicas del procesamiento del lenguaje natural y recuperación de información, las herramientas de detección automáticas de reutilización son capaces de realizar multitud de comparaciones de códigos fuente de forma eficiente.

En esta tesis proponemos un conjunto de modelos que pueden aplicarse indistintamente a nivel monolingüe o translingüe. Es decir, se pueden comparar dos códigos que están escritos en el mismo, o en distinto, lenguaje de programación. Por lo tanto, nos permite realizar comparaciones entre casi cualquier par de lenguajes de programación a diferencia de las propuestas del estado de la cuestión. Inicialmente, hemos estudiado las modificaciones más comunes realizadas por los programadores para evitar ser detectados. Para tratar estas modificaciones y mejorar la detección, hemos propuesto una serie de preprocesos. Se han evaluado y analizado los modelos tanto en un escenario académico real como en un escenario de detección a gran escala. Finalmente, nuestras mejores propuestas se han comparado con otras propuestas del estado de la cuestión dentro de un mismo marco de evaluación.

Estas pruebas de nuestros modelos se han realizado mediante millones de comparaciones tanto a nivel monolingüe como translingüe empleando diversas técnicas que fueron efectivas al aplicarlas sobre textos escritos en lenguaje natural. La mayor parte de los recursos desarrollados en el marco de esta tesis están a libre disposición de la comunidad científica. Utilizando parte de estos recursos, hemos configurado dos escenarios (monolingües y translingües) de evaluación que son un referente para que actuales y futuros trabajos de investigación puedan ajustar y comparar sus propuestas.

Resum

La detecció automàtica de reutilització en codis consisteix a determinar si un (fragment de) codi ha sigut creat considerant un altre com a font. El plagi i les bifurcacions en projectes de programari són dos exemples de tipus de reutilització en codis font. Amb l'arribada de la Web i els mitjans electrònics ha crescut enormement la facilitat d'accés a codis font per a ser llegits, copiats o modificats. Açò suposa una gran temptació per a programadors que amb propòsits de reduir costos (temporals o econòmics) decideixen utilitzar codis font prèviament depurats i provats. Aquest fenomen ha causat que experts en llenguatges de programació estudien aquest problema.

La gran quantitat de recursos en la Web fa impossible una anàlisi manual de codis font sospitosos d'haver sigut reutilitzats. Es per aquest motiu que existeix una necessitat urgent de desenvolupar eines automàtiques capaces de detectar amb precisió els casos de reutilització. Basant-se en tecnologies de teoria de llenguatges i recuperació d'informació, les eines de detecció automàtiques de reutilització són capaces de realitzar multitud de comparacions de codis font de forma eficient.

En aquesta tesi proposem un conjunt de models que poden aplicar-se indistintament a nivell monolingüe o translingüe. És a dir, es poden comparar dos codis que estan escrits en el mateix, o diferent, llenguatge de programació. Per tant, ens permet realitzar comparacions entre quasi qualsevol parell de llenguatges de programació a diferència de les propostes de l'estat de la qüestió. La nostra experimentació ha seguit un cert paral·lelisme entre la detecció de reutilització monolingüe i la translingüe. Inicialment, hem estudiat les modificacions més comunes realitzades pels programadors per evitar ser detectats. Per tractar aquestes modificacions i millorar la detecció, hem proposat una sèrie de preprocesos. S'han avaluat i analitzat els models tant en un escenari acadèmic real com en un escenari de detecció a gran escala. Finalment, hem comparat les nostres millors propostes amb altres propostes de l'estat de la qüestió dins d'un mateix marc d'avaluació.

Aquestes proves i comparacions dels nostres models s'han realitzat mitjançant milions de comparacions tant a nivell monolingüe com translingüe emprant diverses tècniques que van ser efectives en aplicar-se sobre textos escrits en llenguatge natural. La major part dels recursos creats en el marc d'aquesta tesi han estat de creació pròpia i estan a lliure disposició de la comunitat científica. Utilitzant part d'aquests recursos, hem proposat dos escenaris (monolingüe i translingüe) d'avaluació que són un referent perquè actuals i futurs treballs d'investigació puguin ajustar i comparar les seves propostes.

Abstract

Automatic detection of source code re-use consists in determining whether a (piece of) code has been created considering another source. Plagiarism and forks in software projects are two examples of types of re-use in source codes. With the advent of the Web and electronic media it has grown enormously the ease of access to source code to be read, copied or modified. This represents a great temptation for developers with the aim of reducing (time or economic) costs, decide to use previously debugged and tested source codes. This phenomenon has caused experts in programming languages to study the problem.

The large amount of resources available on the Web makes impossible a manual analysis of suspect source codes of being re-used. Therefore, there is an urgent need to develop automated tools that can accurately detect re-used cases. Automatic re-use detection tools based on natural language processing techniques and information retrieval are able to perform many comparisons of source codes efficiently.

In this thesis we propose a set of models that are suitable at both monolingual or crosslingual level. That is, two source codes written in the same, or different, programming language can be compared. Therefore, it allows us to make comparisons between almost any pair of programming languages unlike the proposals of the state of the art. First, we studied the most common changes made by programmers to avoid the detection. To address these changes and improve the detection, we have proposed a set of pre-processing. The models have been evaluated and analysed in real academic settings as well as large-scale scenarios. Finally, our best proposals were compared with some of the state of the art proposals within the same evaluation framework.

These tests of our models were performed millions of monolingual and crosslingual comparisons using several techniques that were effective when applied to detection re-use in texts written in natural language. Most of the resources developed in the framework of this thesis are freely available to the scientific community. Using part of these resources, we have set up two evaluation scenarios (monolingual and crosslingual) that are a reference for current and future research works can adjust and compare their proposals.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

En la actualidad hay una gran facilidad de acceso a la información gracias a la masificación de recursos y contenidos en Internet como blogs, foros, repositorios o cualquier contenido digital. Los usuarios tienen disponible contenido diverso en la Web como música, libros, imágenes o programas que pueden ser leídos, copiados e incluso modificados fácilmente. Esta facilidad de acceso a la información permite a usuarios malintencionados utilizar cómodamente contenido de otro autor como propio. El fenómeno de la reutilización[1] es un problema creciente en entornos tales como el ámbito académico. En torno al 80 % de estudiantes universitarios admitieron haber reutilizado contenidos al menos una vez (Chapman et al., 2004). La dificultad que supone la comprobación de un documento contra todo el espacio Web hace patente la necesidad de disponer de herramientas automáticas que realicen esta comprobación. Es evidente que los canales de información y persuasión al alumno en este tema no siempre resultan ser tan satisfactorios, así como tampoco la dificultad añadida de prohibir copiar y pegar fragmentos de código fuente entre alumnos. En resumen, además de la educación sobre éticas del comportamiento y añadir barreras que dificulten la reutilización, también son necesarios los sistemas que facilitan la detección de casos de plagio entre estudiantes.

Dentro del ámbito académico encontramos dos tipos de reutilización muy frecuentes: la reutilización de textos y la reutilización de códigos fuente. La reutilización de textos se ha definido como "la situación en la cual un material previamente escrito es utilizado de nuevo conscientemente para la creación de nuevo texto o versiones" según el trabajo de Clough (2003). En los últimos años, se ha observado un considerable interés en crear nuevos modelos para mejorar la detección de reutilización automática en textos.[2] Se han detectado muchos tipos de aprovechamiento de material ajeno como: copia literal, copia con sustitución por sinónimos, re-estructuración de las oraciones manteniendo el sentido original e incluso traducción desde otro idioma.

A pesar de que se han realizado más esfuerzos en la detección de reutilización en textos, también existen propuestas diversas en la detección sobre códigos fuente (Arwin et al., 2006; El Bachir

[1] Nos referimos al concepto de reutilización como hiperónimo de plagio, es decir, cuando tenemos certeza de que (parte de) un contenido se ha tomado de un documento diferente pero sin ninguna información acerca del consentimiento del autor original.

[2] Ver por ejemplo la competición internacional PAN `http://pan.webis.de`.

Menai et al., 2010; Prechelt et al., 2002; Rosales et al., 2008). La reutilización en códigos fuente ocurre cuando un programador utiliza (parte de) un programa "...escrito por otra persona y, intencionalmente o involuntariamente, no realiza el reconocimiento adecuado, presentándolo como un trabajo propio" según Cosma et al. (2008). Para los programadores, el fácil acceso a códigos fuente supone una gran tentación para reutilizar códigos fuente que han sido testeados y verificados. Esta reutilización de código fuente responde a propósitos tales como la reducción del tiempo de programación, reducir errores o costes económicos en el desarrollo del software.

Tanto a nivel académico como a nivel empresarial, se han encontrado múltiples casos de uso no autorizado de código fuente. La existencia de licencias que permiten utilizar código fuente de otros programadores hace necesario hablar de reutilización y no de plagio. En esta tesis, no se pretende comprobar si existe un uso legítimo o no de códigos fuente sino de encontrar colaboración entre programadores. Según un informe de la *Business Software Alliance*, las pérdidas por uso fraudulento de software asciende a billones de euros.[3] Actualmente, las empresas de software tienen un especial interés en preservar su propiedad intelectual. En una encuesta sobre 3 970 desarrolladores, más del 75 % admitió que había reutilizado fragmentos de códigos fuente de fuentes externas para desarrollar su propio software.[4]

1.1 Motivación y objetivos

La mayoría de estudios realizados fueron aplicados sobre grupos cerrados (Arwin et al., 2006; Rosales et al., 2008; Wise, 1992; Prechelt et al., 2002) debido a que la reutilización de código fuente se suele producir a partir de trabajos propios previos o, de los trabajos de compañeros. Sin embargo, la expansión de la Web ha permitido el salto de los entornos educativos a Internet. Dos ejemplos de ello son las plataformas educativas Coursera[5] y UPVX[6]. En estas plataformas, un gran número de estudiantes necesitan resolver los mismos problemas para poder certificar los conocimientos adquiridos. Por lo tanto, la tarea de detección de reutilización tiene una dificultad añadida porque todos los códigos fuente contienen (hasta cierto punto) un grado de similitud por tener que resolver la misma tarea. En el curso de Procesamiento del Lenguaje Natural (PLN) de Coursera de 2012, más de 50 000 estudiantes fueron inscritos. Esta masificación de los entornos académicos requiere que se incluyan mecanismos de control y mejora de su aprendizaje (Kuo et al., 2010). No solo se aprecia este crecimiento en la enseñanza en línea; también se observa en concursos de programación tales como las olimpiadas internacionales de programación o la *ACM International Collegiate Programming Contest*.[7] Así como en los entornos tradicionales académicos, estos entornos masivos de programación no aprueban la cooperación entre participantes para resolver las tareas. El empleo de herramientas de evaluación automáticas de trabajos de programación mejoran la experiencia de aprendizaje de los estudiantes a la vez que desalienta la colaboración no permitida entre estudiantes (Spinellis et al., 2007). Además, la combinación de las herramientas de evaluación con detectores de reutilización en códigos fuente debería reducir significativamente el plagio entre estudiantes. En la misma Universitat Politècnica de València desde Marzo del 2015 se ha implantado

[3] `http://globalstudy.bsa.org/2013/`

[4] `http://www.out-law.com/page-4613`

[5] `https://www.coursera.org/`

[6] `https://www.upvx.es/`

[7] Olimpiadas Informáticas: `http://www.ioinformatics.org/`, Olimpiadas Informáticas Españolas: `https://olimpiada-informatica.org/` y competición de *Association for Computing Machinery*: `http://icpc.baylor.edu/`

dentro de la plataforma PoliformaT[8] la herramienta Turnitin[9] para dar soporte al profesorado en la detección y disuasión de la reutilización en los trabajos académicos (*Turnitin en PoliformaT* 2015).

Una encuesta reciente (Chuda et al., 2012) revela que la reutilización de código fuente representa el 30 % de la reutilización en el mundo académico. Además, un 63 % admitieron poner a disposición de sus compañeros sus trabajos alguna vez. En este contexto, los estudiantes están altamente tentados a reutilizar códigos fuente porque muy frecuentemente tienen que resolver problemas similares o incluso los mismos para sus tareas escolares. Por lo tanto, la tarea de detectar reutilización en código fuente se convierte en una tarea aún más complicada dado que dentro de la detección existe un solapamiento considerable en el problema a resolver. Por otra parte, la enorme cantidad de comparaciones manuales que requieren colecciones de mediano y gran tamaño crea la necesidad de desarrollar herramientas automáticas para detectar con precisión la reutilización en códigos fuente. La inmensa cantidad de repositorios existentes en la Web hacen el análisis manual de reutilización en códigos fuente inabordable. Los sistemas de detección de reutilización automáticos son capaces de recuperar candidatos a ser casos de reutilización casi en tiempo real.

El enfoque clásico para la detección de reutilización se ha basado en la estructura de ejecución del código fuente, siendo su aplicación a nivel monolingüe. Sin embargo, un tipo particular de reutilización de código fuente que está siendo estudiado recientemente es el escenario de reutilización translingüe (Arwin et al., 2006; Flores et al., 2015b). Este ocurre cuando un programador/a encuentra un código fuente escrito en un lenguaje de programación LP pero lo necesita en un lenguaje distinto LP'. Mediante una traducción manual o automática se puede cometer reutilización de código fuente translingüe. Otra posible situación es la de recuperar códigos fuente cuando un programador necesita una determinada implementación de un algoritmo en un lenguaje de programación LP' pero este solo dispone de la implementación en LP. Este escenario supone un reto mayor que la detección monolingüe porque distintos lenguajes de programación pueden no compartir ciertas palabras reservadas, librerías y sintaxis de programación. Generalmente el problema de la reutilización translingüe se ha abordado entre lenguajes de programación muy cercanos que incluso comparten un compilador para ambos lenguajes y así poder abordar el problema como un caso (pseudo-)monolingüe. En esta tesis de investigación se pretende aplicar un enfoque nuevo y que además sea aplicable a nivel translingüe sin restricción de lenguajes de programación. En concreto nos centramos en determinar si técnicas que han funcionado en la detección de plagio en lenguaje natural son aplicables a lenguajes de programación entendiendo estos como lenguajes con sus sintaxis, reglas ortográficas y vocabulario específico.

El cúmulo de escenarios donde se comete reutilización tanto a nivel académico como empresarial ha motivado este trabajo de investigación marcando los siguientes objetivos a cumplir:

- Desarrollo de técnicas utilizadas en el ámbito del procesamiento del lenguaje natural y recuperación de información, adaptándolas para detectar reutilización en códigos fuente.
- Desarrollo y aplicación de modelos de traducción a la detección de reutilización translingüe en códigos de programación.
- Creación de varios corpus para la detección de reutilización en diferentes lenguajes de programación y contextos tanto a nivel monolingüe como a nivel translingüe.

[8] `https://poliformat.upv.es/`
[9] `http://turnitin.com/`

- Obtención de nuevos recursos a partir de competiciones de programación.
- Desarrollo de una herramienta on-line para la detección de reutilización (monolingüe y translingüe) entre códigos de programación.
- Impulsar y organizar competiciones internacionales de detección de reutilización en códigos fuente que permitan difundir la investigación en el área y, a su vez, promover un marco de evaluación común para futuros trabajos de investigación.

1.2 Preguntas de investigación

Durante la elaboración de esta tesis doctoral hemos identificado distintos problemas para abordar la reutilización automática en códigos fuente:

1. La reutilización de códigos fuente supone en muchos entornos casos de plagio, por lo que debido a problemas éticos y legales, no se encuentran colecciones de códigos fuente disponibles para su estudio.
2. Los escasos escenarios/casos de estudios disponibles del estado de la cuestión generalmente son de tamaño reducido o incluso colecciones para demostraciones.
3. La reutilización de código fuente se puede realizar entre lenguajes de programación incluso añadiendo una modificación posterior del código fuente traducido.

Estos restos nos llevan a las siguientes preguntas de investigación:

Preguntas sobre detección de reutilización

1. ¿Qué modificaciones realiza un programador para evitar la detección?
2. ¿Se pueden aplicar modelos de detección de reutilización en textos sobre casos de reutilización en códigos fuente?
3. ¿Qué modelos obtienen mejores resultados para la detección de reutilización?
4. ¿Cómo podemos abordar la detección translingüe?
5. ¿Existen distintos tipos de reutilización translingüe?
6. ¿Podemos detectar del mismo modo estos distintos tipos de reutilización translingüe?

Preguntas sobre recursos

7. ¿Cómo construir colecciones de códigos fuente para el estudio y desarrollo de herramientas de detección automática de reutilización?
8. ¿Qué condiciones debe cumplir una colección de códigos fuente para considerar que contiene casos de reutilización?

9. ¿Es posible crear colecciones de códigos fuente con casos simulados de reutilización?
10. ¿Son útiles los casos simulados para la detección de casos reales de reutilización?
11. ¿Es posible crear colecciones de códigos fuente con casos de reutilización translingüe?

Nuestro trabajo de investigación se centra en las preguntas de investigación anteriormente enunciadas. Para ello, hemos adaptado modelos efectivos en la detección de reutilización en textos para detección de reutilización en códigos fuente. También hemos creado una serie de colecciones de códigos fuente de libre disposición tanto a nivel monolingüe como translingüe para propósitos de investigación en el campo de la detección automática de reutilización en códigos fuente. En el marco de actividades de las competiciones PAN, hemos organizado tareas de detección de reutilización a nivel monolingüe y translingüe creando un marco de evaluación común para investigaciones en el área.

1.3 Contribuciones

Las principales contribuciones de esta tesis se encuentran resumidas a continuación.

Se ha abordado el problema de la detección de reutilización en códigos fuente basándose en técnicas de procesamiento del lenguaje natural y recuperación de información. De este modo, hemos propuesto y adaptado un conjunto de modelos que mostraron un buen rendimiento en la detección de reutilización y similitud en textos. Los resultados obtenidos por estos modelos han mostrado ser válidos y más eficaces que las demás aproximaciones del estado de la cuestión. (Capítulos 4, 5 y 6)

Se han identificado las modificaciones más frecuentes en el entorno académico para evadir la detección de reutilización en códigos fuente. La identificación del cómo se realiza la reutilización de código fuente es el primer paso para estudiar cual es la mejor forma de detectarlo. También es importante conocer y diferenciar los mecanismos/modificaciones que se realizan tanto a nivel monolingüe como a nivel translingüe. (Apartados 5.1 y 5.2.1)

La ronda inicial de la competición internacional Google Code Jam se ha analizado en búsqueda de reutilización de códigos fuente. En esta ronda se pretende descubrir si en un entorno propicio para la reutilización, esta es cometida y en qué grado. Es importante detectar actitudes deshonestas tanto en competiciones de programación como en entornos académicos. (Apartados 3.2.1 y 5.1.2)

Se ha analizado el repositorio de códigos fuente Rosettacode.org en busca de casos de reutilización translingüe. También se han generado casos simulados de reutilización translingüe entre los lenguajes de programación más populares. Sobre este repositorio hemos estudiado el desempeño de los modelos propuestos recuperando códigos fuente relacionados y traducidos. Este repositorio supone un gran recurso para la investigación en la detección automática translingüe, así como también es posible que sea la fuente de futuros casos de reutilización en códigos fuente. (Apartados 3.2.2 y 5.2.3)

Se han creado colecciones que contienen reutilización de códigos fuente de características variadas. Parte de las colecciones son recursos monolingües aunque también se han creado recursos translingües. Dependiendo del objetivo, se han utilizado colecciones de gran tamaño, o de tamaño medio.

Además, también se han creado casos de reutilización simulados para realizar experimentos con un mayor número de muestras. (Capítulo 3)

Se han propuesto entornos de evaluación para la detección de reutilización en códigos fuente. Tanto a nivel monolingüe como translingüe se han facilitado corpus de entrenamiento y evaluación para comparar las propuestas en las mismas condiciones. Estos marcos de evaluación han sido propuestos dentro del laboratorio PAN para la detección de reutilización en código fuente (SOCO) y para la detección de reutilización translingüe en código fuente (CL-SOCO). (Capítulo 6)

1.4 Estructura de la tesis

El trabajo realizado en el marco de esta tesis doctoral para resolver la problemática de la reutilización de código fuente se presenta en los capítulos y apéndices que se describen a continuación:

- **Capítulo 2** Estado de la cuestión

  En este capítulo se muestra una visión general de la detección de reutilización. Se presta especial atención en la distinción entre plagio y reutilización. Primero se hace una revisión de los principales trabajos sobre la detección de reutilización en textos que ha sido el punto de arranque de esta tesis y a continuación, se muestra el estado de la cuestión de la detección de reutilización en código fuente tanto a nivel monolingüe como a nivel translingüe.

- **Capítulo 3** Recursos

  Este capítulo describe los corpus utilizados en este trabajo de tesis. Debido a la enorme dificultad para conseguir recursos con los que poder entrenar los sistemas desarrollados y compararlos con otros trabajos, la totalidad de los recursos han sido recopilados, tratados y etiquetados durante el trabajo de investigación de esta tesis. Actualmente, todos ellos se encuentran disponibles para fines de investigación.

- **Capítulo 4** Modelos propuestos

  Este capítulo propone diferentes modelos para resolver la problemática de la detección automática de reutilización en códigos fuente. Estos modelos están orientados a considerar los códigos fuente como si de un texto se tratara. La principal ventaja de utilizar modelos diseñados para trabajar con textos es que permite evitar utilizar una herramienta de compilación para cada lenguaje de programación. Por otra parte, facilita enormemente el tratamiento de la detección translingüe de códigos fuente ya que no se requiere manejar múltiples compiladores.

- **Capítulo 5** Experimentación

  Este capítulo está dividido en dos partes. La primera parte describe la experimentación realizada a nivel monolingüe, para detectar la reutilización entre códigos fuente escritos en un mismo lenguaje de programación. La segunda parte representa la contribución más novedosa de este trabajo de investigación. Los pocos trabajos que existen en la detección de reutilización translingüe de códigos fuente solo son capaces de trabajar entre un conjunto reducido de lenguajes de programación. En esta parte, se describen los experimentos realizados con los mo-

delos propuestos en el anterior capítulo que son capaces de realizar la detección prácticamente entre cualquier par de lenguajes de programación.[10]

- **Capítulo 6** Evaluación en la competición internacional de detección de reutilización en código fuente

  Con el objetivo de promover el desarrollo de sistemas capaces de detectar la reutilización en código fuente, en el marco de esta tesis doctoral se ha organizado una competición durante dos años consecutivos. En la primera parte del capítulo se describe la competición centrada en detección de reutilización monolingüe mientras que la segunda parte se centra en reutilización translingüe. En ambos casos se presenta un estudio comparativo de los resultados conseguidos para los sistemas participantes en cada competición respeto a los resultados conseguidos por los sistemas de mejor rendimiento desarrollados en el marco de esta tesis.

- **Capítulo 7** Conclusiones y trabajos futuros

  Este capítulo resume las principales aportaciones de este trabajo de investigación, además de responder a las preguntas de investigación planteadas en la introducción. También se enuncian las posibles líneas a seguir en trabajos futuros.

- **Apéndice A** Herramienta DeSoCoRe.

  La herramienta DeSoCoRe diseñada durante este trabajo de investigación es descrita en este apéndice. Se trata de una herramienta visual capaz de comparar códigos fuente a nivel de función entre diferentes lenguajes de programación. En este apéndice se detalla su comportamiento así como su manejo.

- **Apéndice B** Publicaciones relacionadas

  Las publicaciones científicas derivadas de este trabajo de investigación se muestran en este apéndice. Este incluye artículos en revistas, conferencias y jornadas. Además, se incluye un resumen del alcance de las publicaciones.

[10] Entre los lenguajes de programación existen lenguajes inherentes al estilo de programación y también enfocados por objetivo. Por ejemplo, existen lenguajes de programación orientados a objetos, orientados a programación concurrente y funcional. También pueden ser separados según la cercanía al lenguaje máquina (lenguajes de bajo nivel) o la cercanía al lenguaje natural (lenguajes de alto nivel). La reutilización entre distintos paradigmas de programación se considera marginal/prácticamente inexistente por lo tanto no se estima conveniente realizar detección automática de reutilización entre lenguajes de programación de distinta naturaleza.

Capítulo 2

Estado de la cuestión

En este capítulo se presentan los principales trabajos en detección de reutilización. En el apartado 2.1 se resume el estado de la cuestión en detección de reutilización sobre textos. Se enuncian trabajos y herramientas de detección a nivel monolingüe, terminando el apartado con un corto análisis de los modelos propuestos a nivel translingüe. Algunos de los modelos propuestos en esta tesis aplicados a códigos fuente están inspirados en los trabajos de detección de reutilización en textos descritos en este capítulo. El segundo apartado de este capítulo trata sobre la detección de reutilización en códigos fuente. A su vez, este se encuentra dividido en dos partes, separando la detección monolingüe y translingüe. En el apartado 2.2.1 se describe la problemática de la reutilización en códigos fuente, se comentan áreas relacionadas como son la ofuscación de código fuente, detección de clones o atribución de autoría en código fuente, además de discutir los principales trabajos en detección monolingüe de reutilización en código fuente. En el apartado 2.2.2 se describen los pocos trabajos conocidos en detección translingüe de reutilización en códigos fuente. La falta de trabajos en este campo ha sido la gran motivación para adaptar y aplicar sobre códigos fuente técnicas que obtuvieron buenos resultados en detección translingüe de textos.

2.1 Detección automática de reutilización en textos

Existe cierta ambigüedad que es necesario precisar a la hora de hablar de reutilización y plagio entre obras, ya sea a nivel de textos escritos en lenguaje natural como a nivel de códigos fuente (o programas) escritos en un lenguaje de programación. 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 que a lo largo de esta tesis, se usará con mayor frecuencia.

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 fuertemente ligada a otros campos del PLN como puede ser la atribución de autoría (Rudman, s.f.; Stein et al., 2007a), 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; y *(iv)* traducción de texto siendo su detección compleja debido a que es necesario conocer ambos idiomas y las traducciones muchas veces precisan de cambios por formas que no existen en alguno de los lenguajes.

A la hora de abordar la detección de reutilización es necesario distinguir diferentes enfoques en los que se va a centrar el proceso, como por ejemplo si se va a abordar un problema monolingüe o translingüe. Resulta 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 (Barrón-Cedeño et al., 2010).

También se puede clasificar el tipo 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 textos potencialmente sospechosos de reutilización (Stein et al., 2007b; Potthast et al., 2009).

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 este se compara con el estilo de cada fragmento de texto (Clough et al., 2010). Si existe una variación del estilo en un fragmento, este se marca para una revisión manual para que un experto humano tome la decisión final. Esto implica que recae en el revisor la decisión de si se ha producido o no reutilización. Un ejemplo de herramienta que muestra 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.

Una de las aproximaciones de mayor eficacia en atribución de autoría (Stamatatos, 2009a) que ha sido aplicada para detección intrínseca de plagio está basada en perfiles de caracteres de n-gramas (Stamatatos, 2009b). El buen desempeño de esta aproximación reside en la capacidad de capturar conectores del lenguaje (en inglés *function words*), (la mayoría son de tamaño menor a cuatro caracteres), sufijos y prefijos. La idea de los perfiles de caracteres de n-gramas es la de caracterizar el documento como un vector de n-gramas más frecuentes para estimar el nivel de similitud entre otros documentos e incluso fragmentos.

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

Código fuente

```
for i = 1:100
  if ( mod(i,15) == 0 )
    disp("FizzBuzz");
  elseif ( mod(i, 3) == 0 )
    disp("Fizz")
  elseif ( mod(i, 5) == 0 )
    disp("Buzz")
  else
    disp(i)
  endif
endfor
```

Tokens extraídos

```
[
    "for",
    "i",
    "if",
    "mod",
    "disp",
    "FizzBuzz",
    "elseif",
    "Fizz",
    "Buzz",
    "else",
    "endif",
    "endfor"
]
```

Bolsa de palabras

```
[
    ["for", 1]
    ["i", 4]
    ["if", 1]
    ["mod", 3]
    ["disp", 3]
    ["FizzBuzz", 1]
    ["elseif", 2]
    ["Fizz", 1]
    ["Buzz", 1]
    ["else", 1]
    ["endif", 1]
    ["endfor" 1]
]
```

Figura 2.1: Ejemplo de la extracción de tokens de un código fuente y su posterior conversión a bolsa de palabras. En el ejemplo se han excluido los números como tokens del código fuente pero estos pueden ser considerados dependiendo de la aproximación utilizada.

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., 2007a) 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 casos de reutilización.

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 (Barrón-Cedeño et al., 2009b). Respecto a la etapa de comparación, existen distintas aproximaciones pero, en general, en todas ellas se distinguen tres fases: preproceso, representación de la información y aplicación de una medida de similitud.

Para poder comparar textos se han considerado 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 paro, 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 posible preproceso que consiste en sustituir todas las palabras que sean derivadas de una por la raíz de esta última. Con esta técnica (en inglés llamado *stemming*) se detecta mejor la paráfrasis del texto como por ejemplo cambiar el tiempo verbal de una oración.

Para representar la información tras realizar el preproceso se han utilizado diversas técnicas que van desde construir *bolsas de palabras* (en inglés *bag of words*), hasta representaciones en *huellas digitales*

$código_1$	=	size = os.path.getsize('input.txt')
$código_2$	=	size = os.path.getsize('/input.txt')
$md5sum(código_1)$	=	$944c926e8ded46b7065d1d2473488233$
$md5sum(código_2)$	=	$51bd30e1e52a16365e9f1f2112366f1a$

Figura 2.2: Ejemplo de una función hash sobre dos cadenas escritas en el lenguaje Python. Solo un caracter cambia por completo el resultado de la función hash.

(en inglés *fingerprint*). El concepto de bolsa de palabra 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 de caracteres se refiere a una secuencia de caracteres contiguos de tamaño n. Mientras que los n-gramas de palabras consisten en una secuencia de palabras contiguas de una cantidad n (Barrón-Cedeño et al., 2009c). En la figura 2.1 se muestra un pequeño ejemplo de la creación de la bolsa de palabras. Por otra parte, el concepto de huella digital consiste en a partir de fragmentos del texto generar 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. En la figura 2.2 se representa un ejemplo de función hash sobre dos fragmentos de códigos fuente mediante la suma MD5 (Rivest, 1992).

Las representaciones de cada texto se comparan entre sí para estimar su similitud aplicando distintas medidas. En el trabajo de Barrón-Cedeño et al. (2009a) 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 una sencilla métrica, el coeficiente de Jaccard (Jaccard, 1901), que consiste en dividir el tamaño de la intersección de dos conjuntos de términos entre el tamaño de su unión como se observa en la ecuación 2.1. Como se puede observar, el coeficiente de Jaccard no considera la frecuencia del término (cuántas veces ocurre el término en el documento) ni tampoco su rareza (en cuántos documentos de la colección aparece el término).

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

En el mejor de los casos, cuando se trata de máxima similitud, 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, cuando no hay similitud, 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 métricas más utilizadas es la de la similitud del coseno, que consiste en calcular el ángulo que forman dos conjuntos de términos dentro de un espacio vectorial. La explicación de esta fórmula se encuentra más detallada en el apartado 4.2 por haber sido utilizada en los experimentos de este trabajo de investigación.

Una aproximación interesante se encuentra propuesta en el trabajo de Stamatatos (2011). Este trabajo parte de la idea de que al modificar (parte de) un texto, las palabras que se modifican

son reemplazadas por sinónimos. Por lo tanto, palabras tales como determinantes o conectores permanecerán invariables en la mayoría de casos. A partir de este concepto, establece el concepto de n-gramas de stopwords. Considerando los n-gramas de stopwords obtiene unos resultados competitivos comparados con los mejores de la competición de detección de plagio PAN-10[2] (Potthast et al., 2010a). Además obtiene resultados significativamente mejores en la detección de casos de alta reutilización donde la mayoría de palabras se ha reemplazado por sinónimos.

Entre los modelos basados en huella digital se utiliza la técnica de *Winnowing* (Schleimer et al., 2003) y SPEX (Bernstein et al., 2004). La técnica de winnowing consiste en generar un valor hash al texto para cada n-grama de caracteres obteniendo una secuencia de *hashes*. Estos n-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, para detectar la similitud de los textos se compararan los valores de sus huellas. Por otra parte, 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 se genera una secuencia de hashes de los documentos. Sobre estas secuencias se generan bigramas, y seleccionan 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.

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. Un ejemplo de estos métodos es la divergencia de Kullback-Leibler (Kullback et al., 1951). La divergencia de Kullback-Leibler es un indicador de la similitud entre dos funciones de distribución. Consiste en medir cómo de cerca están dos distribuciones de probabilidad A y B así como se muestra en la ecuación 2.2.

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

El sistema PPChecker (Kang et al., 2006) calcula la cantidad de información copiada del documento original basándose en patrones lingüísticos observados en los casos de plagio. Este sistema utiliza como unidad de comparación las frases donde los autores identificaron cinco tipos de patrones: *(i)* copia exacta; *(ii)* palabras añadidas; *(iii)* palabras eliminadas; *(iv)* palabras sustituidas entre oraciones; y *(v)* la frase cambiada al completo. La sustitución de palabras se identifica mediante el recurso WordNet[3]. A través de WordNet se identifican palabras con poca "distancia" semántica. Estos patrones se identifican en base a tres condiciones: *(i)* número de palabras en común; *(ii)* número de palabras distintas; y *(iii)* el tamaño de la intersección de palabras. Cada patrón requiere un valor diferente de similitud dado que en la copia exacta se espera una (casi) completa intersección de palabras mientras que cuando se sustituyen palabras este valor será menor. PPChecker obtuvo mejores resultados que otros modelos. Concretamente, se comparó con dos modelos como referencia de detección de copias basados en elementos sintácticos, uno a nivel de documento y el otro a nivel de oración.

[2] http://pan.webis.de/clef10/pan10-web/plagiarism-detection.html
[3] `https://wordnet.princeton.edu/`

Además de PPChecker se han propuesto multitud de sistemas de detección de copias en documentos como COPS, SCAM, Siff, CHECK, etc. El sistema COPS (Brin et al., 1995) se desarrolló en el marco del proyecto de librería digital de Stanford. Este realiza una comparación entre los documentos registrados a nivel de oración. Está principalmente enfocado a encontrar oraciones escritas exactamente igual, de modo que no permite solapamientos parciales, es decir, es sensible a la más mínima modificación del texto. A raíz de la herramienta COPS, surgió la herramienta SCAM para mejorar su eficacia (Shivakumar et al., 1995). SCAM utiliza las frecuencias de las palabras para detectar las copias, incluso es capaz de encontrar copias con solapamiento parcial pero aumenta la tasa de falsos positivos entre documentos que comparten vocabulario. Otra herramienta pensada para manejar grandes cantidades de documentos es Siff (Manber, 1994). Está basado en la herramienta *diff* de Unix que permite comparar dos documentos para encontrar diferencias a nivel de línea. Debido al coste computacional de *diff*, Siff crea una representación del documento en forma de huella digital y esta huella del documento será la que se comparará, reduciendo enormemente el tamaño de las comparaciones. La herramienta CHECK (Si et al., 1997) extrae información de la estructura y las palabras clave del documento para detectar las coincidencias entre dos documentos. La principal desventaja de este sistema es que solo se puede utilizar sobre documentos donde la información se encuentre estructurada. Otra herramienta a destacar es WCopyfind (Dreher, 2007) que permite detectar solapamientos entre seis o más palabras como unidad de comparación. Esta cuenta el número de palabras coincidentes entre oraciones de dos documentos y calcula un ratio de reutilización dividiendo entre el total de palabras del documento. El usuario es el encargado de considerar qué tamaño de solapamiento es el adecuado para determinar que dos oraciones/documentos están reutilizados.

Así como se ha comentado anteriormente, existe un tipo de reutilización más complejo y menos investigado: la reutilización translingüe. En Potthast et al. (2011) se proponen 3 aproximaciones distintas para abordarlo. La primera aproximación consiste en la técnica del análisis explícito translingüe (*Cross-Language Explicit Semantic Analysis*, CL-ESA), donde la comparación del documento fuente con 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. La segunda aproximación consiste en el alineamiento entre lenguajes basado en los principios estadísticos de la traducción automática (*Cross-Language Alignment-based Similarity Analysis*, CL-ASA). A partir de traducciones se genera un diccionario estadístico automático de todas las posibles traducciones de una palabra al otro idioma y, posteriormente, se comprueba si el documento fuente es una traducción del documento sospechoso. La tercera y última aproximación es la más sencilla de las tres y obtiene buenos resultados si las lenguas guardan cierta relación. Esta aproximación (*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 reutilización en código fuente

En esta sección se describe como se ha abordado el problema de la detección automática de la reutilización en código fuente. Primero nos centraremos en la detección de reutilización a nivel monolingüe y después abordaremos el problema a nivel translingüe. La detección automática de reutilización en códigos fuente se ha abordado mayoritariamente a nivel monolingüe como se reflejará en los dos siguientes apartados, debido a que es un fenómeno que ocurre en el ámbito académico, campo muy cercano al campo de la investigación. Por otra parte, entraña mayor dificultad la reutilización translingüe, tanto para ser cometida como para ser detectada. Sin embargo, en la figura 2.3 se observa que cada lenguaje de programación está influenciado por otros, y por lo tanto, contiene elementos (mejorados) de los lenguajes que los inspiraron. Esta influencia entre lenguajes permite un aprendizaje más fácil de nuevos lenguajes asociando los elementos comunes entre estos. Por otra parte, también facilita que un programador pueda cometer reutilización entre lenguajes de programación.

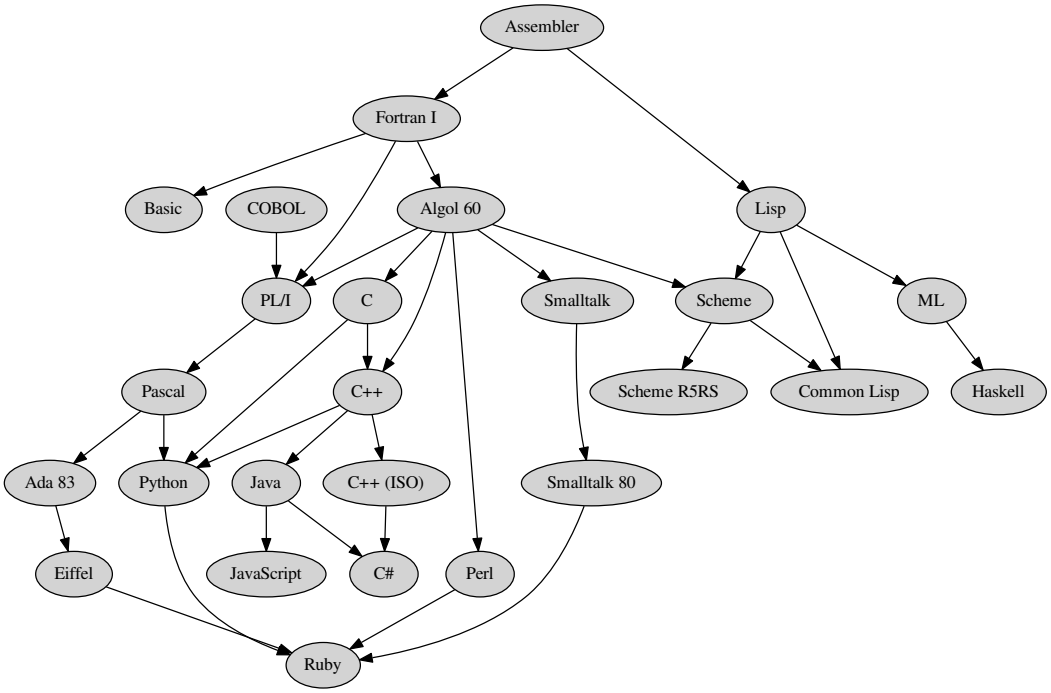


Figura 2.3: Árbol de las influencias de los principales lenguajes de programación durante la historia. Extraído de la Web Historia de los lenguajes de programación. URL: `http://www.levenez.com/lang/`.

2.2.1 Reutilización a nivel monolingüe

La reutilización se produce por el uso de un material externo como propio, con o sin reconocimiento del autor. En este contexto, cuando el material reutilizado consiste en código fuente, la detección de reutilización de código fuente es muy importante tanto a nivel académico como a nivel de industria del software.

Dentro de la industria del software se persigue preservar la propiedad intelectual. Las empresas invierten grandes sumas de dinero desarrollando software y, donde la legislación lo permite, protegen este software 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. Hay casos donde empresas y organismos se ven envueltos en litigios y acuerdos millonarios debido a la reutilización de código fuente no autorizada.[4] Sin embargo, la reutilización del software comercial no necesariamente tiene que ser considerada como plagio. Existen situaciones donde un código fuente se pone a disposición pública bajo licencias que protegen la propiedad de sus creaciones con propósitos no comerciales.[5]

En el mundo académico se evalúa la capacidad de programación de los alumnos a través del código fuente desarrollado como solución a un problema propuesto, pero en el caso de incurrir en reutilización esta destreza no se demuestra realmente. Es este motivo de interés en disponer de herramientas que ayuden al evaluador a detectar la reutilización efectuada por el alumno y, a su vez, disuadir del engaño. Este tipo de reutilización se puede cometer copiando tanto desde la Web como del código desarrollado por un compañero. En una encuesta académica (Cosma et al., 2008), la mayoría de los entrevistados está de acuerdo en que, cuando la reutilización está permitida, los estudiantes deben reconocer adecuadamente las partes del código fuente escritas por un tercero, sino, esta acción será considerada como plagio.

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 dificultando la tarea de detectar si están utilizando código fuente ajeno o no. Por estos motivos la detección de código fuente se limita a repositorios de código fuente locales o repositorios públicos de la red.

Además, existe el problema de 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 fuente para otras aplicaciones. Actualmente, se dispone de herramientas que realizan estas tareas automáticamente como son SandMark o BCEL[6] de la Apache Software Foundation. La ofuscación, en un principio, fue diseñada para evitar que se obtuviera información o diseños de un código fuente y de esta manera, dificultar

[4] `http://www.reuters.com/article/oracle-google-lawsuit-idUSL1N10A3A620150730`

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

[6] *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/`

su reutilización. Obviamente si un código fuente ha sido reutilizado y se han aplicado técnicas de ofuscación será mucho más difícil su detección.

Una herramienta interesante en forma de plugin del IDE (en inglés *Integrated Development Environment*) o entorno de desarrollo Eclipse[7] es ARTIFICE (Meyer, 2012). Esta herramienta realiza las siguientes transformaciones: *(i)* renombra variables, campos y métodos; *(ii)* expande expresiones (por ejemplo, expande la expresión $i{+}{+}$ en $i = i + 1$); *(iii)* contrae expresiones si es posible; y *(iv)* transforma bucles y expresiones condicionales por equivalentes (por ejemplo, transforma la expresión *if/else* en <E> ? <A> : <B>). En la figura 2.4 se muestra un ejemplo de transformación que realiza ARTIFICE.

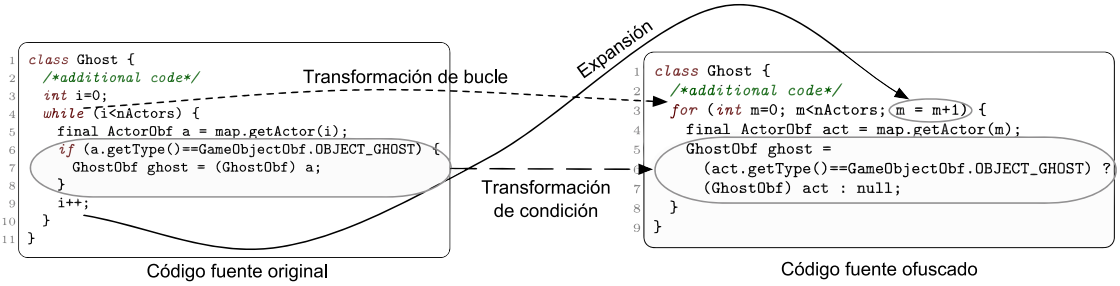


Figura 2.4: Ejemplo de tipos de ofuscaciones que realiza la herramienta ARTIFICE. Extraído del artículo Schulze et al. (2013).

Se producen multitud de casos donde, sin disponer del código fuente, se descubre su funcionalidad interna utilizando pruebas unitarias. A este proceso 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 qué está compuesto, cómo funciona 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[8] para ejecutar aplicaciones nativas de Microsoft Windows bajo la plataforma Linux/UNIX.

Respecto a las técnicas más empleadas por los estudiantes para dificultar el proceso de detección por parte del profesor existen varios estudios. Whale (1990) concluye en que los cambios más frecuentes realizados por los estudiantes son: cambios en comentarios, cambios de identificadores, cambios de orden de los operandos, cambio de tipos de datos, reemplazar expresiones por equivalentes, añadir sentencias redundantes, cambios de sentencias independientes del orden de ejecución, etc.

Faidhi et al. (1987) realizaron una clasificación de los tipos de modificaciones según su dificultad de ser modificado como se muestra en la figura 2.5. Estos niveles van desde la copia exacta hasta cambios de estructura general del código fuente. La tabla 2.6 muestra en un mismo código fuente ejemplos de los siete niveles de reutilización. En la sección 4.2 se explican más detalladamente estos niveles y cómo hemos abordado su detección. Consideramos que falta un nivel más que no ha sido

[7] `https://eclipse.org/`

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

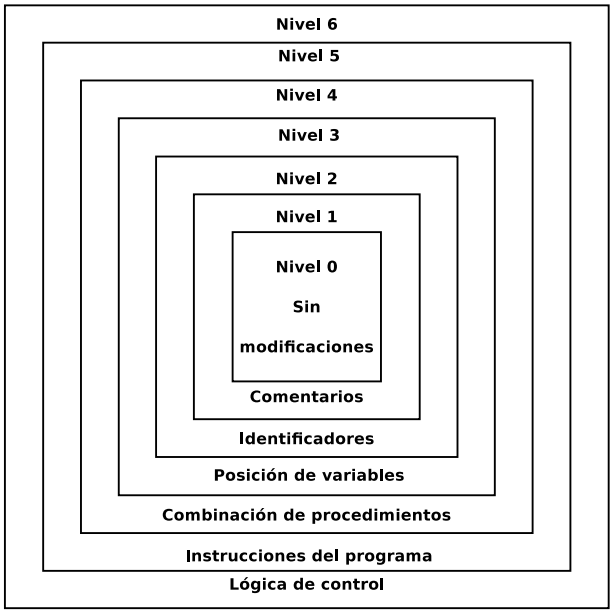


Figura 2.5: Niveles de reutilización propuestos en (Faidhi et al., 1987)

comentado en el trabajo Faidhi et al. (1987), y que es el principal objeto de estudio de esta tesis: la traducción de código entre diferentes lenguajes de programación.

Al igual que ocurre en textos escritos en lenguaje natural, se deberá distinguir entre análisis a nivel monolingüe, sobre el mismo lenguaje de programación, o translingüe, entre diferentes lenguajes de programación. Dentro del análisis de detección de reutilización en código fuente a nivel monolingüe destacan dos tendencias principales en la comunidad científica (Clough, 2000; Hislop, 1998). La primera de ellas consiste en la comparación de características del propio código fuente. Se consideran como características, entre otras, el número de líneas de un código fuente, el número de palabras y caracteres, el número de saltos de línea, el número de líneas con sangría, la cantidad y el tipos de los tokens. Por otra parte, la segunda tendencia, y más utilizada para la detección de reutilización de código fuente, ha sido la comparación de las estructuras del código. Se entiende por estructura de código la estructura sintáctica de este, 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 in-orden 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 en códigos fuente. 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

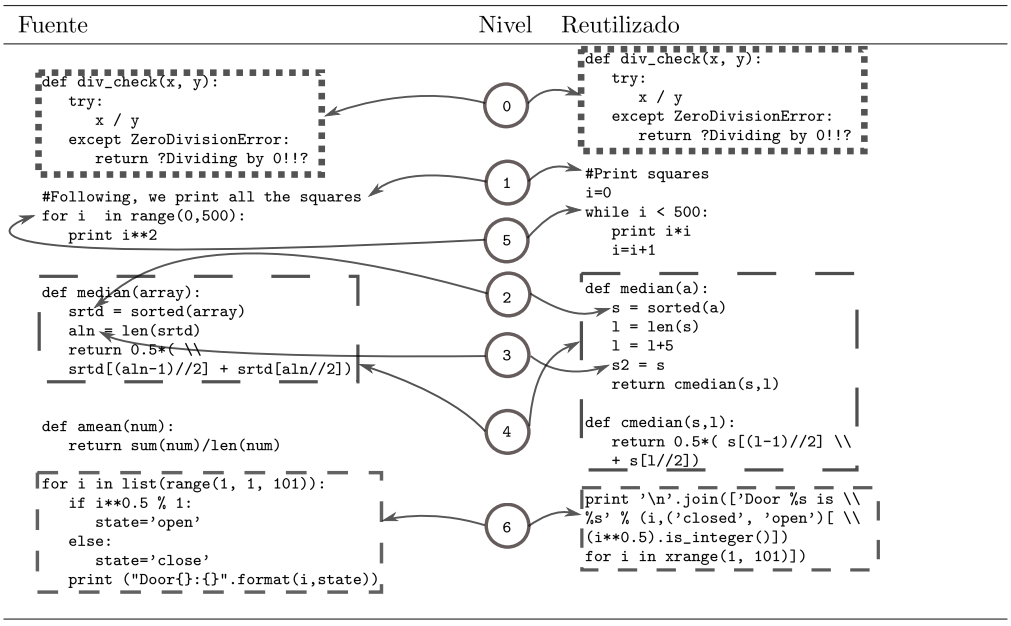


Figura 2.6: Ejemplos de los 7 niveles diferentes de modificaciones aplicadas en la reutilización de código fuente [niveles propuestos en Faidhi et al. (1987)]

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 bajo este paradigma.

El primer trabajo conocido en el ámbito de la detección de reutilización en código fuente es el propuesto por Halstead (1972), en el que trata de encontrar similitud entre códigos fuente a través de métricas tales 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 aplicada a algoritmos que realizan la misma tarea, obtenga resultados 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 sobre algoritmos en diferentes lenguajes (ADA, Pilot, Procedure) intentan verificar la eficacia de la fórmula definida pero tan solo se utilizan 14 muestras positivas, y ninguna muestra negativa. Además, estos algoritmos no han sufrido ningún tipo de modificación como los mencionados anteriormente para evitar la detección de reutilización.

Otro trabajo en la línea de comparación de características es el de Selby (1989). Selby propone distintos tipos de características a las utilizadas 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.

Un trabajo que se puede considerar tanto de comparación de características como de comparación de estructura es el de McCabe (1976). El planteamiento es el siguiente: se construye un grafo de un código fuente a partir de la estructura de ejecución, a continuación se obtiene una medida de complejidad del grafo que consiste en la suma de sus nodos, ejes y el número de caminos independientes que tiene el grafo desde el nodo inicio hasta el nodo final. En la misma línea que el trabajo de McCabe se encuentra el trabajo de Harrison et al. (1981). En este se propone una medida de complejidad para un código fuente en base al nivel de anidamiento durante la ejecución. Es importante destacar que estas dos medidas son altamente dependientes del compilador y , además, la segunda también del intérprete del código compilado.

Utilizando la comparación de estructuras, los sistemas de detección de reutilización 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. 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 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, así como funciones, transformando el código en otro equivalente para simplificar la estructura.

Jankowitz (1988) es uno de los primeros trabajos que propone utilizar la estructura del código para realizar la detección. Su estrategia se basa en construir el árbol de ejecución, realizar un recorrido en post-orden, 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 Wise (1992). YAP3 en una fase inicial genera una secuencia de tokens correspondiente a la estructura descartando comentarios, constantes, identificadores, convirtiendo el texto a minúsculas, 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) (Karp et al., 1987). Este algoritmo sirve para buscar la subcadena de longitud maximal común entre dos cadenas.

La herramienta más popular se llama JPlag[9], descrita en el trabajo de Prechelt et al. (2002). En ella también se realiza un preproceso ignorando los comentarios, identificadores y espacios en blanco. JPlag 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 y, además, se emplean tablas hash para realizar búsquedas con 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

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

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

Utilizando la herramienta JPlag internamente, los autores del trabajo Chunhui et al. (2013) abordan como prevenir la reutilización de código fuente en el ámbito académico desde dos puntos de vista: *(i)* educando a los estudiantes con ética y honestidad, y *(ii)* en caso de que el primer punto fallase, mediante la monitorización, detección y penalización de estos casos. Para evitar el plagio proponen una primera fase que consiste en un editor de código fuente que permite acceder a cada alumno mediante su propia credencial. Este editor tiene restringida la opción de copiar y pegar contenido desde el exterior de la aplicación a no copiar partes del código fuente ya escrito por el alumno. Además, guarda el código fuente de manera cifrada evitando que pueda ser reutilizado por otros alumnos. La segunda fase consiste en recolectar los códigos fuentes de los estudiantes y analizarlos con JPlag. Se considerarán casos de reutilización aquellos que estén entre 90-100 % de similitud, casos no reutilizados los que estén entre 0-10 % de similitud y susceptibles de ser analizados manualmente los que obtienen entre 10-90 % de similitud en JPlag. De 38 códigos analizados, se encuentran 10 como casos de reutilización, 15 como casos no reutilizados y 13 como posibles casos de reutilización. Después de la revisión manual de los 13 casos etiquetados como posibles, se considera que en total 18 han cometido plagio y 20 no.

Con la misma filosofía de encontrar alineamientos de fragmentos de código fuente se encuentra el trabajo Burrows et al. (2007). El sistema consiste en dos fases: una fase inicial que utilizando el índice invertido (véase el apartado 4.2 o Witten et al. (1999), pág. 109 para más información) y la medida de similitud Okapi BM25 (Robertson et al., 1992) actúa como un filtrado previo. En la segunda fase, una vez filtrada la colección, se realiza una búsqueda más intensiva entre los pares de códigos fuente mediante búsqueda de fragmentos idénticos. Demuestran mediante una comparación que son capaces de obtener resultados al mismo nivel que la herramienta JPlag en colecciones de códigos fuente de pequeño y gran tamaño, siendo el sistema altamente escalable para manipular grandes colecciones.

Basado en la filosofía de encontrar subcadenas comunes más largas se encuentra el trabajo de Baer et al. (2012). En concreto el patrón utilizado es el de los espacios en blanco, donde el código fuente se convierte en un único patrón reemplazando todos los caracteres visibles por la etiqueta "`X`" y los espacios en blanco por "`S`" dejando los saltos de línea tal cual aparecen. A continuación, se calcula un valor de similitud basado en la subcadena común más larga entre ambos códigos fuente. En la experimentación, Baer et al. utilizaron códigos fuente escritos en C extraídos del kernel de Linux. En ella demostraron que códigos fuente similares, obtenidos de distintas versiones del kernel, mostraron un alto promedio de similitud, mientras que, pares de códigos fuente diferentes obtuvieron un valor medio bajo en esta medida de similitud.

Otro trabajo en la línea de comparar estructuras sintácticas del código fuente es Xiong et al. (2009) donde se transforma código escrito en el lenguaje de programación C a un lenguaje intermedio CIL[10] que contiene pocas estructuras diferentes para construir un árbol y sin redundancias. Se genera un árbol sintáctico abstracto con el complemento CDT de la plataforma de desarrollo Eclipse. 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

[10] `http://sourceforge.net/projects/cil`

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 manipular 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 Rosales et al. (2008). 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 de subcadenas contiguas comunes más largas; *(ii)* longitud acumulada, localización de 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.

Siguiendo la línea de agrupar diversas medidas para la toma de la decisión de si un código fuente se considera como reutilizado, se encuentra la herramienta Sherlock (Joy et al., 1999). Esta herramienta compara cada par de códigos fuente de cinco maneras distintas para la decidir si es un caso de reutilización. Estas comparaciones se realizan buscando fragmentos continuos sin saltos dentro del código fuente considerando a este como una secuencia de textos. Los códigos fuente son comparados bajo las siguientes características: *(i)* el código fuente en su forma original, *(ii)* el código fuente con los espacios en blanco eliminados, *(iii)* el código fuente con los comentarios eliminados, *(iv)* el código fuente sin comentarios ni espacios en blancos, y *(v)* el código fuente sin delimitadores como pueden ser el paréntesis, corchete, coma o puntos y coma. Finalmente, como resultado se muestra un grafo cuyos nodos representan los códigos fuente y las aristas los casos de reutilización entre códigos.

La herramienta MOSS se propone en el trabajo Schleimer et al. (2003). Esta permite comparar 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. Otro trabajo basado en la técnica de Winnowing es el de Marinescu et al. (2013). En este artículo explican detalladamente la herramienta que utilizan para detectar plagio entre estudiantes de Ingeniería Informática (Marinescu et al., 2012). Para ello utilizan el algoritmo de Winnowing ignorando los espacios en blanco, comentarios, librerías cargadas, palabras reservadas operadores y delimitadores. Aunque actualmente soporta los lenguajes Java, C++ y C# es fácilmente extensible a otros lenguajes dándole como entrada a la aplicación los tokens que debe ignorar como son los comentarios, la carga de librerías, las palabras reservadas, etc. Al igual que JPlag, esta herramienta devuelve un listado ordenado por similitud entre pares de posibles casos de plagio y permite resaltar las secciones comunes detectadas entre dos códigos fuente. Aunque el algoritmo permite detectar fragmentos que han sido cambiados de posición e incluso donde se ha modificado el espaciado y tabulación, los cambios en identificadores afectan gravemente a la eficacia del sistema dado que es prácticamente lo único que se utiliza para identificar cadenas comunes.

Una técnica que ha resultado efectiva en detección de similitudes entre textos y que se ha aplicado a códigos fuente es la del análisis semántico latente (en inglés *Latent Semantic Analysis, LSA*)(Cosma

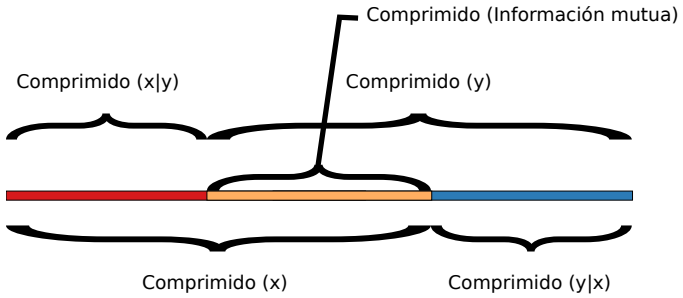


Figura 2.7: Representación de la información mutua dentro del espacio de compresión: dos cadenas (x e y) se representan por dos longitudes (Comprimido(x) y Comprimido(y)). Las cadenas comparten información mutua representada por comprimido(información mutua). Comprimido($x|y$) es más pequeña que Comprimido(x) porque existe una contribución de y.

et al., 2012). El trabajo de Cosma et al. está basado en tres fases: *(i)* preproceso: mantener o eliminar comentarios, y considerar o no considerar las palabras reservadas del lenguaje de programación; *(ii)* pesado: se miden las frecuencias de los tokens mediante la combinación de pesos locales (por ejemplo tf) y pesos globales (por ejemplo idf); y *(iii)* reducción de la dimensionalidad: empleando el análisis latente semántico (para conocer más detalles, véase el apartado 4.5) se reduce el espacio de dimensiones a un tamaño concreto. La experimentación se realizó dentro de un esquema de recuperación de información, es decir, dado un código fuente consulta, se recuperan los candidatos más probables a ser reutilización de este. Para construir el corpus emplearon los sistemas ya descritos anteriormente Sherlock y MOSS seguidos de un postprocesamiento manual. En el trabajo reportan como valor óptimo de número de dimensiones del espacio reducido de 30, suponiendo una reducción sustancial del espacio vectorial y por lo tanto del número de cálculos a realizar. El mayor inconveniente de este experimento es que se realizó sobre una colección de reducidas dimensiones. Un trabajo similar al anterior es el propuesto en Marcus et al. (2004) aprovechando que LSA reduce las dimensiones situando los conceptos/palabras/tokens relacionados muy próximos, tratan de buscar fragmentos semánticamente relacionados dentro del código fuente.

La propuesta de Zhang et al. (2007) 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 traducen 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) (Ziv et al., 1977) como una aproximación a la complejidad de Kolmorov, como refleja el estudio de Yang et al. (1998). De ambos diccionarios de compresión se obtiene la información que tienen en común como se muestra en la figura 2.7. Se realiza una comparación de este algoritmo con JPlag y MOSS con casos de plagio reales. Estos casos reales han sido manipulados artificialmente con posterioridad para ocultar la copia en cinco niveles de modificación según los niveles propuestos por Faidhi et al. (1987). 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.

Con el incremento del poder de cálculo por parte de los ordenadores, se empieza a trabajar en tipos de cálculos más complejos como los grafos. Por ejemplo, el trabajo de Krinke (2001) se basa en los grafos de dependencia del código fuente donde se buscan los subgrafos más similares y a continuación son mostrados al usuario en el código fuente original. Como el problema es NP-completo se implementa una aproximación con buenos resultados en un corpus muy pequeño y diverso por lo que es relativamente fácil distinguir qué partes de un código fuente se corresponden con las de otro. Cabe resaltar que con esta aproximación son necesarios 47 segundos para procesar un código fuente de 8 000 líneas por lo que es una aproximación bastante lenta. También cabe destacar que solo funciona a nivel monolingüe en el lenguaje C, por lo que su aplicación es muy restringida.

Otra aproximación basada en grafos es la de Mishne (2003). En este artículo se propone utilizar un concepto nuevo dentro del tratamiento de código fuente: los grafos conceptuales. El uso de grafos conceptuales en recuperación de información ha demostrado ser efectivo (Genest et al., 1997). Para aplicarlo a códigos fuente se utiliza el análisis que realiza el compilador, tanto a nivel sintáctico como a nivel semántico, para así construir el grafo conceptual. Como conceptos se utilizan las siguientes características: asignaciones, bloques, operadores de comparación, enumerados, llamadas a función, operaciones lógicas y matemáticas, cadenas, variables y bucles. Mientras que como relaciones se utilizan las relaciones siguientes: *condiciones_de, contiene_a, comentarios, define_a, depende_de, parámetros y devuelve.*

Mediante estos conceptos y relaciones, junto con la ayuda de un compilador, se construye el grafo conceptual de un código fuente. Para realizar la comparación se establecen manualmente los pesos y los tipos de relaciones que participaran en el valor de similitud final. Las relaciones directas, por ejemplo un bucle en el *código_1* y un bucle en el *código_2*, tienen un valor cercano a 1 mientras cualquier otra relación un valor de casi 0. Su experimentación es comparada con un modelo basado en la distancia de edición entre cadenas. El corpus para realizar búsquedas se compone del código fuente del compilador GCC. Toman 25 códigos fuente al azar y realizan un par de consultas: *(i)* la primera con el propio código fuente sin modificar, por lo que el primer resultado de la búsqueda debería ser el mismo código; y *(ii)* estos 25 códigos modificando identificadores, cambiando funciones por otras equivalentes, comentarios y cadenas. Comparan el sistema de grafos conceptuales con búsqueda con distancia de edición, búsqueda por tipos (usando *wildcards*) y la medida Okapi BM25. Mientras que los modelos basados en uso de cadenas funcionan mejor con el código fuente idéntico (valor de similitud 1 frente a 0,9 con grafos conceptuales), con los códigos modificados la similitud de los modelos de cadenas disminuye notablemente (0,30 frente a 0,81 del modelo de grafos). El modelo basado en la medida Okapi BM25 muestra buenos resultados en el caso de búsqueda con códigos idénticos y cercanos al modelo de grafos en los códigos modificados. Por ello, realizan un último experimento combinando ambos modelos con diferentes pesos, obteniendo el mejor resultado usándolos a partes iguales (0,5× el valor de similitud de cada modelo). A la vista del comportamiento de ambos sistemas donde para la construcción del grafo es necesario utilizar un compilador, además del alto número de comparaciones y el coste de comparar grafos enteros[11], se concluye que no es esta aproximación una opción válida para la recuperación de código fuente debido a que Okapi muestra buenas prestaciones en un tiempo muy reducido utilizando los valores predeterminados para el inglés (no se ha entrenado con código fuente).

[11] El coste computacional es cúbico con el tamaño del grafo.

Siguiendo con la representación estructural del código fuente, en el trabajo de Feng et al. (2013) se comparan cuatro aproximaciones distintas: basadas en texto, en tokens, en árboles y en semántica. En el artículo proponen un sistema basado en el árbol sintáctico generado a través del compilador. Para ello, construyen inicialmente el árbol sintáctico, a continuación guardan un vector de nodos ordenado según en número de sub-nodos. Además cada nodo se representa como el valor de la función hash de sus nodos hijos. De esta forma, al comparar dos códigos, solamente se comparan aquellos que tengan el mismo número de sub-nodos para realizar una comparación eficiente (comparar el árbol sintáctico completo de dos códigos puede suponer un alto coste de cálculo en casos con cientos de miles de códigos).

Los autores comparan distintos escenarios de reutilización de código fuente: renombrar variables, renombrar identificadores, renombrar tipos de datos (int, $float$...), reordenar datos independientes, reordenar datos dependientes). También se realiza una comparación con detectores de clones. Debido a que la funcionalidad de los detectores de clones no es la de detectar reutilización, se ven más afectadas en estos escenarios que las herramientas diseñadas específicamente para la detección de reutilización.

El uso de árboles sintácticos permite detectar muy bien las modificaciones típicas para evitar su detección como son el renombrar partes del código (identificadores, tipos...) o reordenar parte de la ejecución de código fuente. Por otra parte, se ven altamente afectados por cambios del código fuente por operaciones/funciones equivalentes y la inserción de funciones vacías que modificarían el número de sub-nodos que tiene cada nodo. Además este tipo de sistemas al requerir el uso de un compilador para la extracción del árbol sintáctico, es altamente dependiente del lenguaje de programación.

Otra aproximación basada en la estructura del código fuente, en concreto en la secuencia de llamadas de las funciones (en inglés *control flow graph*), es el trabajo (Chae et al., 2013). El sistema propuesto consiste en buscar subgrafos comunes entre dos códigos fuente representados como un grafo de dependencias de llamadas. La búsqueda de subgrafos es un problema computacionalmente complejo por lo que la aplicación de este modelo resulta muy costosa para colecciones de códigos fuente de mediano y gran tamaño. El sistema se presenta como altamente efectivo, sin embargo, el corpus sobre el que se ha realizado presenta casos de reutilización aparentemente trivial. En concreto, se han tomado de 28 proyectos software, dos versiones sobre las que se supone que existen diferencias. La variabilidad temática de los proyectos software va desde proyectos de mensajería instantánea (Pidgin) hasta reproductores musicales (Foobar2000). Se considera que los casos de reutilización son triviales debido a la prácticamente nula relación entre proyectos.

Una aproximación que trata de reducir el alto coste computacional de trabajar con el árbol sintáctico del código fuente es el trabajo de Cui et al. (2010). En este se describe la herramienta de detección de plagio en código CCS (en inglés *Code Comparison System*) que compara los códigos fuente a través de utilizar valores hash de los árboles sintácticos. En consecuencia, CCS realiza una comparación de códigos fuente de la siguiente manera: *(i)* primero realiza un recorrido transversal del árbol sintáctico y obtiene el valor de la función hash de cada subárbol de ambos códigos fuente; *(ii)* los subárboles son separados según el número de nodos inferiores para su comparación; y *(iii)* aquellos subárboles con el mismo valor de función hash representan los fragmentos de códigos fuente considerados como reutilización.

Campos relacionados

Otro campo relacionado con la detección de reutilización es el de atribución de autoría. En este campo se trata de identificar quién es el autor de un determinado código fuente entre una selección de autores. La herramienta SCAP fue utilizada para detección de autoría en textos (Kešelj et al., 2003) y posteriormente se ha aplicado sobre códigos fuente (Frantzeskou et al., 2008). El método consiste en dividir en n-gramas todo el código fuente considerando todos los caracteres incluso espacios, saltos de línea, etc. Posteriormente se ordenan según frecuencia y se obtienen los L más frecuentes. Estos L n-gramas se consideran el perfil del autor del código fuente y es lo que se utiliza para realizar la atribución. De cada autor se han concatenado todos sus códigos fuente disponibles y se ha creado un perfil como el mencionado anteriormente. Se determina que un código fuente pertenece a un autor por el número de n-gramas en común entre el perfil del código fuente y el perfil del autor ($perfil\_código\_sospechoso \cap perfil\_autor$) sin considerar las frecuencias de los n-gramas. Este sistema es sencillo, rápido e independiente del lenguaje.

Un buen ejemplo para demostrar la cercanía entre la atribución de autoría con la detección de reutilización es el trabajo de Bandara et al. (2012). En este trabajo los autores enfocan un método para atribución de autoría en código para la detección de plagio. Para ello se hacen valer de tres algoritmos distintos: Naive Bayes, k-NN y AdaBoost. Como entrada del sistema se reciben atributos o características del código fuente. Estos atributos son tales como el número de caracteres por línea, palabras por línea, frecuencia de tipo de acceso (por ejemplo, *public, protected, default, private*), frecuencia de tipo de comentario (‘//’, ‘/**/’), longitud del identificador o número de veces que aparece el caracter ‘_’ en los identificadores. Para entrenar y validar su sistema, Bandara et al. utilizan un corpus preparado para atribución de autoría escrito en Java (Lange et al., 2007) que consiste en 10 autores (1 645 códigos fuente en total) obteniendo un acierto del 86,64 % frente al 55,0 % de obtenido en Lange et al. (2007). Bandara et al. remarcan que su sistema no funcionaría correctamente en el caso de que programadores siguieran un estándar de programación y reglas de formato en sus proyectos por el tipo de características empleadas en la detección. En un escenario de reutilización donde los programadores tratan de modificar el estilo del código fuente original, por lo tanto su rendimiento se vería afectado notablemente. Este sistema es dependiente del lenguaje de programación ya que necesita identificar comentarios, distinguir entre tipos de las variables, distinguir los tipos de acceso (*public, private...*) y reconocer los identificadores. La aplicación de este sistema se encuentra restringida a nivel monolingüe.

Un campo estrechamente relacionado con la detección de reutilización en código fuente es el de detección de clones (Koschke, 2007). Los clones en ingeniería del software son segmentos de código que son similares en base a una definición de similitud. Esta similitud puede ser textual, léxica, sintáctica o semántica. Similitud semántica consiste en que un segmento de código tenga pre-condiciones y post-condiciones similares. Algunos autores identifican que 7-23 % del código se encuentra duplicado; llegando en casos extremos al 59 %. En la figura 2.8 se muestra un ejemplo del funcionamiento de la detección de clones y su resultado.

Normalmente se encuentran distintos tipos de clones: fragmentos de código fuente copiados exactamente, fragmentos con sintaxis idéntica pero con cambios de variables, tipos e identificadores o con inclusión o exclusión de instrucciones. El comportamiento común de un programador es el de utilizar un código fuente copiado como plantilla y a partir de ese momento personalizarlo al

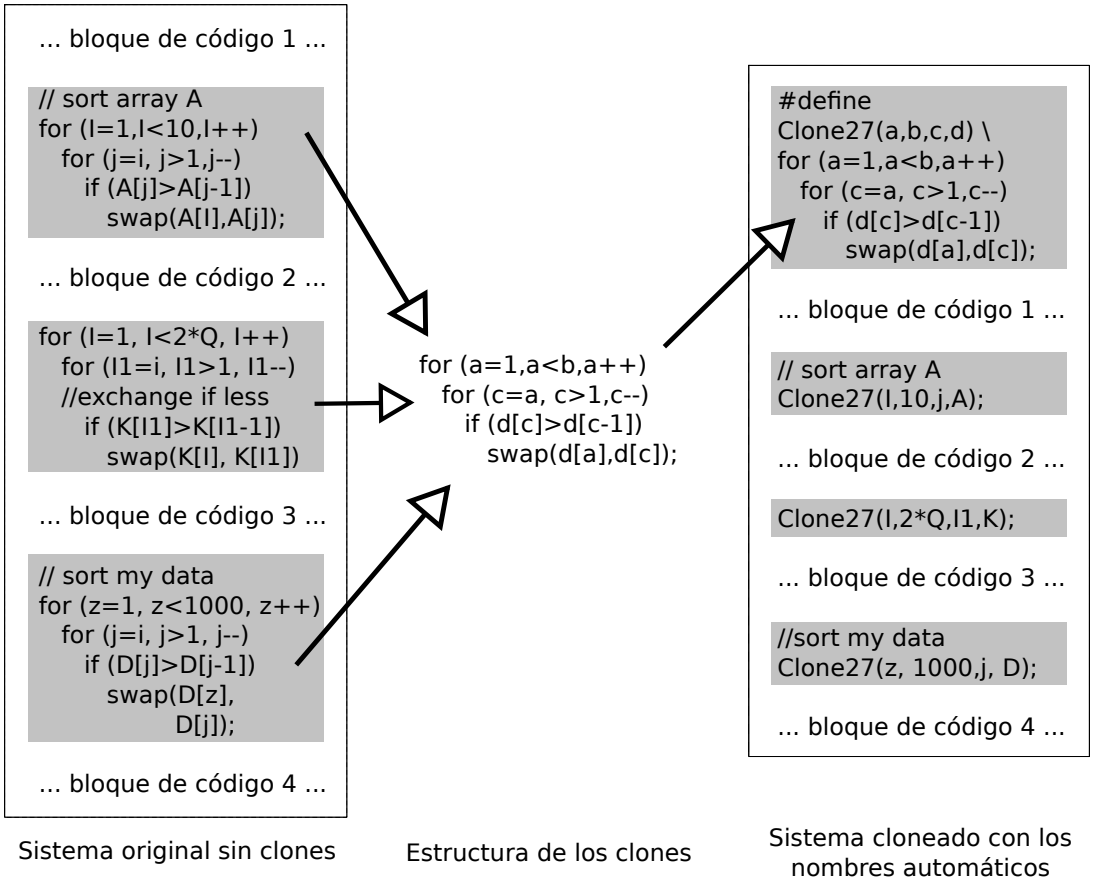


Figura 2.8: Ejemplo de detección y sustitución de clones en código fuente. Ilustración adaptada de la web Semantic Designs. URL: `http://www.semdesigns.com/products/clone/`.

contexto donde se ha copiado. Se consideran como tipos de clonación la bifurcación[12] (en inglés *fork*), uso de plantillas[13] (en inglés *templating*) y la personalización (en inglés *customization*) que ocurre cuando un código fuente existente no se adecúa a los requerimientos del nuevo problema y necesita ciertas modificaciones. La aplicación principal de un detector de clones es la de refactorizar el código fuente y así reducir el número de errores cometidos por el programador al repetir fragmentos de código. Así pues, con la refactorización se consigue una mejora del diseño del código fuente existente separando la parte funcional de la parte estructural preservando la funcionalidad y eliminando duplicidades en el código fuente. Cabe destacar que el proceso de refactorización es bidireccional, es decir, permite realizar modificaciones tanto para descomponer el código fuente por funcionalidad como para clarificar la funcionalidad del código fuente. Durante el proceso de refactorización se requiere un gran conocimiento a nivel de compilación por parte del sistema. Por una parte se encuentran los *conocimientos sintácticos*: se debe poder distinguir entre una variable y

[12] `https://es.wikipedia.org/wiki/Bifurcación_(desarrollo_de_software)`
[13] `https://en.wikipedia.org/wiki/Template_metaprogramming`

una función que tengan el mismo nombre. Por otra parte están los *conocimientos semánticos* donde se debe distinguir entre variables con el mismo nombre pero en entornos diferentes, un ejemplo muy común es la variable *i* muy utilizada para bucles (Baxter et al., 1998). Además también se encuentran los *conocimientos de dependencias*, es importante conocer si una variable se modifica durante un fragmento de código modificando su valor (esto se analiza en el grafo de dependencias, en inglés *dependency graph*). Un proceso típico de refactorización realizado por el programador es el comúnmente llamado en el área de la programación *boilerplate*[14]. Este consiste en fragmentos del código fuente que se incluyen en muchas parte del código fuente sin apenas modificaciones. Por ejemplo, en la programación orientada a objetos, los métodos para obtener y modificar las variables de una clase (*get* y *set*). Según el estudio Velez et al. (2015), solo un 5 % del código fuente escrito en Java captura el núcleo de la funcionalidad del código fuente.

Una aplicación de la detección de clones es buscar, entre distintas versiones de un código, el origen de cierto error (en inglés *origin analysis*) (Tu, 2002). El mismo problema surge cuando se comparan o combinan distintas versiones de un proyecto software (Hunt et al., 2002). La detección de reutilización de código fuente se enfrenta a problemas similares, pero de mayor dificultad que la detección de clones. En los casos de reutilización, el programador trata de camuflar su copia, lo que hace que la reutilización sea más difícil de detectar. Para utilizar detectores de clones clásicos con el objetivo de detectar casos de reutilización sería necesario convertir previamente los programas a una forma canónica. Esta conversión, por otra parte, daría lugar a falsos positivos.

2.2.2 Reutilización a nivel translingüe

En el apartado 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. La reutilización translingüe no se ha estudiado con tanta profundidad como a nivel monolingüe tanto por la mayor dificultad que implica traducir de un lenguaje de programación a otro como por la dificultad que entraña su detección.

Los lenguajes de programación tienen un vocabulario más corto que los textos en lenguaje natural, y es frecuente que tengan muchos elementos en común o equivalentes. Además, la facilidad de un programador para utilizar y entender un nuevo lenguaje posibilita la reutilización entre lenguajes de programación. 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[15]. Este 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.[16] 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.

[14] `https://en.wikipedia.org/wiki/Boilerplate_code`
[15] `http://tangiblesoftwaresolutions.com/`
[16] `http://www.developerfusion.com/tools/`

El primer trabajo conocido que detecta similitudes entre códigos fuente escritos en diferentes lenguajes de programación es Halstead (1972) y Halstead (1977). Este trabajo, que ya ha sido comentado en el apartado anterior, fue uno de los pioneros en la detección de reutilización en código fuente. Halstead trata de encontrar una fórmula común para calcular la similitud entre las distintas implementaciones de un algoritmo en varios lenguajes de programación: ADA, Pilot y Procedure. En concreto define el "volumen" de un código fuente según la fórmula siguiente:

$$V = (N_1 + N_2)log_2(n_1 + n_2) \tag{2.3}$$

siendo n_1 y n_2 el número de ocurrencias de operadores y operandos y N_1 y N_2 el número de operadores y operandos distintos.

La segunda aproximación conocida es Arwin et al. (2006). De este trabajo se deriva una herramienta llamada XPlag que permite detectar reutilización entre múltiples lenguajes utilizando código intermedio generado por un compilador. Para ello se necesita un compilador que permita más de un lenguaje. En este caso se utiliza el compilador *GNU Compiler Collection* (GCC)[17] que soporta los lenguajes C, C++, Java, Fortran y Objective C, aunque en este estudio solo 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. Solamente se generan 10 casos simulados de reutilización translingüe utilizando el traductor Jazillian de código fuente de C a Java, el cual ya 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 (Robertson et al., 1999). Esta medida de similitud es altamente efectiva en búsqueda general de texto. Okapi BM25 extiende la aproximación *idf* y *tf*, que se explica en la sección 4.2. La ecuación 2.6 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.

$$\alpha_{t,d} = \frac{(k_1 + 1)tf_{t,d}}{k_1((1 - b) + b \cdot \frac{|d|}{L_{avg}}) + tf_{t,d}} \tag{2.4}$$

$$\beta_{t,d_q} = \frac{(k_3 + 1)tf_{t,d_q}}{k_3 + tf_{t,d_q}} \tag{2.5}$$

$$\sum_{t \in d_q} idf_t \cdot \alpha_{t,d} \cdot \beta_{t,d_q} \tag{2.6}$$

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

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

un compilador común a todos los 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.

En el trabajo descrito en Narayanan et al. (2012) se propone una medida de similitud que utiliza un particular esquema de pesado combinando diferentes características del código fuente. Es capaz de manipular y comparar códigos fuente escritos en los lenguajes de programación C, C++, C# y Java. En la primera fase, la de preproceso, se eliminan los comentarios y los espacios en blanco. En la fase de tokenización, se genera una firma de las palabras reservadas y los identificadores que tienen en cuenta valores como la frecuencia de aparición. También se tiene en cuenta valores como número de líneas tras el preproceso, número de caracteres y número de datos de entrada y salida requeridos. Todos estos valores forman la "huella" del código fuente para su comparación. Utilizando los valores calculados de las huellas de dos códigos fuente y utilizando la medida de complejidad propuesta en Halstead (1972) obtienen el valor de similitud entre dos códigos fuente. La gran limitación de este trabajo es que los resultados no son concluyentes porque el conjunto de datos en los que se ha realizado la experimentación no es suficientemente grande, en concreto se experimenta en dos colecciones de 20 y 30 códigos fuente.

Un trabajo de actualidad sobre detección translingüe de reutilización en códigos fuente es el realizado en Pramono et al. (2014). En este se inspecciona un caso de estudio ya previamente conocido como caso de plagio entre lenguajes de programación en un juego. En concreto, una aplicación original escrita para la plataforma Symbian OS[18] fue "portada" ilegalmente a la plataforma Android[19]. Ambas plataformas utilizan una versión modificada del lenguaje Java por lo que la reutilización manual de una plataforma a otra es relativamente sencilla. Para la detección en el caso de estudio hacen uso de dos aproximaciones distintas, una aproximación textual y una aproximación estructural. En la aproximación textual emplean palabras clave del código fuente original como *jump* o *hero* entre otras. De estas palabras clave analizan la frecuencia y la dispersión en ambas aplicaciones comprobando que es altamente similar. En la aproximación estructural se centran en la división por paquetes de las aplicaciones encontrando una estructura de tres fases muy similar en ambas aplicaciones: menús, juego y resultado. Por ambas aproximaciones se han mostrado evidencias de que el caso de estudio contiene reutilización ilegal a nivel translingüe.

2.3 Conclusiones

En este capítulo se ha mostrado una visión general de la detección de reutilización, prestando especial atención en la distinción entre plagio y reutilización. Se ha hecho una revisión de los principales trabajos sobre la detección de reutilización en textos que ha sido el punto de arranque de esta tesis y se ha mostrado el estado de la cuestión de la detección de reutilización en código fuente tanto a nivel monolingüe como a nivel translingüe. 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 describirán los recursos desarrollados en el marco del doctorado y que han sido objeto de estudio de reutilización de código fuente monolingüe y translingüe.

[18] `http://licensing.symbian.org/`
[19] `https://www.android.com/`

Capítulo 3

Recursos

En general, resulta complicado disponer de recursos que contengan casos reales de reutilización debido a los problemas de confidencialidad que esto acarrea. Esta dificultad impide la investigación y mejora de modelos que permitan la detección de reutilización automática. La construcción de colecciones con casos reales de reutilización: *(i)* ayuda a estimular la investigación en detección automática de reutilización; *(ii)* permite la comparación de distintas aproximaciones; *(iii)* ayuda a entender mejor la reutilización; y *(iv)* enseña a los estudiantes a hacer un uso apropiado del material de terceros (Clough, 2003).

Crear un recurso de estas características supone un alto coste. No se trata de una simple colección de códigos fuente sino que requiere un procesado por parte de expertos programadores para depurar errores, anonimizar y etiquetar cada par de códigos fuente como plagiados o no. Además, el etiquetado debe realizarse por parte de varios anotadores para conseguir unos juicios de relevancia del corpus confiables en función del acuerdo alcanzado. Incluso en algunos casos se realizan traducciones para simular casos de reutilización translingüe.

En este trabajo de investigación se han creado recursos para avanzar y promover la investigación en la detección automática de reutilización de código fuente a través de: *(i)* corpus reales con casos de reutilización de código fuente; *(ii)* corpus con casos simulados de reutilización; y *(iii)* corpus de códigos fuente similares obtenidos en repositorios de la Web.

El escenario de corpus reales con casos de reutilización se ha creado a partir de varios corpus procedentes de trabajos académicos para los cuales ha sido necesario un consentimiento previo para ser utilizados con fines científicos. Estos corpus están descritos en el apartado 3.1.

Los casos de reutilización simulados se han creado a partir de traductores de código fuente. Mediante el uso de este tipo de herramientas se permite simular otro tipo de reutilización que hasta ahora ha sido escasamente estudiado: la reutilización de códigos fuente escritos en diferentes lenguajes de programación. Estos casos simulados han sido creados en los corpus descritos en los apartados 3.1.3 y 3.2.2.

Uno de los corpus obtenidos a través de la Web ha sido extraído de una competición de programación por su alto grado de similitud con los cursos masivos producidos en el escenario académico, además de permitir estudiar la reutilización en un volumen elevado de códigos fuente que no se ha considerado anteriormente. También se ha creado un corpus a partir de un repositorio Web donde se recogen implementaciones distintas de un mismo problema en una gran cantidad de lenguajes de programación. Este tipo de recurso es muy útil para evaluar tanto las diferencias entre lenguajes de programación como la posibilidad de encontrar soluciones alternativas a un mismo problema a través de su similitud. Estos dos recursos se encuentran descritos en los apartados 3.2.1 y 3.2.2 y forman parte de los trabajos de investigación Flores et al. (2015c) y Flores et al. (2015b) respectivamente.

3.1 Recursos académicos

Uno de los entornos más propicios para la reutilización de código fuente es el académico donde un grupo de estudiantes debe resolver el mismo problema. La necesidad de evaluar a cada estudiante y la dificultad de comparar todos los trabajos académicos en busca de evidencias de copia hace que la detección manual de reutilización de código fuente sea impracticable por parte del profesor. En este escenario surge la necesidad de herramientas de detección automática de reutilización que den soporte al profesor. Por ello, en este apartado se recogen tres corpus con casos de copia en códigos fuente que permiten comprobar y evaluar modelos de detección automática.

A continuación se presentan tres corpus recopilados del ámbito académico. El primero de ellos, el corpus SPADE, es un corpus translingüe que proviene de una plataforma con implementaciones en diferentes lenguajes de programación. Los otros dos corpus, ILN y A&T++, nacen a partir de casos reales de copias detectadas entre estudiantes de cursos universitarios.

3.1.1 Corpus SPADE

La plataforma SPADE[1] (del inglés *Smart Python multi-Agent Development Environment*) permite desarrollar sistemas multiagente u organizaciones a través de la tecnología XMPP/Jabber. En el marco de la asignatura Sistemas Multiagente del Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital, del Departamento de Sistemas Informáticos y Computación de la Universitat Politècnica de València, se desarrolló la API (del inglés *Application Programming Interface*) para esta plataforma en los lenguajes de programación Java y C++ basándose en la ya escrita en Python.

En la tabla 3.1 se muestran algunas estadísticas del corpus donde el conjunto de códigos escritos en el lenguaje Python consiste en 4 códigos fuente con 10 503 tokens. Estos códigos están escritos por los colaboradores del proyecto e implementan la funcionalidad completa de la plataforma. Los códigos escritos en el lenguaje C++ consisten en 5 códigos fuente con un total de 1 318 tokens. Su implementación trata de mantener la estructura original de los códigos escritos en Python salvo un código fuente Python que se ha representado en dos códigos C++. La tercera parte del corpus escrita en Java consiste en 4 códigos fuente con 1 100 tokens. La diferencia en número de tokens de los códigos fuentes escritos en C++ y Java respecto a Python se debe a que la parte de la API

[1] https://code.google.com/p/spade2/

Lenguaje	Python	C++	Java
Tokens	10 503	1 318	1 100
Longitud media de los tokens	3,24	3,46	4,52
Tipos	671	144	190
Número de códigos fuente	4	5	4
Tipos por código fuente	167,75	28,8	47,5

Tabla 3.1: Estadísticas del corpus SPADE.

implementada en estos dos lenguajes solo corresponde a una parte de su funcionalidad. Concretamente, el núcleo del servidor que controla la comunicación entre agentes se mantiene en Python, mientras que la lógica de cada agente se ha portado a los otros dos lenguajes de programación.

Por lo tanto, existe reutilización parcial entre los códigos escritos en C++ y Java debido a que ambos lenguajes realizan la misma tarea que los códigos escritos originalmente en Python. Los casos Python→C++ y Python→Java representan un escenario real de reutilización translingüe, mientras que C++—Java representa la reutilización triangular entre los tres lenguajes teniendo Python como pivote.

3.1.2 Corpus ILN

Este corpus está compuesto por una colección de 183 programas escritos en el lenguaje Python entregados por estudiantes de la asignatura Ingeniería del Lenguaje Natural del Máster Universitario en Ingeniería y Tecnología de Sistemas Software del Departamento de Sistemas Informáticos y Computación de la Universitat Politècnica de València. Cada programa corresponde a la solución de una de las 5 tareas propuestas por los profesores de la asignatura con carácter individual o grupal, siendo los problemas 2 y 4 individuales y el resto puede desarrollarse en grupos. En la tabla 3.2 se muestran algunas estadísticas del corpus.

Tarea	Núm. de códigos	Tokens	Longitud promedio	Tipos	Pares reutilizados
1	30	5 101	11,416	1 390	4
2	50	7 892	7,642	1 872	5
3	29	5636	6,899	948	6
4	47	8 932	5,305	1 028	4
5	27	5 879	7,625	976	7

Tabla 3.2: Estadísticas del corpus ILN.

Sobre los 26 pares con evidencias de reutilización se ha realizado una comprobación manual con el objeto de estudiar los tipos de modificaciones realizados sobre los códigos fuente por los alumnos para dificultar su detección. En la tabla 3.3 se resumen los cambios encontrados. Cabe destacar que un mismo caso de reutilización puede contener varios tipos de modificaciones en fragmentos distintos del código fuente. Como recurso más habitual se encuentra la copia exacta y cambios en los identificadores, correspondiendo ambos tipos de modificaciones a los dos primeros niveles de Faidhi (véase figura 2.5).

Tipo de modificación	Cantidad
Copia exacta	11
Cambio de identificadores	11
Cambio de posición	3
Eliminar parte del código	3
Cambio de comentarios y cadenas	1
Combinar/desglosar funciones	1
Traducción de variables	1
Cambios en instrucciones	1

Tabla 3.3: Tipos de modificaciones encontradas en el corpus ILN.

3.1.3 Corpus A&T++

Este corpus nace a partir de la detección manual de casos reales de plagio entre estudiantes en el curso *Secure Electronic Commerce* del segundo semestre del año 2003 en la *School of Computer Science and Information Technology* del *Royal Melbourne Institute of Technology* en Melbourne, Australia facilitado por los autores del trabajo Arwin et al. (2006). En concreto, se trata de dos colecciones escritas en los lenguajes de programación C y Java, con 79 y 259 códigos fuente respectivamente.

Lenguaje	Núm. de códigos	Tokens	Long. promedio	Tipos	Pares reut.	Acuerdo
C	79	13 006	7,649	2 553	26	0,480
Java	259	83 755	7,531	9 647	84	0,668

Tabla 3.4: Estadísticas del corpus A&T++.

En la tabla 3.4 se muestran los detalles del corpus, además del resultado de la anotación manual del corpus y el acuerdo entre anotadores. Ambas colecciones fueron preprocesadas para anonimizar los trabajos de los estudiantes. La colección escrita en Java presenta una mayor cantidad de códigos fuente aunque el porcentaje de pares de códigos fuente reutilizados detectados en ambos casos es similar, alrededor de un 32 %.

Estos datos se han obtenido mediante la revisión manual de todos los pares de códigos fuente para establecer unos juicios de relevancia (*gold standard*) fiables para futuros experimentos. Hemos llevado a cabo el proceso de anotación de los casos de reutilización por tres revisores de forma independiente. Así pues, el conjunto final de casos de reutilización viene dado por aquellos pares de códigos fuente que han sido etiquetados por dos o más revisores. Para la colección escrita en el lenguaje de programación C, el *gold standard* final lo constituyen 26 pares de códigos fuente que se consideran reutilizados porque al menos dos de tres revisores así lo han determinado. Este mismo proceso se ha llevado a cabo sobre la colección escrita en el lenguaje Java determinando un total de 84 pares de códigos fuente reutilizados.

Para cuantificar el grado de acuerdo entre anotadores se ha utilizado la medida *Fleiss' kappa* (Fleiss, 1971). Estos valores están en el intervalo $[0-1]$ siendo 0 un acuerdo pobre y 1 un acuerdo perfecto. Para interpretar los valores de κ resultantes del cálculo del acuerdo se toman los valores de la tabla 3.5 propuestos en un estudio sobre las medidas de acuerdo (Landis et al., 1977). Como se puede observar, para nuestra colección de códigos fuente reutilizados escritos en C se ha obtenido

un acuerdo moderado mientras que para la colección de códigos escritos en Java se ha conseguido un acuerdo sustancial.

κ	Interpretación
<0	Acuerdo pobre
0,01 — 0,20	Acuerdo leve
0,21 — 0,40	Acuerdo equitativo
0,41 — 0,60	Acuerdo moderado
0,61 — 0,80	Acuerdo sustancial
0,81 — 1,00	Acuerdo casi perfecto

Tabla 3.5: Acuerdo entre anotadores por rangos de valores de κ.

Por otra parte, como parte de este trabajo de investigación hemos enriquecido el corpus con unas muestras para experimentación a nivel translingüe. Se han traducido los 79 códigos escritos en C al lenguaje Java utilizando el traductor de código fuente *C++ to Java Converter* con número de versión 2.7 de la empresa *Tangible Software Solutions Inc.*[2] Así pues, se consideran casos de reutilización translingüe a los pares traducidos del lenguaje C al lenguaje Java (si A.c ha sido traducido a Java como A.java, <A.c, A.java> es un caso de reutilización) y a los casos propagados de la colección de códigos fuente escrita en el lenguaje C (si el par <A.c, B.c> era un caso de reutilización, se consideran <A.c, B.java> y <B.c, A.java> como casos de reutilización). En total se consideran 131 pares de códigos fuente con reutilización translingüe (79 traducciones + 52 pares propagados de la colección original).

3.2 Recursos en la Web

En este trabajo de investigación hemos creado dos recursos de gran tamaño cuyo origen es la Web. El primer recurso proviene de la competición de programación online Google Code Jam, cuyo desarrollo y objetivo son similares a los de cursos masivos de programación online dentro del marco académico. Su desarrollo consiste en que un colectivo numeroso de programadores tiene que resolver los mismos problemas, y su objetivo es evaluar los resultados de este colectivo. El segundo, Rosettacode.org, es un recurso translingüe extraído de un repositorio de códigos fuente que resuelven problemas conocidos implementados en una amplia cantidad de lenguajes de programación, cuyo objetivo es ayudar en el aprendizaje de estos lenguajes a través de los distintos ejemplos.

Ambos corpus son de gran volumen y con implementaciones en distintos lenguajes de programación. La diferencia fundamental entre estos dos recursos es la proporción de implementaciones respecto al número de problemas. El corpus Google Code Jam consiste en pocos problemas pero con gran cantidad de implementaciones de cada uno de ellos, mientras que Rosettacode.org tiene una alta cantidad de problemas con pocas implementaciones de cada uno de ellos.

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

3.2.1 Corpus Google Code Jam

En la edición de 2012 de la competición Google Code Jam, participaron más de 20 000 programadores pertenecientes a 149 regiones del mundo y que utilizaron alguno de los más de 70 lenguajes de programación de la competición.[3] El objetivo de la competición es alcanzar la ronda final a través de distintas rondas previas cumpliendo los requisitos de cada una de ellas. En las primeras rondas el participante lo puede hacer desde cualquier lugar y tiene tiempo suficiente para poder resolver todos los problemas. Por ejemplo, en la ronda inicial (*Qualification round*) se dispone de 25 horas. En esta ronda es donde se concentra la mayor cantidad de participantes y aquellos que no alcanzan un número determinado de puntos no pueden acceder a la siguiente ronda. Por lo tanto, también es en esa ronda donde hay una mayor facilidad y cantidad de tiempo para poder colaborar entre participantes. En la *Qualification round*, se proponen 4 problemas a resolver: (*A*) *Speaking in Tongues*, (*B*) *Dancing with the Googlers*, (*C*) *Recycled Numbers*, y (*D*) *Hall of Mirrors*. La clasificación a las siguientes rondas requiere que los participantes proporcionen soluciones correctas a los problemas propuestos en un determinado tiempo. Los participantes no necesitan resolver todos los problemas para pasar a la siguiente ronda, pero sí necesitan conseguir una puntuación mínima. Los problemas pueden tener distintas tareas variando el tiempo límite y el tamaño de los datos de entrada para poder comprobar la calidad de las soluciones a los problemas. En concreto, los problemas B, C y D tienen tareas con datos de entrada pequeños (tareas pequeñas B_s, C_s y D_s) y con datos de entrada grandes (tareas grandes B_l, C_l y D_l), mientras que el problema A solo tiene una tarea pequeña (A_s). Para las tareas pequeñas se dispone de 4 minutos para descargar los datos, ejecutar el programa y devolver los resultados al servidor para su evaluación, mientras que para las tareas grandes se dispone de 8 minutos. Aunque un programa enviado a la tarea pequeña resuelva la tarea grande, puede que debido al tamaño de los datos, el programa no termine su ejecución en el tiempo permitido y, por lo tanto, requiera otra implementación más eficiente.

Los problemas están marcados con una puntuación según su dificultad, siendo el problema A el de menor dificultad de los cuatro y el problema D el de mayor dificultad. Los problemas B y C se consideran de dificultad media. El problema A ha sido descartado para la experimentación debido a que en los códigos fuente entregados por los participantes se encontraron mezclados colecciones de datos, diccionarios, etc. con el código fuente siendo imposible su separación automática.

En la figura 3.1 se muestran los datos de participación de cada tarea comparada con el total de programas que resolvieron correctamente las tareas. Para las tareas más sencillas (es decir, B_s, B_l, y C_s) la mayoría de envíos resolvían correctamente el problema; con una tasa de éxito entre el 89 %-95 %. Por lo contrario, en las tareas más complejas de resolver (C_l, D_s y D_l) la tasa de éxito decae significativamente al rango de 63 %-71 %. Como se esperaba inicialmente, y acorde a la puntuación de los problemas, cuanto más complejos son los problemas, menor es la tasa de éxito. Por otra parte, la reducida participación en las tareas más complejas (comparándose con las tareas más sencillas) refuerza la hipótesis de que la intención de muchos participantes es solamente obtener la puntuación mínima para avanzar a la siguiente ronda sin intentar resolver las tareas más difíciles.

Para poder realizar un estudio sobre si existe reutilización en el entorno de una competición con facilidades aparentes para que exista colaboración entre participantes, se ha seleccionado solo los programas escritos en los lenguajes de programación más utilizados en la competición: C/C++,

[3] La colección de programas se encuentra libremente disponible en: `https://code.google.com/codejam/contest/1460488/scoreboard?c=1460488#vf=1`

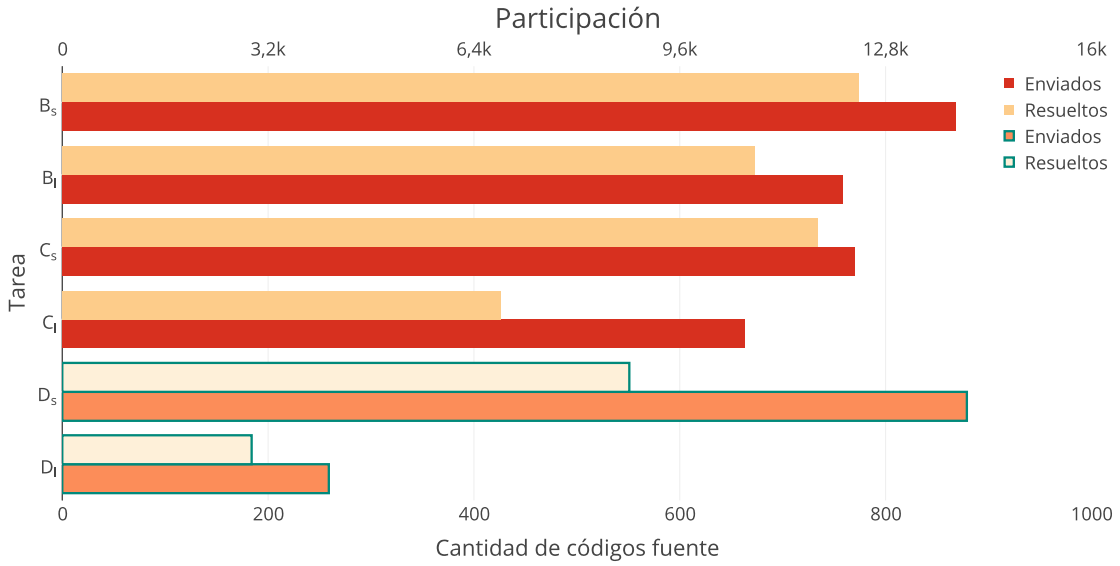


Figura 3.1: Cantidad de códigos fuente enviados por tarea: códigos fuente enviados frente a códigos fuente correctos. Las cantidades del problema D se muestran considerando el eje inferior debido a que la participación fue mucho menor.

Java, y Python. El total de códigos fuente enviados en estos tres lenguajes de programación es 34 301 (un 84,7 % del total). En la tabla 3.6 se incluyen algunas estadísticas sobre este subconjunto de códigos fuente. Los problemas B y C han requerido menos tokens, mientras que el problema más complejo es el que más tokens ha necesitado para ser resuelto. En el capítulo de experimentación, apartado 5.1.2, se mostrará que existen casos de reutilización en este corpus (de 360 pares de códigos fuente analizados manualmente se encontraron 216 reutilizados).

Lenguaje/Tarea	B_s	B_l	C_s	C_l	D_s	D_l	Total
C/C++	5 302	4 706	4 816	3 248	295	104	18 471
Java	2 696	2 321	2 727	1 557	79	29	9 409
Python	1 902	1 652	1 876	900	77	14	6 421
Total códigos	9 900	8 679	9 419	5 705	451	147	34 301
Puntos por tarea	10	10	10	15	15	25	90
Tokens promedio	204	208	200	206	558	665	211

Tabla 3.6: Estadísticas del corpus extraído de la *Qualification Round* de Google Code Jam 2012.

3.2.2 Corpus Rosettacode

Rosettacode.org es un repositorio Web de códigos fuente que presenta diferentes soluciones a un mismo problema en múltiples lenguajes de programación. Mostrando las similitudes y diferencias entre los distintos lenguajes de programación pretende conocer y aprender nuevos lenguajes de programación. Rosettacode.org dispone de 781 problemas distintos implementados en 605 lenguajes de programación.[4] Cabe destacar que no todos los problemas tienen soluciones en todos los lenguajes de programación y que un mismo problema puede tener varias soluciones en el mismo lenguaje de programación.

A partir de este repositorio se ha creado un recurso translingüe constituido por un corpus comparable y un corpus paralelo. En el ámbito del procesamiento del lenguaje natural, se considera *corpus comparable* a una colección de textos similares escritos en distintos lenguajes. El recurso comparable por excelencia es la Wikipedia (Potthast et al., 2008) por las siguientes razones: *(i)* la gran cantidad de lenguajes en que está escrita; *(ii)* el gran número de documentos que contiene; y *(iii)* los artículos que describen un mismo tópico están enlazados entre idiomas. Para cada tópico existen documentos independientes entre sí según la lengua en que está escrito, así pues el contenido de cada uno evoluciona independientemente del resto. Trasladando este concepto al tratamiento de código fuente, consideramos a Rosettacode.org como un recurso comparable de códigos fuente por: *(i)* la gran cantidad de lenguajes de programación en que está escrito; *(ii)* el volumen de códigos fuente es considerablemente grande; y *(iii)* los códigos en distintos lenguajes de programación que resuelven un mismo problema están enlazados entre sí.

	C	Java	Python
C	600		
Java	424	448	
Python	546	433	598

Tabla 3.7: Número de problemas en común entre cada par de lenguajes seleccionados de Rosettacode.org.

Para crear un corpus comparable de códigos fuente se han tomado de Rosettacode.org los problemas con soluciones escritas en los tres lenguajes más utilizados y que son los que tienen mayor número de implementaciones en común como se muestra en la tabla 3.7. En una instantánea de la página Web tomada el 27 de febrero de 2012, existían 600 problemas con soluciones escritas en el lenguaje de programación C, 598 en Python y 448 en Java. Debido a que muchos problemas tienen más de una solución en un mismo lenguaje, la cantidad final de pares de códigos fuente extraídos es mayor a la cantidad de problemas contenidos en el repositorio. Por ejemplo, el caso con mayor número de implementaciones en un mismo lenguaje con 14 soluciones escritas en el lenguaje Python es el problema *Binary string*. En la tabla 3.8 se muestran las características principales del corpus seleccionado.

Corpus paralelo se denomina a una colección formada por pares de textos que se consideran traducciones uno del otro. Estos textos han sido traducidos a nivel de oración y por lo tanto están alineados a nivel de oración. Como ejemplo de este tipo de corpus se encuentra la colección JRC Acquis[5] descrita en Steinberger et al. (2006) que consiste en traducciones de documentos legislativos

[4] Accedido el 13 de febrero de 2016.

[5] `http://optima.jrc.it/Acquis/index_2.2.html`

Lenguaje	Tokens	Longitud promedio	Tipos
C	177 962	5, 426	11 644
Java	78 398	6, 912	6 440
Python	114 197	5, 848	10 214

Tabla 3.8: Estadísticas del corpus Rosettacode.org.

de la Unión Europea a los lenguajes de toda la región. En código fuente el concepto más cercano a la oración es el de la instrucción.

En el repositorio Rosettacode.org se encontraron indicaciones en algunas soluciones acerca de la existencia de códigos fuente escritos en un lenguaje que proceden de traducciones realizadas por los colaboradores. Con el fin de utilizar estas traducciones como corpus paralelo hemos realizado un etiquetado por tres anotadores entre todas las soluciones que contienen códigos fuente escritos entre los tres pares de lenguajes. El etiquetado consiste en determinar si un par de códigos fuente se considera comparable o paralelo. Se determina si es comparable o paralelo en el caso que dos o más anotadores coincidan en su etiquetado. Después del etiquetado de los códigos fuente entre los lenguajes C++ y Java se obtuvo un valor de κ de 0,656 que significa un acuerdo sustancial (Landis et al., 1977). En total se encontraron 62 pares de códigos fuente paralelos entre este par de lenguajes. Esto representa una cantidad pequeña para aprender un diccionario estadístico. Se realizó un etiquetado por tres anotadores entre los lenguajes Java y Python resultando también una cantidad pequeña, 66 pares de códigos fuente paralelos de un total de 1588 pares. El coste temporal del etiquetado manual y el número tan bajo de pares de códigos fuente paralelos que se extrajeron de la colección son los motivos por los que se ha realizado una simulación de códigos paralelos.

Para simular un escenario paralelo, utilizamos traductores automáticos de código fuente. La construcción del corpus paralelo se desarrolla de la siguiente manera: se toman todos los códigos escritos en el lenguaje C y se traducen al lenguaje Java utilizando la herramienta de traducción *C++ to Java Converter* de la empresa Tangible Software Solutions Inc. A continuación los códigos que han sido traducidos al lenguaje Java se traducen al lenguaje Python utilizando la herramienta *java2python*[6]. Después de la traducción se han eliminado comentarios introducidos por los traductores. Cabe destacar que *java2python* crea código fuente sintácticamente (casi) igual de válido que los originales en Java, mientras que *C++ to Java Converter* puede reestructurar y refactorizar código fuente si es necesario. Esto implica que puede haber diferencias entre las longitudes de los códigos fuente original y traducido. Para el proceso de traducción, se tomaron los 959 códigos fuente escritos en el lenguaje C que forman parte del par C-Java del corpus comparable. No todos los códigos fuente tomados para la creación del corpus comparable han podido traducirse al lenguaje Python porque los traductores no han sido capaces de generar código fuente a partir del original. Por ello, solamente forman parte del corpus paralelo aquellos que tienen traducción en los tres lenguajes de programación. Por último, a causa del coste y complejidad del etiquetado manual instrucción a instrucción, se ha realizado un alineamiento a nivel de código fuente en lugar de a nivel de instrucción. En la tabla 3.9 se muestra la cantidad de pares de códigos fuente comparables y paralelos entre cada par de lenguajes. En el capítulo de experimentación, en el apartado 5.2.3, se considerarán los pares de códigos fuente paralelos como casos de reutilización y los comparables como no reutilizados.

[6] `https://github.com/natural/java2python`

Pares de lenguajes	C-Java	Java-Python	C-Python
Comparable	959	1 588	1 408
Paralelo	335	335	335

Tabla 3.9: Número de pares de códigos fuente comparables y paralelos por cada par de lenguajes de programación.

3.3 Conclusiones

La detección automática de reutilización en el mundo académico permite ayudar al profesor a identificar la colaboración entre sus estudiantes. Para mejorar los sistemas de detección automática de reutilización en código fuente se necesita disponer de recursos para entrenar, ajustar parámetros y evaluar estos sistemas. En este capítulo se han descrito los recursos que se han creado con este fin durante el presente trabajo de investigación.

Los corpus académicos se han creado a partir de trabajos de estudiantes donde, previamente y de forma manual, se encontraron casos reales de reutilización, es decir, copias. La evaluación y ajustes de los modelos con este tipo de colecciones resulta insuficiente por su tamaño reducido. Además, debido a problemas éticos y de privacidad, es prácticamente imposible encontrar recursos académicos de libre disposición, como por ejemplo de cursos masivos de programación en la Web, para el estudio de reutilización a gran escala. Por ello, se han recopilado códigos fuente de una competición internacional de programación y de un repositorio multilingüe de códigos fuente que son dos recursos de gran dimensión sobre los cuales no se han realizado estudios de reutilización automática con anterioridad.

El objetivo final es poner los recursos desarrollados a disposición de los investigadores y organismos interesados en desarrollar herramientas automáticas de detección de reutilización de códigos. Actualmente, ya se han difundido algunos de estos recursos en el marco de una competición de sistemas de detección de reutilización en código fuente (véase el capítulo 6), despertando un creciente interés en la comunidad científica que solicitan estos recursos para futuros trabajos en esta línea de investigación.

En el siguiente capítulo se detallarán las aproximaciones utilizadas en este estudio para abordar la reutilización en código fuente en general, ya sea a nivel monolingüe o a nivel translingüe. Se trata de aproximaciones que no dependen de ningún lenguaje de programación ni de ningún compilador por lo que son aplicables a cualquier lenguaje de programación.

Capítulo 4

Modelos propuestos

Uno de los principales objetivos de esta tesis es abordar el problema de la reutilización de códigos fuente escritos en un lenguaje de programación como si se tratase de un texto escrito en un lenguaje natural. Por este motivo, en este trabajo de investigación se han estudiado los principales modelos aplicados a reutilización de textos en lenguaje natural, se han implementado y adaptado para aplicarse sobre códigos fuente. En este capítulo se describen los modelos implementados para la detección de reutilización en código fuente. Todos los modelos presentados en este capítulo siguen el esquema mostrado en la figura 4.1 donde una colección de códigos fuente C es tratada inicialmente mediante un preproceso concreto. Tras este preproceso se obtiene un conjunto de características que representan a cada código fuente, y finalmente, se aplica una medida de similitud obteniéndose como resultado un listado de pares de códigos fuente junto al valor de similitud entre cada par.

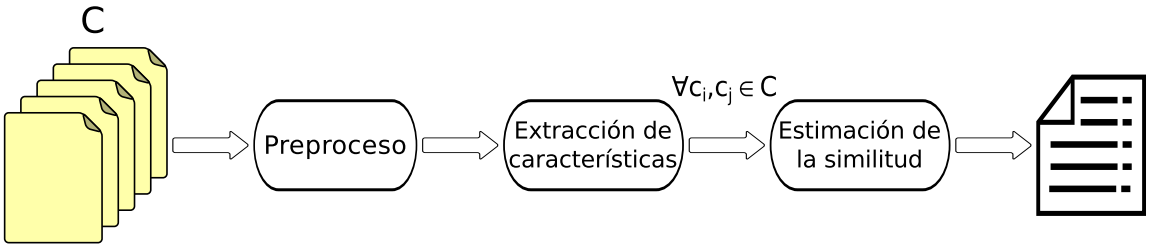


Figura 4.1: Esquema de comparación de los modelos propuestos. El resultado consiste en una lista de pares de códigos fuente junto al valor de similitud del par.

4.1 SoCo-WCR: Modelo basado en el ratio de palabras

El modelo SoCo-WCR (en inglés *Source Code Word Count Ratio*) parte de la creencia siguiente: dos programas que implementan la misma solución para resolver un problema contienen un número similar de tokens, incluso si se trata de soluciones escritas en lenguajes de programación distintos. Por lo tanto, dos códigos fuente con una gran diferencia en número de tokens es poco probable que correspondan a una solución similar.

Para realizar la comparación de una colección de códigos fuente, cada código fuente es preprocesado inicialmente. En esta primera etapa se eliminan todos los separadores sintácticos como son las comas, llaves, corchetes, etc. y solamente se consideran los tokens tales como identificadores, nombres de funciones/métodos y palabras reservadas del lenguaje de programación. Como resultado de la fase de preproceso se obtiene un listado de tokens para cada código fuente. Posteriormente, en la fase de extracción de características se calcula el número de tokens que representa a cada código fuente. Esta cantidad es la única característica que representará a cada código fuente en la fase de estimación de similitud. En la estimación de similitud, para cada par de códigos fuente se calcula la proporción entre el código fuente de menor y el de mayor cantidad de tokens. Esta proporción es calculada mediante la ecuación 4.1 donde se describe el cálculo de similitud siendo la función *longitud* la cantidad de tokens de un código fuente. Se considera similitud 1 cuando ambos códigos fuente tienen el mismo número de tokens. Como resultado se obtiene una lista de pares de códigos fuente junto al valor de similitud estimado entre cada par.

$$\varphi(c, c') = \frac{min(longitud(c), longitud(c'))}{max(longitud(c), longitud(c'))} \tag{4.1}$$

SoCo-WCR ha mostrado una alta correlación con valoraciones de anotadores humanos en tareas de medición de similitud entre lenguajes en textos (Barrón-Cedeño et al., 2014), a pesar de su sencillez y de no utilizar recursos externos. En la figura 4.2 se muestra un ejemplo de cálculo utilizando este modelo. Los tokens extraídos en la fase de preproceso se muestran coloreados.

4.2 SoCo-NG: Modelo basado en *n*-gramas

Este modelo, SoCo-NG (en inglés *Source Code N-Grams*), está basado en el modelo de espacio vectorial usando n-gramas para representar las dimensiones. Existen muchos lenguajes de programación con similitudes sintácticas y morfológicas como por ejemplo C–C++, C++–Java. Algunos lenguajes incluso comparten parte del vocabulario de palabras reservadas. Estas similitudes se pueden aprovechar para detectar reutilización utilizando n-gramas para representar el código fuente. Incluso se preservan las similitudes cuando se utilizan términos cortos como n-gramas de caracteres.

Para minimizar los cambios introducidos por el programador se reduce el conjunto de símbolos a un alfabeto mediante un preproceso aplicado a código fuente de la colección C. Una vez realizado este preproceso, se obtiene una secuencia de caracteres en minúsculas, sin espacios, ni saltos de línea ni tabulaciones para cada código fuente. En la fase de extracción de características, el código fuente preprocesado se divide en n-gramas de caracteres con solapamiento, es decir, el siguiente n-grama contiene los últimos $n-1$ caracteres del anterior. Estos n-gramas se almacenan en una

Código fuente c_1

```
long fact(int n) {
    if (n < 0) {
        System.err.
            println(
        "No negative
            numbers");
        return 0;
    }
    long ans = 1;
    for (int i = 1;
            i <= n;
            i++) {
        ans *= i;
    }
    return ans;
}
```

longitud(c_1)=25

Código fuente c_2

```
long fact(int n) {
    if (n < 0){
        System.err.
            println(
        "No negative
            numbers");
        return 0;
    }
    return (n < 2) ?
        1 :
        n * fact(n-1);
}
```

longitud(c_2)=18

Código fuente c_3

```
long recFibN(int n)
    {
 return (n < 2) ?
            n :
        recFibN(n-1)
            +
        recFibN(n-2);
}
```

longitud(c_3)=11

$\varphi(c_1, c_2) = 18/25 = \mathbf{0{,}72}$ $\varphi(c_1, c_3) = 11/25 = 0{,}44$

Figura 4.2: Ejemplo de comparación del modelo SoCo-WCR. Los dos primeros códigos fuente corresponden a diferentes implementaciones de la solución al problema del número factorial, mientras que la tercera de la solución al problema de Fibonacci. Basándose en el número de palabras de cada código, el código fuente c_1 es más similar a c_2 que a c_3.

bolsa de palabras que en este caso serán bolsas de n-gramas.[1] Además, en esta bolsa de palabras también se almacena la frecuencia de aparición de cada n-grama. Al perder la localización espacial de los n-gramas no importa si en otro código fuente se altera la distribución del código fuente. Por ejemplo mover la declaración de una función del inicio del código fuente a cualquier otra posición de este.

Una vez estimados los pesos de los n-gramas contenidos en cada código fuente en base a su frecuencia, por ejemplo con tf o tf-idf, nos interesa conocer la similitud existente entre la colección C de códigos fuente. Tf (en inglés *term frequency*) 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 han aparecido 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, es decir, aquellos términos que están en un código fuente y no en el resto, por tanto lo caracterizarán mejor. Dado que tf destaca las palabras relevantes dentro del código y *idf* destaca las palabras relevantes entre los códigos fuente, el método de pesado tf-idf, considera ambas propiedades y consiste en el simple producto de ambas medidas.

[1] Durante todo el documento de tesis, aunque se empleen bolsas de n-gramas se utilizará el término *bolsa de palabras* por ser más conocido en el ámbito de la recuperación de información.

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 (McNamee et al., 2004). En la fase de estimación de la similitud se comparan todos los pares de códigos fuente de la colección de dos en dos. Para aplicar esta medida de similitud es necesario disponer de dos vectores representados en el mismo espacio vectorial. En nuestro caso, disponemos de dos bolsas de palabras de la fase de extracción de características, siendo considerado cada n-grama de la bolsa de palabras como una característica del espacio vectorial, por lo que el espacio vectorial está compuesto por todos los n-gramas contenidos en ambas bolsas de palabras.[2] Los valores obtenidos con el sistema de pesado representan al código fuente en el espacio vectorial. En una bolsa de palabras puede haber n-gramas no contenidos en la otra; esta ausencia se representa con un valor 0.

Después de tener caracterizadas las bolsas de palabras como vectores dentro de un espacio vectorial, la función de similitud mide en ese espacio vectorial el coseno del ángulo que forman ambos vectores. La fórmula del coseno se puede deducir a partir de la ecuación del producto euclídeo de dos vectores 4.2:

$$A \cdot B = \|A\| \, \|B\| \cos(\theta). \tag{4.2}$$

Por lo que finalmente el cálculo del coseno equivale a calcular el producto escalar de dos vectores y dividirlo por la raíz cuadrada del sumatorio de los cuadrados de los componentes del primer vector multiplicada por la raíz cuadrada del sumatorio de los componentes del segundo vector como se desglosa en la ecuación 4.3:

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}. \tag{4.3}$$

Los vectores correspondientes a dos códigos fuente cuyas bolsas de palabras sean idénticas, tendrán las mismas características y formarán un ángulo de 0°. Al calcular el coseno del ángulo obtendremos 1 que corresponde a la máxima similitud, $cos(0°) = 1$. En el caso contrario, cuando ambas bolsas de palabras no comparten ningún n-grama, y por lo tanto ninguna característica en el espacio vectorial, los vectores serán perpendiculares y formarán un ángulo de 90°, siendo 0 el valor del coseno, $cos(90°) = 0$. Utilizando la función del coseno disponemos de una medida de similitud sencilla, normalizada en el rango $[0 - 1]$ y que permite comparar numéricamente cómo de similares son dos códigos fuente. La similitud entre dos códigos fuente se expresa como se muestra en la ecuación 4.4 donde c y c' son dos códigos fuente y t representa los términos:

$$cos(c, c') = \frac{\sum_{t \epsilon c \cap c'} tf_{t,c} tf_{t,c'}}{\sqrt{\sum_{t \epsilon c} tf_{t,c}^2 \sum_{t \epsilon c'} tf_{t,c'}^2}}. \tag{4.4}$$

[2]En verdad el espacio vectorial consiste en que existen tantas dimensiones como representaciones de n-gramas. Una gran cantidad de n-gramas no se va a representar al comparar códigos fuente quedando un espacio vectorial disperso. Por ello, para simplificar costes espaciales y temporales se realizan los cálculos directamente sobre aquellas dimensiones presentes en los códigos fuente.

Cuando se aplica este modelo sobre códigos fuente se puede realizar un preproceso eliminando partes del código fuente como son los comentarios, funciones o métodos que no se ejecutan o, como se verá en el capítulo de experimentación, eliminando caracteres repetidos en gran cantidad que el programador utiliza para resaltar la separación de comentarios.

código fuente c_2	=	"long fact(int n) "
código fuente c_3	=	"long recFibN(int n)"
preproceso + trigramas(c_2)	=	lon ong ngf gfa fac act ct(t(i (in int ntn tn)
preproceso + trigramas(c_3)	=	lon ong ngr gre rec ecf cfi fib ibn bn(n(i (in int ntn tn)
[ecf, ngr, cfi, int, n(i, ong, ntn, fac, gfa, (in, ngf, ibn, ct(, lon, fib, gre, t(i, act, bn(, rec, tn)]		
repr. vectorial(c_2)	=	[0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1]
repr. vectorial(c_3)	=	[1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1]
coseno(c_2, c_3)	=	0,4472135955

Figura 4.3: Ejemplo del cálculo de la similitud con el modelo SoCo-NG dados dos códigos fuente. En la bolsa de trigramas de caracteres se muestra el código previamente preprocesado y dividido en trigramas. Finalmente se calcula el valor del coseno del ángulo que forman ambos vectores.

En la figura 4.3 se muestra un ejemplo de aplicación del modelo SoCo-NG utilizando trigramas de caracteres sobre dos fragmentos de los códigos fuente c_2 y c_3 del ejemplo de la figura 4.2. En la primera etapa se realiza el preproceso eliminando saltos de línea, espacios, tabulaciones, se convierten a minúsculas los caracteres y se extraen los n-gramas de cada código fuente. En la segunda fase se realiza el pesado de los n-gramas con tf, y estos pesos estimados se consideran como una dimensión independiente en un espacio vectorial con el valor de magnitud obtenido. Finalmente se comparan las representaciones de los códigos fuentes en el espacio vectorial mediante el cálculo del coseno del ángulo que forman los vectores. Este valor nos mostrará la similitud existente entre ambos códigos fuente.

Dado que se va a comparar cada código fuente contra el resto de códigos fuente de la colección, una multitud de cálculos se repiten al aplicar la fórmula del coseno 4.4, por ejemplo los sumatorios $\sum_{i=1}^{n}(c_i)^2$ ó $\sum_{i=1}^{n}(c'_i)^2$ para cada comparación. Una forma más eficiente de realizar este cálculo consiste en guardar los n-gramas en un índice invertido como se muestra en la figura 4.4. El índice invertido de n-gramas consiste en generar un diccionario con todos los n-gramas que hay en los códigos fuente que se van a comparar. Cada entrada del diccionario enlazará con una lista de tuplas. Estas tuplas están constituidas por el identificador del códigos fuente y el número de ocurrencias del n-grama en dicho código. El uso del índice invertido permite que además de comparar dos códigos fuente a la vez, se puedan comparar más de dos, añadiendo tantos códigos fuente como se desee. La información de los códigos fuente no se almacena como una bolsa de palabras, sino como una lista invertida. Por lo tanto, para el cálculo de la similitud del coseno se desglosa la fórmula de la siguiente forma:

1. Para realizar el cálculo del numerador ($\sum_{i=1}^{n} c_i \times c'_i$) se almacenan los datos en una matriz de tamaño $número\_total\_de\_códigos \times número\_total\_de\_códigos$. Dado que la función del coseno cumple la propiedad de simetría (los dos vectores formarán el mismo ángulo dentro del espacio vectorial en el rango 0°-90°), el espacio final necesario en memoria es la mitad por

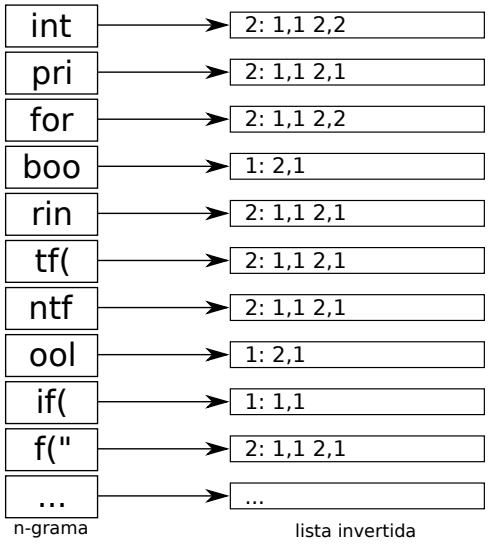


Figura 4.4: Lista de n-gramas invertida con los 9 primeros trigramas de dos códigos fuente. Cada entrada muestra la cantidad de códigos fuente que contienen el n-grama, el identificador individual del código fuente y las ocurrencias del n-grama en cada código.

la propiedad conmutativa del producto. Es decir, se obtiene el mismo valor para el producto del par (i, j) que para el par (j, i), por lo tanto el resultado se puede almacenar en una matriz triangular.

2. Para almacenar el valor del sumatorio $\sum_{i=1}^{n}(c'_i)^2$, se utiliza un vector de tamaño igual al número total de códigos fuente. Una vez calculados los resultados de esta estructura se calcula la raíz cuadrada de los valores.

Es necesario identificar qué cálculos se van a poder realizar recorriendo el diccionario de n-gramas una sola vez. Con el fin de obtener la similitud entre todos los códigos fuente, para cada par de códigos se divide el resultado en el paso 1 por el producto del valor de los dos códigos fuente calculado en el paso 2. Llegado a este punto, se dispone de tantos valores de similitud, como pares de códigos fuente se han comparado.

4.3 SoCo-Sliding: Modelo basado en ventana deslizante

Un aspecto no estudiado en trabajos como el de Whale (1990) o Faidhi et al. (1987) reside en considerar si la reutilización va a ser total o solo de partes del código original. Es por ello que surge la necesidad de identificar 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. En el ejemplo mostrado en la figura 4.5 se muestra la detección en un par de códigos fuente. La única modificación realizada sobre el código fuente original ha sido la de intercambiar el fragmento de código 1 por el 3. En el caso de haber sido una

copia exacta se mostraría una única diagonal desde la esquina superior izquierda hasta la inferior derecha.

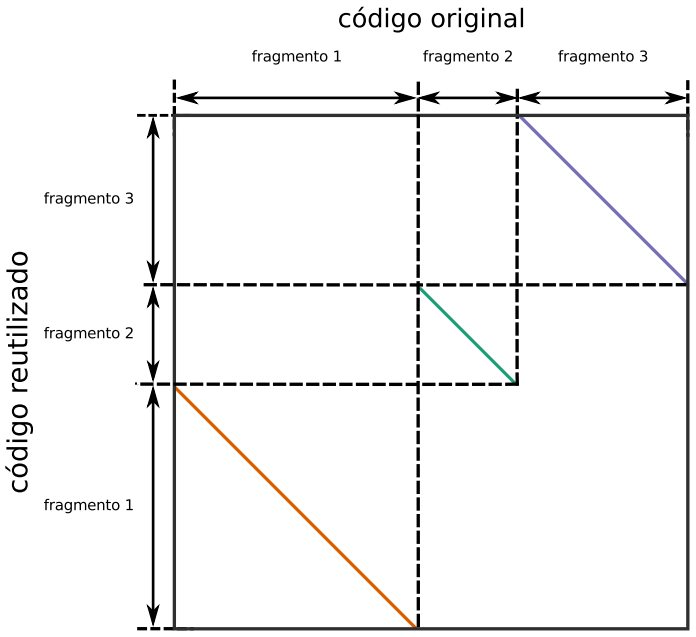


Figura 4.5: Ejemplo de detección de fragmentos en dos códigos fuente. El fragmento 1 ha sido intercambiado por el fragmento 3.

Partiendo de esta idea, inicialmente se aplica sobre la colección de códigos fuente C el mismo preproceso que en el modelo a nivel de documento: se eliminan los saltos de línea, los espacios en blanco, tabulaciones y se convierte todo el texto a minúsculas. Como resultado obtenemos un vector de caracteres que representa cada código fuente. En la etapa de extracción de características, se divide cada vector de caracteres en fragmentos que consisten 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 SoCo-Sliding (en inglés *Source Code Sliding window*) 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.

A partir del primer caracter, 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 sitúa 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 un número mayor de caracteres a la longitud de la ventana anterior. Una vez que cada código fuente se ha dividido 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, así

como se ha descrito para el modelo SoCo-NG. Se ha de incluir un proceso de normalización de las bolsas de palabras, dado que la última ventana de un código fuente puede tener un tamaño menor que el resto de ventanas. En el caso del modelo SoCo-NG se disponía de dos bolsas de palabras para comparar cada par de códigos, mientras que en el modelo SoCo-Sliding se dispone de tantas bolsas de palabras como cantidad de fragmentos en que se hayan dividido ambos códigos fuente. Supongamos que disponemos de un código fuente c y otro c', de los cuales se ha extraído N y M ventanas respectivamente ($c_1 \dots c_N$ y $c'_1 \dots c'_M$). En la fase de estimación de similitud, se van a comparar las N ventanas de c con las M de c', dando como resultado una matriz de valores de similitud de tamaño $N \times M$. En la figura 4.6 se muestra el esquema de la fase de similitud del modelo SoCo-Sliding donde se recibe como entrada las ventanas de los códigos fuente c y c'; las ventanas del código c se comparan con las ventanas de c' mediante una medida de similitud, en este caso la del coseno.

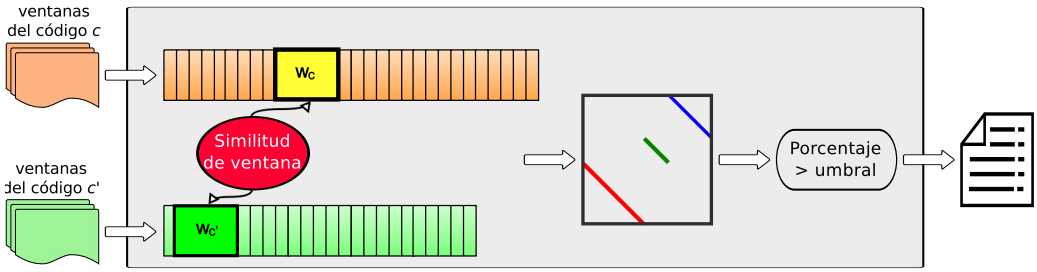


Figura 4.6: Esquema de la fase de estimación de similitud de dos códigos c y c' en el modelo basado en ventana deslizante (SoCo-Sliding).

Al igual que ocurre en el modelo SoCo-NG, se repiten muchos cálculos al comparar una colección de códigos fuente. En el caso del modelo SoCo-Sliding, los cálculos repetidos van a ser muchos más porque dentro de la comparación de un par de códigos fuente también se repiten muchos cálculos (cada ventana se compara con todas las ventanas del otro código fuente). Es por ello que en este modelo se aprovecha doblemente la estructura del índice invertido. La única modificación que requiere esta estructura es que en vez de utilizar un identificador para el código fuente en la lista de tuplas de cada entrada del índice invertido, se utiliza un identificador de ventana. Por lo tanto, las tuplas del índice invertido quedan definidas como el identificador de ventana junto al número de ocurrencias del n-grama en dicho ventana.

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, 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 entre dos códigos fuente se desglosa la fórmula de la siguiente forma:

1. Para realizar el cálculo del numerador ($\sum_{i=1}^{n} c_i \times c'_i$) se almacenan 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 formarán el mismo ángulo dentro del espacio vectorial 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 par (i, j) como para el par $(< j, i >)$, por lo tanto se puede almacenar en una matriz triangular.

2. Para almacenar el sumatorio $\sum_{i=1}^{n}(c_i')^2$, se utiliza 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.

Una gran cantidad de cálculos repetidos se van a poder realizar recorriendo el diccionario de n-gramas una sola vez. Para obtener la similitud entre todas las ventanas, para cada par de ventanas se divide el resultado en el paso 1 por el producto del valor de las dos ventanas calculado en el paso 2. 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 códigos fuente partiendo de comparaciones entre ventanas de estos códigos.

Pares de lenguajes	Tamaño ventana (s)	Desplazamiento (l)	Umbral (t)
Java–C++	50	50	0,8
Python–C++	50	50	0,1
Java–Python	30	30	0,2

Tabla 4.1: Parámetros calculados para cada par de lenguajes de programación sobre el modelo de ventana deslizante.

Una primera aproximación para obtener la similitud final 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 de 0 aquellos pares de códigos fuente en cuyas comparaciones ningún par de ventanas supere el umbral y un valor de 1 si todos los pares de ventanas superan dicho umbral. En la tabla 4.1 se muestran las configuraciones a nivel translingüe de los parámetros que obtienen mejores resultados en cada par de lenguajes de programación utilizando el corpus SPADE mediante validación cruzada. La mejor combinación de parámetros entre el par Java–C++ 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 por ello, la similitud entre ventanas que no han sido reutilizadas es mayor que entre dos lenguajes que no comparten sintaxis. Respecto a los otros dos pares de lenguajes, Python–C++ y Java–Python, se ha estimado 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, en concreto 0,1 y 0,2 respectivamente. En el escenario entre Java–Python, al utilizar una ventana menor, hay menos variedad de caracteres por lo tanto se necesita un umbral más alto que el utilizado en el escenario Python—C++.

4.4 SoCo-COG: Modelo basado en cognados

En lingüística los cognados se definen como pares de palabras que pertenecen a diferentes lenguajes que normalmente comparten significado y alguna propiedad fonológica o morfológica (Voga et al., 2007). Utilizar este tipo de palabras nos permite detectar similitudes entre textos escritos en diferentes lenguajes. Por ello, estos pares de palabras se pueden utilizar como traducciones mutuas y como modelo para alinear oraciones entre pares de lenguajes (Simard et al., 1993). Por ejemplo, son cognados en inglés y español: <*analyze*, *analizar*>, <*artistic*, *artístico*>, <*concert*, *concierto*>, entre

otros. En lenguajes de programación también existen palabras heredadas de otros lenguajes de programación, como <*for*, *foreach*>. Además, una de las modificaciones principales cuando se reutiliza de código fuente es modificar ligeramente los nombres de identificadores como variables, nombres de funciones o clases. A este tipo de pares de tokens modificados les llamaremos *pseudo-cognados* porque se han creado de manera artificial no transmitiéndose entre lenguajes de programación. Así pues, con un modelo de detección de reutilización basado en cognados (SoCo-COG, en inglés *Source Code based on Cognates*), podemos detectar este tipo de modificaciones en el código fuente. En la tabla 4.2 se muestran algunos ejemplos más de cognados tanto en lenguaje natural como en códigos fuente.

Lenguaje natural	Portugués	`capacidade`
	Inglés	`capacity`
	Español	`vocabulario`
	Inglés	`vocabulary`
Código fuente	C	`for`
	C#	`foreach`
	Visual Basic	`Integer`
	C++	`int`

Tabla 4.2: Ejemplos de cognados en el lenguaje natural y en códigos fuente.

Para detectar automáticamente los cognados y pseudo-cognados es necesario definir qué se considera un cognado dentro del ámbito de los lenguajes de programación. No se trata de una lista cerrada de tokens compartidos por dos lenguajes de programación debido a la libertad del programador para modificar nombres de variables, funciones, clases, etc. Para poder identificar los cognados y pseudo-cognados en un código fuente, en la fase de preproceso se eliminan los delimitadores del código fuente como paréntesis, llaves, etc., y se convierten a minúsculas todos los caracteres obteniéndose como resultado todos los tokens del código fuente.

source code c	=	"int calculator = myHappyAndNewClass.guessTheNumber();"
source code c'	=	"int calc = myHappyClass.guesTheNum(); float counter = 0;"
pre-process + candidates(c)	=	int calculator myhappyandnewclass guessthenumber
pre-process + candidates(c')	=	int calc myhappyclass guesthenum float counter 0
direct cognates (c, c')	=	(int, int)
Levenshtein cognates (c, c')	=	(calculator, calc) (myhappyandnewclass, myhappyclass) (guessthenumber, guesthenum)
$\varphi(c, c')$	=	4 / ((4+7)/2) = 4 / 5.5 = 0.7272

Figura 4.7: Ejemplo del proceso del cálculo de similitud utilizando el modelo SoCo-COG.

En la fase de extracción de características, se consideran como candidatos a cognados todos aquellos tokens con al menos un dígito numérico y aquellos tokens compuestos por solo letras con al menos 3 caracteres. En la etapa de estimación de la similitud se eligen los cognados encontrados entre ambos códigos fuente. Se eligen como cognados aquellos tokens que sean exactamente iguales. En el ejemplo de la figura 4.7 solo el token *int* es elegido en este paso. Además, entre aquellos pares de candidatos que compartan los tres primeros caracteres, se eligen aquellos con mayor valor de distancia de

Levenshtein (Levenshtein, 1966). Es importante mencionar que cada token solo se considera en un único cognado, es decir, el mismo token no puede aparecer en dos cognados distintos. Tras establecer los cognados entre dos códigos fuente, se aplica la ecuación 4.5 para estimar la similitud entre los dos códigos fuente:

$$\varphi(c, c') = \frac{cognados}{(candidatos_1 + candidatos_2)/2} \quad , \tag{4.5}$$

donde los parámetros $candidatos_1$ y $candidatos_2$ son el número de candidatos a cognado en cada código fuente, y el número de cognados emparejados es el parámetro *cognados*. Este proceso se aplica a todos los pares de códigos fuente de la colección C dando como resultado un listado de pares de códigos fuente con sus valores de similitud.

4.5 SoCo-LSA: Modelo basado en análisis semántico latente

La mayoría de modelos de recuperación de información dependen de coincidencias exactas entre los términos de la consulta de búsqueda y los términos de los documentos de la colección donde se realiza la búsqueda. Sin embargo, estos modelos pueden fallar al recuperar documentos relevantes que no comparten términos con las búsquedas. Una de las razones es que los modelos estándar (por ejemplo el modelo booleano) tratan los términos como si fuesen dimensiones independientes cuando en realidad no lo son. El análisis semántico latente (*LSA*, en inglés *Latent Semantic Analysis*) (Deerwester et al., 1990) permite modelar la relaciones entre términos de un mismo documento y así mejorar la recuperación.

LSA examina la similitud entre "contextos" en los que aparece un término y crea un nuevo espacio vectorial de menores dimensiones siguiendo el principio de que términos que aparecen en contextos similares están próximos. LSA utiliza el método de Descomposición en Valores Singulares (*SVD*, en inglés *Singular Value Decomposition*) para descubrir las relaciones entre los términos de una colección de documentos. Las relaciones calculadas automáticamente son específicas del dominio de los documentos de la colección. No requiere de diccionarios externos, tesauros o bases de conocimiento para definir las relaciones porque se calculan a partir de analizar colecciones de documentos.

La técnica SVD es similar a la descomposición en valores propios de una matriz (en inglés *eigenvector*) y al análisis factorial (Cullum et al., 1985). Se empieza construyendo una matriz de términos que aparecen en los documentos del mismo modo que en el modelo booleano o de espacio vectorial (Salton et al., 1986). Esta matriz se descompone en un conjunto de k factores ortogonales que se calculan por aproximación lineal. El valor del parámetro k es menor que el tamaño del espacio vectorial que representa la matriz de términos. La aproximación lineal "descubre" la estructura "latente" en la matriz a partir de la frecuencia de los términos en los documentos.

El resultado de SVD es un conjunto de vectores que representan la situación de cada término y documento en un espacio reducido de tamaño k. La recuperación se realiza situando los términos de la consulta en el espacio vectorial reducido. Posteriormente, los documentos se ordenan en base a su similitud con la consulta utilizando generalmente la métrica del coseno.

Los modelos tradicionales representan los documentos como combinaciones lineales donde los términos no están correlacionados. En cambio, LSA representa los términos como valores continuos en cada una de las dimensiones ortogonales siendo los términos dependientes unos de otros. Cuando dos términos aparecen en contextos (documentos) similares, estos tendrán vectores similares en la matriz reducida. LSA resuelve parcialmente algunas de las deficiencias de asumir independencia entre los términos. En el trabajo Deerwester et al. (1990) se presentan descripciones matemáticas y ejemplos de la base del modelo LSA y el método SVD.

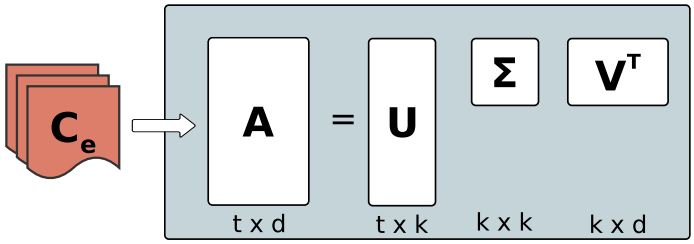


Figura 4.8: Esquema de la fase de entrenamiento del modelo SoCo-LSA dada una colección de códigos fuente de entrenamiento C_e.

En la figura 4.8 se detalla la fase de entrenamiento del modelo SoCo-LSA. A partir de una colección de entrenamiento de códigos fuente C_e se crea una matriz A de frecuencias de tamaño $número\_términos \times número\_de\_códigos$. Utilizando la descomposición SVD descrita anteriormente se obtienen las matrices U, Σ y V. La matriz V corresponde con la representación de los códigos fuente de la colección C_e en el nuevo espacio vectorial que tiene k dimensiones. Las matrices U y Σ nos permitirán proyectar al nuevo espacio vectorial cualquier código fuente que esté representado como la colección de entrenamiento, es decir, con las mismas t dimensiones. Por ello, durante el proceso de estimación de similitud se utilizaran ambas matrices.

Para el proceso de estimación de similitud con este modelo, inicialmente la colección de códigos fuente C a comparar se preprocesa eliminando espacios en blanco, tabulaciones, saltos de líneas y los caracteres se convierten en minúsculas. La fase de extracción de características se describe en la figura 4.9. Cada código fuente se convierte a una representación vectorial de t dimensiones. Para realizar la proyección de cada código fuente se realiza el producto con las matrices U y Σ^{-1} respectivamente dando lugar a una representación vectorial de k dimensiones. Una vez obtenidas las representaciones de todos los códigos fuente, para estimar su similitud se comparan de dos en dos el nuevo espacio vectorial. Esta fase es idéntica a la descrita para el modelo SoCo-NG, donde se comparan dos vectores mediante la similitud del coseno.

LSA también se puede adaptar fácilmente a la tarea de recuperación de información entre distintos lenguajes como se propuso en Landauer et al. (1990). Para ello, a priori se traduce una colección de documentos a otro idioma para crear un conjunto de documentos paralelos de entrenamiento. Cada documento se concatena con su traducción formando una colección dual de documentos en ambos idiomas. Este conjunto se analiza con LSA y el resultado es un espacio reducido de k dimensiones donde los términos relacionados están cerca unos de otros. Debido a que los documentos de entrenamiento contienen términos en ambos lenguajes, el espacio vectorial reducido también contiene términos pertenecientes a ambos. La utilización conjunta de los documentos evita que

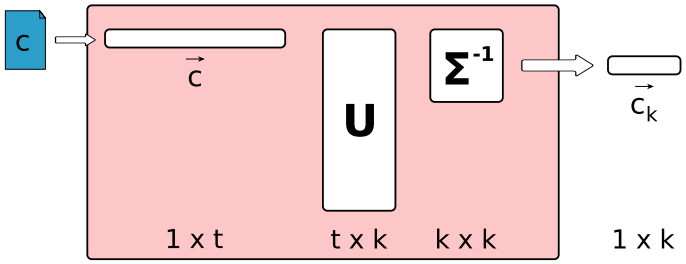


Figura 4.9: Esquema de la fase de extracción de características del modelo SoCo-LSA dado un código fuente c.

para realizar consultas sea necesario un proceso de traducción. Las palabras de un lenguaje tendrán una representación similar a las de sus traducciones en el espacio reducido de k dimensiones.

Por lo tanto, la búsqueda de documentos similares se realiza proporcionando solamente los documentos en un único lenguaje. Debido a que los documentos han sido entrenados concatenados con las traducciones, los términos en ambos lenguajes siguen distribuciones similares en el espacio reducido. De esta forma, se pueden realizar búsquedas independientes del lenguaje ya que se han calculado previamente las relaciones "latentes" entre los términos de ambos lenguajes.

Para aplicar este modelo sobre códigos fuente a nivel monolingüe se extraen n-gramas, ya sean secuencias de caracteres, palabras o secuencias de palabras, como dimensiones del código fuente. Para encontrar las similitudes entre una colección de códigos fuente es necesario entrenar el modelo SoCo-LSA con las características extraídas de cada código fuente y así obtener el espacio reducido de k dimensiones. Para comprobar la similitud de un código fuente contra los códigos fuente de la colección, se extraen las características de este y se proyectan sobre la matriz reducida de k dimensiones. Esta proyección devuelve la similitud de este código fuente consulta respecto a los códigos fuente de la colección.

Para aplicar el modelo SoCo-LSA a nivel translingüe, así como ocurre en textos, se requiere un corpus paralelo escrito en dos lenguajes de programación. Como se describió en el apartado 3.2.2, consideramos como códigos fuente paralelos traducciones de un lenguaje de programación a otro distinto. Con este corpus paralelo de códigos fuente se calcula el espacio reducido de k dimensiones translingüe que contiene relaciones entre términos de ambos lenguajes de programación. Para comparar dos códigos fuente escritos en distintos lenguajes de programación es necesario representar por separado ambos códigos fuente en el espacio vectorial de k dimensiones y comparar la similitud de estas proyecciones.

En la figura 4.10 se muestran las curvas de Cobertura[3] sobre el ranking variando las dimensiones k del espacio vectorial reducido. Se ha calculado la Cobertura sobre los ranking top-{1, 2, 3, 4, 5, 10, 20, 50}. Las dimensiones evaluadas se encuentran desde 2 hasta 1 024 siguiendo potencias de 2. Estos parámetros se han estimado a partir de la colección de códigos fuente paralela Rosettacode.org descrita en el apartado 3.2.2. Se aprecia que a partir de 256 dimensiones el acierto del modelo se

[3]Nos referimos al término Cobertura como a la medida de clasificación conocida en inglés como *Recall*.

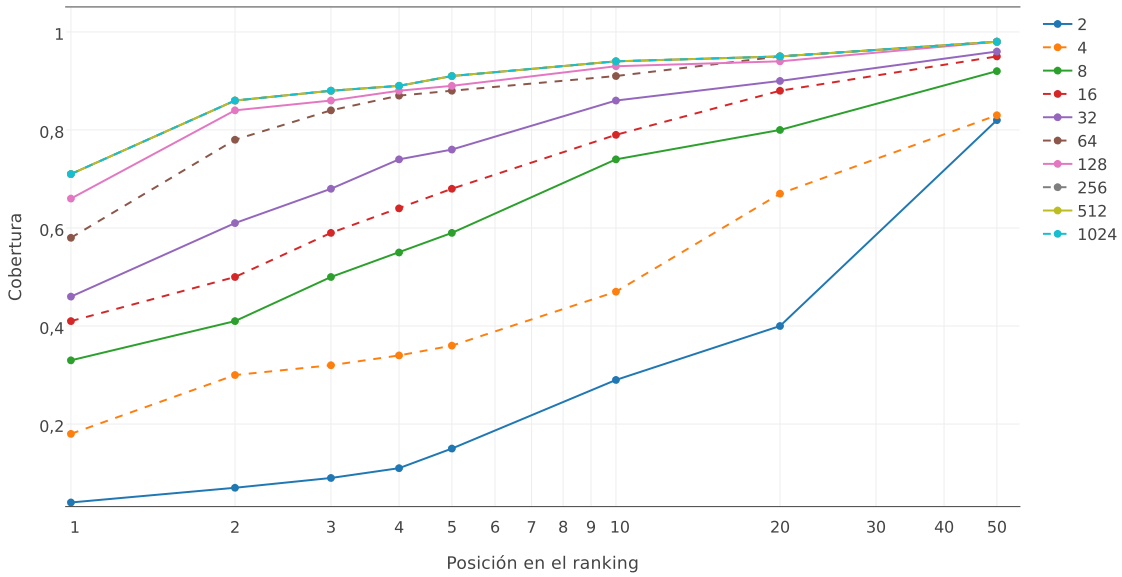


Figura 4.10: Curvas de Cobertura sobre los top-m pares más similares encontrados para cada valor de k dimensiones del modelo SoCo-LSA sobre el par de lenguajes C-Java comparable.

estabiliza. Por lo tanto no es necesario utilizar un número mayor de dimensiones que van a obtener los mismos resultados con un coste computacional mayor.

4.6 SoCo-ESA: Modelo de análisis semántico explícito

SoCo-ESA (en inglés *Source Code Explicit Semantic Analysis*) se basa en el modelo de análisis semántico explícito (ESA) que ha sido aplicado para recuperación de información textual (Gabrilovich et al., 2007). ESA representa el texto a través de una cantidad masiva de conocimiento. Cada texto se representa como una combinación de pesos sobre unos conceptos predeterminados.

Un texto d se representa como el vector $\overrightarrow{d}$ de similitudes respecto a conceptos x de un espacio de conocimiento K. Cada posición de este vector representa la similitud del texto d con cada concepto en K. El cálculo de la similitud de un texto de entrada con un concepto se puede realizar aplicando alguna métrica de similitud como la del coseno. Este modelo utiliza conceptos definidos por un anotador humano. Por ejemplo, en el procesamiento de textos se utiliza Wikipedia como una representación masiva de conocimiento, siendo cada artículo un concepto del espacio de conocimiento K.

Cuando dos textos d y d' escritos en los lenguajes L y L' se comparan contra los conceptos, como en las ecuaciones 4.6 y 4.7, estos se representan mediante los vectores de similitudes $\overrightarrow{d}$ y $\overrightarrow{d'}$ que

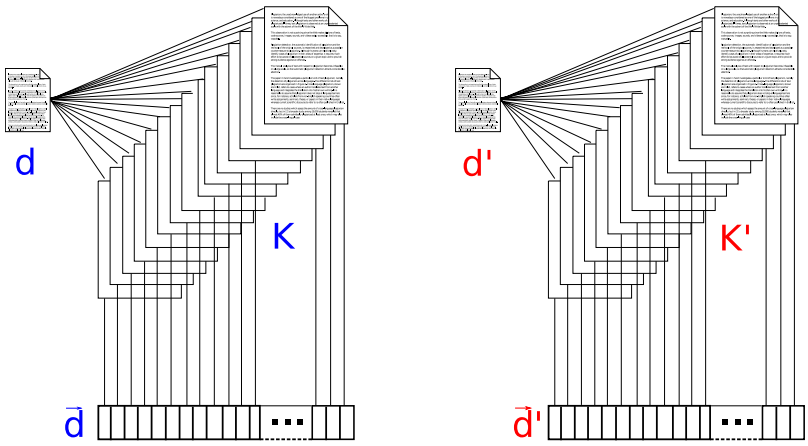


Figura 4.11: Esquema de comparación del modelo basado en análisis explícito semántico. En un escenario monolingüe d y d' están escritos en el mismo lenguaje y $K = K'$, mientras que en un escenario translingüe d y d' están escritos en lenguajes diferentes L y L' así como K y K'consideradas colecciones comparables.

se comparan entre ellos como se muestra en la ecuación 4.8. Si ambos textos tienen una similitud elevada en los mismos conceptos, se deduce que ambos textos tratan de un tema similar o relacionado. Por lo contrario, si los conceptos con mayor similitud correspondientes a ambos textos son completamente distintos se deduce que el contenido de los documentos tratan temas distintos.

$$\overrightarrow{d} = \{sim(d,x) \quad \forall x \in K \quad / \quad K \in L\} \tag{4.6}$$

$$\overrightarrow{d'} = \{sim(d',x) \quad \forall x \in K' \quad / \quad K' \in L'\} \quad , \; es \; decir, \tag{4.7}$$

$$\overrightarrow{d} = [sim(d,x_0), sim(d,x_1), \ldots, sim(d,x_K)] \quad \overrightarrow{d'} = [sim(d',x_0), sim(d',x_1), \ldots, sim(d',x_K)]$$

$$\varphi(d,d') = sim(\overrightarrow{d}, \overrightarrow{d'}) \tag{4.8}$$

En la figura 4.11 se resume el cálculo de la similitud con el modelo ESA. Cuando d y d' están escritos en el mismo lenguaje, se calculan sus respectivos vectores con la misma colección K que representa el espacio de conocimiento. En el ejemplo, K y K' serían la misma colección en una comparación monolingüe. Sin embargo, si ambos documentos están escritos en lenguajes distintos L y L' se utilizan colecciones distintas, es decir, $K \neq K'$. En concreto, se utilizan colecciones comparables de textos que previamente han sido etiquetadas como comparables a nivel de concepto, es decir, que cada concepto está expresado en ambas colecciones. Wikipedia sigue siendo un recurso muy útil para la aplicación de este modelo a nivel translingüe en textos dado que contiene una gran cantidad de conceptos etiquetados manualmente y enlazados por idiomas. Utilizando los enlaces se encuentra disponible una gran cantidad de conceptos para crear la representación del conocimiento incluso entre lenguajes distintos.

Para poder aplicar este modelo sobre código fuente a nivel monolingüe es necesario disponer de un recurso que represente el espacio de conocimiento y que este espacio esté marcado manualmente por conceptos así como ocurre con los textos en la Wikipedia. En el apartado 3.2.2 se detalla el repositorio Web Rosettacode.org que contiene distintas implementaciones de algoritmos comunes introducidos por los usuarios. Considerando cada problema que resuelve un código fuente como un concepto, podemos considerar Rosettacode.org como un espacio de conocimiento en códigos fuente. Cada código fuente r de la colección Rosettacode.org R corresponde a un código fuente que resuelve un problema determinado. La adaptación a códigos fuente y el cálculo de la similitud se resume en las ecuaciones 4.9 y 4.10:

$$\overrightarrow{c} = \{sim(c, r) \quad \forall r \in R \;\; / \;\; R \in LP\} \tag{4.9}$$

$$\varphi(c, c') = sim(\overrightarrow{c}, \overrightarrow{c'}) \tag{4.10}$$

Una de las particularidades de la colección Rosettacode.org es que además de estar dividida por problemas resueltos, también está etiquetada por lenguajes de programación. Tomando aquellas implementaciones que están en un mismo par de lenguajes de programación LP y LP' se puede construir una base de conocimiento comparable para dicho par de lenguajes. Estas representaciones no son necesariamente traducciones, solo deben contener códigos fuente resolviendo el mismo problema. Es decir, el corpus requerido es un corpus comparable tal como está definido en el apartado 3.2.2. Así pues, se puede utilizar esta base de conocimiento comparable entre códigos fuente para comparar códigos fuente escritos en dos lenguajes de programación distintos.

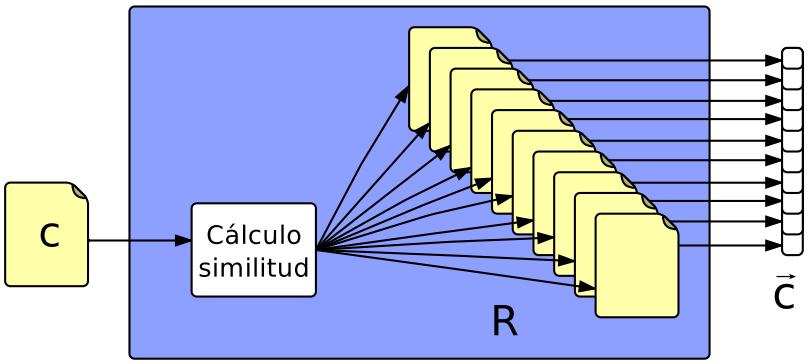


Figura 4.12: Esquema de la fase de extracción de características del modelo SoCo-ESA. En un escenario monolingüe los códigos fuente c, c' y los las representaciones de conocimiento R y R' están escritos en el mismo lenguaje de programación siendo $R = R'$. En un escenario translingüe c y R están escritos en un lenguaje de programación LP distinto al lenguaje LP' de c' y R', siendo R y R' colecciones comparables.

Para comparar una colección de códigos fuente C con el modelo SoCo-ESA, esta colección se preprocesa eliminando espacios en blanco, tabulaciones, saltos de línea y convirtiendo todos los caracteres en minúsculas. En la fase de extracción de características estos códigos fuente se transforman a una representación vectorial para compararlos con la colección R y calcular la similitud del coseno. En la figura 4.12 se muestra el esquema de la fase de extracción de características. Es importante

destacar que cada código fuente de la colección R ha sido preprocesado de la misma forma que la colección de códigos C. Como resultado de esta fase se dispone de una representación vectorial de cada código fuente. El tamaño de este nuevo espacio vectorial se corresponde al tamaño de la colección R, siendo cada dimensión la similitud con cada elemento de esta. Finalmente, en la fase de similitud se realiza la comparación de cada par de vectores que representan a los códigos fuente de C mediante la similitud del coseno. El resultado de esta fase es un listado de pares de códigos fuente junto al valor de similitud estimado entre ambos códigos.

La comparación entre códigos fuente escritos en distintos lenguajes de programación se realiza siguiendo los mismos pasos que a nivel monolingüe. El único cambio respecto a la comparación monolingüe es que la colección R varía dependiendo del lenguaje de programación siendo siempre una colección comparable. Por ejemplo, dos códigos fuente c y c' escritos en los lenguajes de programación LP y LP' respectivamente, siendo $LP \neq LP'$, se comparará el código fuente c con la colección R_{LP}, mientras que el código c' se comparará con la colección $R_{LP'}$. Ambas comparaciones tendrán como resultado vectores del mismo tamaño ($\overrightarrow{c}$ y $\overrightarrow{c'}$). Se espera que un código fuente c comparado contra una colección R de problemas resueltos obtenga valores de similitud cercanos a los de una implementación de la misma solución c' escrita en otro lenguaje de programación comparada con otra colección comparable R' de problemas resueltos escrita en el lenguaje de programación de esta segunda implementación.

4.7 SoCo-ASA: Modelo basado en alineamiento de palabras

Este modelo (Barrón-Cedeño, 2010) se basa en los principios de traducción automática estadística, donde se combina un modelo de traducción y un modelo del lenguaje para generar la traducción más probable de un texto (Brown et al., 1993). En el modelo SoCo-ASA (en inglés *Source Code Alignment based Similarity Analysis*), el modelo de lenguaje se sustituye por un modelo de longitud. El modelo de longitud determina la probabilidad de que un documento se considere traducción de otro según la proporción de la longitud esperada para su traducción a otro lenguaje. El modelo de longitud propuesto por (Pouliquen et al., 2003) se define como:

$$lf(d, d') = e^{-0{,}5\left(\frac{len_d/len_{d'} - \mu}{\sigma}\right)}, \tag{4.11}$$

donde μ y σ son la media y la desviación típica de la diferencia de las longitudes de los documentos traducidos del lenguaje L al L'. De hecho, este modelo calcula una distribución normal con un valor máximo de la media de 1. En el caso de que un documento candidato a ser la traducción de otro documento no tenga la longitud esperada, el factor de normalización penalizará su similitud reduciéndola. Para poder aplicar el modelo en el caso de códigos fuente se deben disponer de una colección de códigos paralela a partir de la cual se calculen los parámetros μ y σ. En la tabla 4.3 se muestran los valores de μ y σ estimados para los lenguajes de programación considerados. Estos parámetros se han estimado a partir de la colección de códigos fuente paralela descrita en el apartado 3.2.2. En resumen, el modelo de longitud tiene por objetivo determinar si la longitud de un documento, o fragmento de un documento, corresponde con la longitud esperada de su traducción.

Parámetro	C→Java	Java→C	Java→Python	Python→Java	C→Python	Python→C
μ	0,92469	1,46322	1,00313	1,00824	0,98885	1,56418
σ	0,50461	3,08911	0,07742	0,08280	0,79928	2,59504

Tabla 4.3: Factor de longitud estimado para cada par de lenguajes de programación. Un valor de $\mu > 1$ para c y su traducción c' implica que $len(c) < len(c')$ donde la longitud se ha medido en caracteres.

El objetivo del modelo de traducción es estimar como de probable es que las palabras del documento (o fragmento) traducido sean traducciones válidas del documento original. Para aplicar el modelo de traducción es necesario disponer de un diccionario estadístico bilingüe. Este diccionario estadístico contiene estimaciones de probabilidad de que una palabra sea una traducción válida de otra; normalmente, se estima utilizando corpus paralelos alineados a nivel de oración. Para estimar las probabilidades de traducción se puede utilizar la extensión del modelo 1 de IBM que se utiliza en Pinto et al. (2009), donde se asume la misma probabilidad para las posiciones de las palabras; por ello el orden de las palabras no afecta al resultado de la similitud (Brown et al. (1993), pág. 268). Como un código fuente se puede traducir de múltiples modos y siguiendo distintas estrategias, con el modelo 1 de IBM evitamos que el cambio de posición de las palabras influya para determinar una buena probabilidad en una traducción[4].

En el trabajo Pinto et al. (2009) se propone una adaptación del modelo de traducción, originalmente destinado a trabajar con oraciones, para estimar similitud entre documentos. Esta adaptación se define como:

$$w(d|d') = \sum_{x \in d} \sum_{y \in d'} p(x, y) \ , \tag{4.12}$$

donde $p(x, y)$ representa las entradas del diccionario estadístico bilingüe. Esta nueva medida ya no es probabilística y por lo tanto ya no está acotada en el intervalo $[0-1]$. Según esta nueva medida, a mayor tamaño de los documentos d y d', mayor será su similitud. La similitud entre dos documentos d y d' será el resultado de la combinación de este modelo de traducción adaptado y el modelo de longitud:

$$\varphi(d, d') = lf(d, d') \cdot w(d|d') \ , \tag{4.13}$$

que reduce el impacto negativo del modelo de traducción adaptado por no estar acotado.

Para poder utilizar este modelo en código fuente se necesitan una serie de adaptaciones del modelo aplicado a textos. En la construcción de corpus paralelos de textos se toma como unidad de alineamiento la oración. El equivalente a la oración en códigos fuente es la instrucción. En la tabla 4.4 se muestra un ejemplo de códigos fuente alineados a nivel de línea. Sin embargo, en códigos fuente el programador tiene la libertad de cambiar partes del código fuente de posición sin alterar el comportamiento de este. Esto añade una alta dificultad al etiquetado manual de códigos fuente a nivel de

[4]Los modelos del 2 al 5 IBM tienen en cuenta el orden de las palabras para determinar una traducción, por lo que pueden influir negativamente en el cálculo de la similitud de las traducciones (Brown et al., 1993)

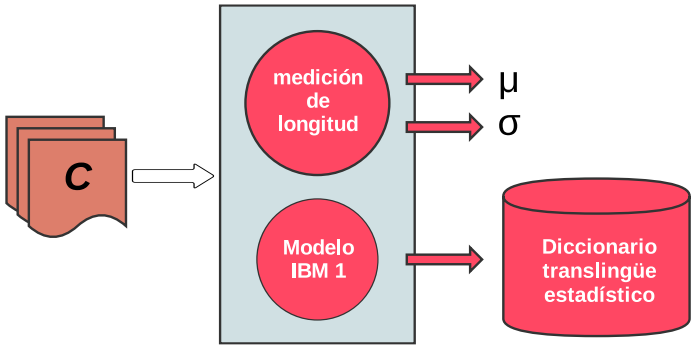


Figura 4.13: Esquema de entrenamiento del modelo SoCo-ASA.

instrucción. En este trabajo de investigación se considera el alineamiento a nivel de código fuente completo para reducir la alta complejidad de la anotación manual. En la fase de entrenamiento, para aprender el diccionario estadístico solo se considera parte del vocabulario del código fuente, en concreto, operadores, identificadores y las palabras reservadas del lenguaje de programación. Además, los identificadores se sustituyen por la etiqueta genérica *id* y las cadenas por *string* debido a la libertad del programador de utilizar prácticamente cualquier cadena de caracteres posible. El diccionario estadístico bilingüe se ha aprendido a partir de la colección de códigos fuente paralela descrita en el apartado 3.2.2 utilizando la herramienta GIZA++ (Och et al., 2003). En la figura 4.13 se muestra el esquema de la fase de entrenamiento de este modelo.

Código fuente c_1 (lenguaje C)

```
for (int i = 1; i <= 100; i++) {
    if ((i %15) == 0)
        cout << "FizzBuzz\n";
    else if ((i %3) == 0)
        cout << "Fizz\n";
    else if ((i %5) == 0)
        cout << "Buzz\n";
    else
        cout << i << "\n";
}
return 0;
```

Código fuente c_2 (lenguaje Java)

```
for(int i= 1; i <= 100; i++){
    if(i %15 == 0){
        System.out.println("FizzBuzz");
    }else if(i %3 == 0){
        System.out.println("Fizz");
    }else if(i %5 == 0){
        System.out.println("Buzz");
    }else{
        System.out.println(i);    }
}
return 0;
```

Tabla 4.4: Ejemplo de dos códigos fuente c_1 y c_2 alineados a nivel de línea.

Por otra parte, se proponen cuatro propuestas diferentes para la creación del diccionario estadístico del modelo de traducción aplicado a códigos fuente: *(i)* aprender el diccionario estadístico a partir de un corpus paralelo de códigos fuente utilizando el modelo 1 de IBM; *(ii)* aprender el diccionario como en *(i)* y filtrar las traducciones más probables con una masa de probabilidad del 20 %; *(iii)* mismo aprendizaje y filtrado que en *(ii)* pero utilizando una masa de probabilidad del 40 %; y *(iv)* utilizar una tabla de conversiones entre términos de los lenguajes de programación ya

existente.[5] Experimentalmente se comprobó que la propuesta que obtiene mejores resultados es la que filtra las entradas del diccionario estadístico con una masa de probabilidad del 40 %.

En la tabla 4.5 se muestra un ejemplo de un diccionario estadístico calculado a partir de dos códigos fuente en C y Java. Se puede comprobar que tokens que aparecen en las mismas líneas se les asigna la misma probabilidad de ser una traducción como por ejemplo los tokens for y int. Cuando un token co-ocure con otro token en varias líneas, el modelo 1 de IBM es capaz de detectar esta relación, como por ejemplo en los tokens if o *cout*. Es importante mencionar que este diccionario se ha estimado en el sentido $c_1 \rightarrow c_2$, ya que si se calculara el diccionario en el sentido contrario, el diccionario resultante será distinto. Esto también ocurre en el modelo de longitud donde la relación de longitudes de códigos fuente en un sentido será mayor que 1 y en el sentido contrario inferior. Por lo tanto, si se quiere realizar una detección de reutilización en el sentido contrario es necesario crear un nuevo diccionario y distinguir en qué sentido se está realizando la comparación con el modelo SoCo-ASA.

Palabra c_1	Palabra c_2	Probabilidad	Palabra c_1	Palabra c_2	Probabilidad
for	for	0,331	λ	λ	0,331
for	int	0,331	if	if	0,999
for	i	0,006	cout	System.out.println	1,000
for	λ	0,331	“FizzBuzz\n”	System.out.println	0,003
int	for	0,331	“FizzBuzz\n”	“FizzBuzz”	0,997
int	int	0,331	else	else	1,000
int	i	0,006	“Fizz\n”	System.out.println	0,003
int	λ	0,331	“Fizz\n”	“Fizz”	0,997
i	i	1,000	“Buzz\n”	System.out.println	0,003
λ	for	0,331	“Buzz\n”	“Buzz”	0,997
λ	int	0,331	return	return	1,000
λ	i	0,006			

Tabla 4.5: Ejemplo de diccionario estadístico estimado a partir de los códigos fuente c_1 al c_2 de la tabla 4.4 sin considerar los símbolos de puntuación ni las constantes numéricas alineando los códigos a nivel de línea. El símbolo λ representa la cadena vacía. Como el token *int* solo aparece en la misma línea que for, el modelo 1 de IBM estima que tiene la misma probabilidad de ser la traducción del token.

Así pues, una vez completada la fase de entrenamiento se dispone de un diccionario estadístico entre lenguajes de programación para el modelo de traducción y de los parámetros μ y σ para el modelo de longitud. Para comparar una colección de códigos fuente C con el modelo SoCo-ASA se sigue el mismo esquema que en el resto de modelos propuestos (ver figura 4.1). En la fase de preproceso se extraen todos los tokens del código fuente incluidos los elementos sintácticos como paréntesis, delimitadores, llaves, etc. y todos los caracteres son convertidos a minúsculas. Las cadenas son reemplazadas por el token *string* y aquellos tokens que no sean palabras reservadas del lenguaje de programación ni signos de puntuación son reemplazados por el token *id*. A continuación, en una fase de extracción de características se calcula el número de tokens en el listado que representa a cada código fuente. Estas cantidades representarán a cada código fuente en el modelo de longitud de la fase de estimación de similitud. En la fase de estimación de similitud, para cada par de códigos

[5] La tabla de conversión utilizada se ha extraído del artículo en la Wikipedia en `http://en.wikipedia.org/wiki/Comparison_of_programming_languages_(basic_instructions)`.

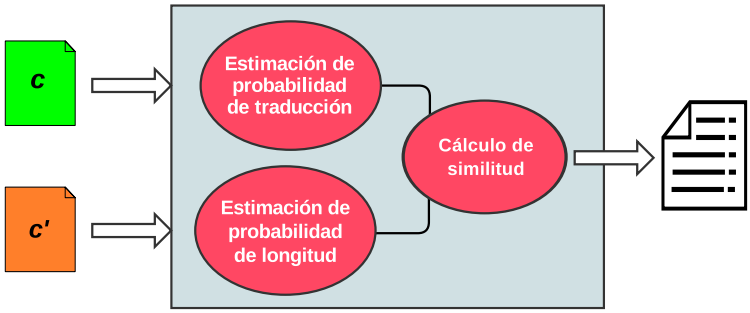


Figura 4.14: Esquema de la fase de estimación de la similitud del modelo SoCo-ASA. Cuando todos los pares de códigos fuente de una colección han sido procesados, el resultado es un listado de pares de códigos fuente junto al valor de similitud entre estos.

fuente se realizan dos operaciones para calcular la similitud final, como se muestra en la figura 4.14. Por una parte se encuentra el modelo de longitud, que con las longitudes de los códigos fuente de la fase anterior junto con los parámetros μ y σ de la fase de entrenamiento compara las proporciones de tamaño de cada par de códigos fuente con la media y desviación típica aprendidos (μ y σ) con el objeto de establecer una medida de similitud basada en su longitud (véase ecuación 4.11). Por otra parte, con los tokens extraídos en la fase de preproceso y el diccionario estadístico de la fase de entrenamiento se calcula la "probabilidad" de que un código fuente sea traducción del otro (véase ecuación 4.12). Es importante recordar que este cálculo ya no es una probabilidad por la adaptación del modelo para comparar documentos. El valor de similitud entre dos códigos fuente se define como el producto escalar de los resultados de estos modelos como se define en la ecuación 4.13. Así pues, como ocurre en el resto de modelos, al aplicar esta fase a todos los pares de códigos fuente de la colección C se obtiene como resultado un listado de pares de códigos fuente junto al valor de similitud estimado para cada par.

4.8 Conclusiones

En este capítulo se ha mostrado una relación de modelos desarrollados en el marco de esta tesis. Estos modelos pueden aplicarse indistintamente a nivel monolingüe o translingüe, es decir, nos permiten realizar comparaciones entre casi cualquier par de lenguajes de programación a diferencia de las propuestas del estado de la cuestión. Estos modelos serán evaluados y sus resultados comparados en diferentes escenarios, tanto a nivel monolingüe como translingüe, en el capítulo 5.

Capítulo 5

Experimentación

En este capítulo se resume la experimentación que hemos realizado utilizando los recursos desarrollados descritos en el capítulo 3 para el ajuste y evaluación de los modelos propuestos en el capítulo 4. Los experimentos están divididos por tipo de reutilización de código fuente (monolingüe o translingüe) y dentro de estos dos grandes apartados separados por recursos utilizados.

5.1 Evaluación en escenarios monolingües

En función de la homogeneidad o heterogeneidad de la colección se pueden distinguir escenarios donde se puede tratar de identificar casos de reutilización de código fuente monolingüe. El primero de estos escenarios consiste en, dado un código fuente c sospechoso, encontrar dentro de una colección heterogénea de códigos fuente C el código(s) fuente c_r origen de la reutilización. Como colección heterogénea se entiende un conjunto de códigos fuente que no tienen origen en común, ni su objetivo es realizar las mismas funciones. Un ejemplo de colección heterogénea es un repositorio público de software donde cada programador crea sus propios proyectos y no tienen ninguna relación con proyectos de otros programadores.

El segundo escenario consiste en, dado un código fuente c sospechoso, encontrar dentro de una colección homogénea de códigos fuente C el códigos(s) fuente origen c_r de la reutilización. La colección homogénea consiste de códigos fuente que dan solución a un mismo problema. Los códigos fuente a entregar de una asignatura en la que todos los estudiantes tienen que resolver el mismo problema es un ejemplo de este tipo de colección. Este escenario se considera más complejo debido a que el grado de similitud entre todos los códigos fuente pueden aumentar los casos de falsos positivos.

En los siguientes experimentos el objetivo es identificar qué modelos permiten encontrar más casos de reutilización de código fuente en colecciones homogéneas y, a su vez, que sean capaces de descartar aquellos códigos fuente que no han sido reutilizados pero que comparten un cierto grado de similitud. En concreto, el primer experimento estudia el impacto que tienen las modificaciones realizadas por los estudiantes sobre los identificadores, proporcionando los valores de similitud y posicionamiento

en el ranking para distintos tipos de preproceso de identificadores. En el segundo experimento se comprueba que el modelo SoCo-NG de detección de reutilización de códigos fuente basado en n-gramas es aplicable en entornos de programación a gran escala. Además, también se compara este modelo con la herramienta JPlag, la más utilizada en las comparaciones en el estado de la cuestión. En el tercer experimento se presentan los ajustes realizados en los distintos modelos propuestos, la comparación de los mismos en términos de eficacia y finalmente se propone el ensamble de estos modelos que aprovechará en cada caso las bondades de los mismos.

5.1.1 Impacto de cambios en identificadores

El propósito de este experimento a nivel monolingüe consiste en estudiar el impacto que tienen los identificadores tales como nombres de variables, funciones, etc. y los diferentes preprocesos de los mismos en un sistema de detección automática de reutilización de códigos fuente. Por lo tanto, en este experimento trataremos de encontrar si algún tipo de preproceso sobre los identificadores es conveniente para mejorar la detección de reutilización en código fuente. Para ello, se ha utilizado el corpus ILN (véase el apartado 3.1.2) que consta de una colección de códigos escritos en Python que resuelven diversas tareas académicas. Cuando se escribe código fuente se usa un vocabulario restringido por el lenguaje de programación para escribir las instrucciones que el programa resultante deberá ejecutar. Además, se usarán identificadores para nombrar variables, funciones, clases, métodos que permiten distinguir y hacer referencia a un espacio en memoria, un bloque de instrucciones, etc. Los identificadores en el código fuente son un mecanismo utilizado por el programador para hacer referencia y más legible una parte del código fuente. Estos identificadores los crea el programador con casi total libertad dentro de unas restricciones tipo que imponen los lenguajes de programación como son que no se pueda incluir el carácter espacio en blanco, que haya distinción entre letras mayúsculas y minúsculas, o que no se pueda utilizar palabras reservadas como identificadores, entre otras. Por ejemplo, en el lenguaje Java un identificador debe empezar por una letra, un guión bajo (_), o un símbolo monetario Unicode[1] (véase los ejemplos $, £, €). A esta letra inicial le puede seguir cualquier dígito, letra, guión bajo o símbolo monetario. Por convenio entre programadores, los símbolos monetarios (especialmente el dólar) están reservados para identificadores automáticamente generados por el compilador y algún tipo de procesador de código.

La libertad de combinación de caracteres para crear los identificadores permite al programador cambiarlos por cualquier otra secuencia de caracteres válida evitando que se asemeje al código fuente original reutilizado. En este experimento se va a medir el impacto que tienen los identificadores en la detección de reutilización de código fuente, estudiando las modificaciones más frecuentes por parte del programador para disimular y evitar ser detectado, ya sea por sistemas automáticos o revisados manualmente.

El corpus ILN es una colección del ámbito académico con 183 códigos fuente escritos en Python compuesto por 5 tareas que incluyen casos reales de pares reutilizados. Las tareas 2 y 4 se resolvieron individualmente mientras que las tareas 1, 3 y 5 se desarrollaron en grupos. Como ya se mencionó en el apartado 3.1.2 se ha analizado manualmente cada caso y se ha observado el tipo de reutilización realizado por los estudiantes. En unos casos se trata de copia exacta y en otros se llevan a cabo algunas modificaciones para que la copia no sea evidente.

[1] Accedido el 30 de Octubre de 2015 `http://unicode.org/charts/PDF/U20A0.pdf`.

En esta experimentación se evalúa cada par reutilizado en términos del valor de similitud y de la posición que ocupa en el ranking de pares de códigos fuente ordenado según dicho valor de similitud. Para calcular la similitud de manera automática se ha utilizado el modelo SoCo-NG con valor $n = 3$ con distintos preprocesos respecto al tratamiento de los identificadores: *(i)* considerar los identificadores tal como han sido escritos en el código fuente; *(ii)* eliminar los identificadores; *(iii)* reemplazar los identificadores por las siguientes *wildcards*[2] "#", "##" y "###". Empleamos tres tipos distintos de wildcard para comprobar el efecto que produce el incrementar el número de caracteres de este sobre el modelo SoCo-NG basado en n-gramas de caracteres.

Por cada tarea se ha construido una tabla donde se muestran los resultados correspondientes a cada par reutilizado indicando su posición en el ranking y, entre paréntesis, su valor de similitud. Estos resultados se presentan de forma tabular con siete columnas: *(i)* "Casos" de pares con reutilización; *(ii)* "Con ident." que muestra los resultados tomando los identificadores; *(iii)* "Sin ident." que los muestra eliminando identificadores; *(iv)* "#", "##" y "###" que corresponden a la sustitución de los identificadores por una de estas cadenas como wildcard; y *(v)* en la última columna "Tipo" se resume el tipo de cambio que se produce en el par reutilizado.

La tarea 1 requiere un número de líneas de código pequeño por lo que existe una alta similitud entre todos los códigos fuente. Esta alta similitud provoca un alto número de falsos positivos y, por lo tanto, dificulta la selección correcta de casos reutilizados. En la tabla 5.1 se muestran los resultados del análisis sobre esta tarea 1. En el par reutilizado 2, se cambiaron todos los identificadores y las instrucciones por otras equivalentes (por ejemplo, $var = var + 100$ por $var+ = 100$). Además, se tradujeron cadenas de texto del español al inglés e incluso se aprecian cambios procedurales. El cambio de los identificadores por parte del estudiante introduce ruido y disminuye la similitud utilizando el modelo SoCo-NG. Cuando no se consideran los identificadores, o estos se sustituyen por una wildcard, se observa que la similitud es mayor y por lo tanto con este tipo de preproceso el modelo ya no es sensible a este tipo de cambios. Cabe resaltar que el uso de wilcard incrementa la similitud entre los pares de códigos fuente al contener más n-gramas en común. Por otra parte, otro tipo de modificaciones sobre el código fuente distinto al cambio de identificadores afecta negativamente a la detección utilizando una wilcard o sin considerar los identificadores como en el caso de los pares reutilizados 2 y 4. Esto se debe a que la similitud entre los códigos fuente con otro tipo de cambios viene dada precisamente por los identificadores.

Caso	Con ident.	Sin ident.	'#'	'##'	'###'	Tipo
1	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
2	150 (0,60)	46 (0,65)	94 (0,91)	76 (0,98)	60 (0,99)	Cambios de ident. y traduc.
3	2 (0,87)	13 (0,74)	5 (0,96)	10 (0,99)	9 (0,99)	Cambios en cadenas y elimina cód.
4	5 (0,85)	18 (0,73)	18 (0,95)	19 (0,99)	24 (0,99)	Cambios de ident. y posición

Tabla 5.1: Posición en el ranking y valor de similitud correspondiente a los pares reutilizados de la tarea 1 aplicando diferentes preprocesos.

En la tabla 5.2 se muestran los resultados sobre la tarea 2. Se encontraron tres casos de reutilización sin modificaciones por lo que no afecta el uso o no de identificadores, ni tampoco el uso de wilcard

[2] La definición de wildcard es un caracter o caracteres comodín que representa cualquier otro caracter o cadena de caracteres. Los caracteres comodines más frecuentes en la informática son: *(asterisco), %(por ciento), _(guion bajo), ?(signo de interrogación). Sin embargo en expresiones regulares el carácter comodín por excelencia es el "." (punto). Definición de wildcard en la Wikipedia: `https://en.wikipedia.org/wiki/Wildcard_character`

porque el resto del código fuente se mantiene idéntico, la reutilización sigue siendo completa y por lo tanto su similitud es máxima (valor 1). En los pares 4 y 5, solo se modificaron los identificadores, por lo cual, al eliminar los identificadores o usar wildcard ha mejorado ligeramente su posición en el ranking y se obtiene un valor de similitud muy cercano a 1.

Caso	Con ident.	Sin ident.	'#'	'##'	'###'	Tipo
1	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
2	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
3	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
4	4 (0,90)	4 (0,96)	4 (0,99)	3 (0,99)	3 (0,99)	Cambios de ident. y posición
5	6 (0,89)	1 (0,99)	3 (0,99)	4 (0,99)	4 (0,99)	Cambios de identificadores

Tabla 5.2: Posición en el ranking y valor de similitud correspondiente a los pares reutilizados de la tarea 2 aplicando diferentes preprocesos.

En la tarea 3 (véase tabla 5.3), se encuentran dos pares de códigos fuente que son copia exacta, tres donde solo se modificaron los identificadores y un par en el que además de modificar los identificadores también se cambió de posición partes del código fuente. En los pares revisados manualmente como copia exacta no existe ningún cambio en cuanto a su similitud y su ranking porque con el uso o no de los identificadores los códigos se mantienen exactamente igual. Los pares de códigos fuente con cambios únicamente en los identificadores mantienen una posición parecida a la obtenida con identificadores y su similitud aumenta hasta valores muy próximos a 1. En el par de códigos fuente donde se realizaron cambios en los identificadores y de posición de partes del código se ha comprobado manualmente que se ha eliminado parte del código fuente original y que el código no es funcional, es decir, no se puede ejecutar porque se han introducido errores. Esto implica que el modelo SoCo-NG que compara códigos fuente a nivel de documento pierda similitud por falta de parte del código. Además, al utilizar wildcard se elimina parte de la similitud de aquellos identificadores que no se modificaron. Así como se ha observado en anteriores tareas, el uso de wildcard ayuda a identificar mejor aquellos casos de reutilización que solo tienen cambios de identificadores, mientras que empeoran los resultados cuando hay otro tipo de cambios como de posición o división/agrupación de funciones.

Caso	Con ident.	Sin ident.	'#'	'##'	'###'	Tipo
1	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
2	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
3	2 (0,88)	2 (0,99)	2 (0,99)	2 (0,99)	2 (1,00)	Cambios de identificadores
4	8 (0,80)	88 (0,70)	140 (0,81)	111 (0,96)	97 (0,99)	Cambios de ident. y posición
5	9 (0,79)	3 (0,92)	4 (0,95)	5 (0,99)	4 (0,99)	Cambios de identificadores
6	3 (0,89)	5 (0,87)	7 (0,92)	11 (0,98)	7 (0,99)	Cambios de identificadores

Tabla 5.3: Posición en el ranking y valor de similitud correspondiente a los pares reutilizados de la tarea 3 aplicando diferentes preprocesos.

En la tabla 5.4 se muestran los resultados de la tarea 4. La solución de esta tarea no requiere una gran cantidad de líneas de código fuente y se puede considerar sencilla dado que la parte de código introducida por parte del alumno es muy pequeña. Todos los alumnos deben utilizar en el código fuente unas cadenas específicas proporcionadas por los profesores. Por ello, existe una alta similitud

entre todos los documentos de la colección y resulta más complicado discernir entre si son un caso de reutilización o no. Como en las tareas anteriores, SoCo-NG detecta con similitud 1 aquellos pares de códigos que son copias exactas; además, las modificaciones sobre los identificadores no afectan a su detección. Por otra parte, los pares de códigos fuente cuyos identificadores han sido modificados empeoraron su posición en el ranking al sustituir los identificadores por una wilcard. En primer lugar, al eliminar identificadores en esta tarea se está eliminando la parte que ha introducido el alumno dentro del código fuente por lo que el código restante es prácticamente idéntico entre los códigos fuente y da lugar a un alto número de falsos positivos. En segundo lugar, al reemplazar los identificadores por una wilcard se introduce el mismo caracter o caracteres adicionales en todos los códigos fuente aumentando su similitud. A mayor número de caracteres utilizados como wildcard, el valor de similitud de todos los pares de códigos será mayor y resulta más complicado distinguir entre los casos de reutilización. Por ejemplo, en los pares reutilizados 2, 3 y 4, la similitud de los códigos fuente no sufre mucha variación, pero el incremento de similitud en los falsos positivos causa que estos pares cambien su posición en el ranking.

Caso	Con ident.	Sin ident.	'#'	'##'	'###'	Tipo
1	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
2	2 (0,98)	7 (0,97)	5 (0,97)	6 (0,97)	54 (0,98)	Cambios en identificadores
3	3 (0,97)	29 (0,84)	214 (0,70)	417 (0,72)	778 (0,95)	Cambios en identificadores
4	5 (0,97)	19 (0,95)	10 (0,95)	21 (0,95)	303 (0,96)	Cambios en identificadores

Tabla 5.4: Posición en el ranking y valor de similitud correspondiente a los pares reutilizados de la tarea 4 aplicando diferentes preprocesos.

La tarea 5 es donde se han encontrado mayor número de casos de reutilización. En el par de códigos fuente 7 la copia es prácticamente exacta, salvo que se han modificado ligeramente las cadenas y se ha eliminado un fragmento pequeño de código fuente. Al no existir cambio de identificadores, la similitud se mantiene prácticamente igual, solo aumenta cuando se incrementa el número de caracteres de la wildcard. En esta tarea se ha encontrado el caso particular de reutilización con copia exacta entre tres pares de códigos fuente, por lo que los tres primeros casos de reutilización en realidad es el mismo caso pero entre tres códigos fuente. En el par reutilizado 5 se aprecia que al eliminar los identificadores aumenta la similitud. Sobre este par se han combinado funciones del código fuente, por lo que existe duplicidad de variables en la combinación. Al eliminar los identificadores se mantiene la estructura del código fuente y las palabras reservadas disminuyendo el ruido introducido por la duplicidad de identificadores.

Caso	Con ident.	Sin ident.	'#'	'##'	'###'	Tipo
1	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
2	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
3	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
4	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	1 (1,00)	Copia exacta
5	10 (0,85)	3 (0,93)	3 (0,94)	4 (0,98)	10 (0,99)	Combina funciones
6	7 (0,92)	6 (0,89)	6 (0,92)	6 (0,97)	12 (0,99)	Elimina parte código
7	5 (0,96)	5 (0,95)	5 (0,96)	5 (0,99)	5 (0,99)	Elimina parte código

Tabla 5.5: Posición en el ranking y valor de similitud correspondiente a los pares reutilizados de la tarea 5 aplicando diferentes preprocesos.

En conclusión, los experimentos anteriores muestran que la utilización de wildcard puede aumentar la similitud y situar en una posición más alta en el ranking los pares de códigos fuente reutilizados si solo se han modificado los identificadores. Además, en aquellos pares reutilizados donde se ha realizado otro tipo de modificaciones como combinar funciones, cambiar la estructura, etc. se empeoran los resultados con el uso de wildcard, así como sin considerar los identificadores. El uso de wilcard aumenta la similitud entre aquellos códigos fuente que no son casos de reutilización ya que se eliminan los identificadores que los diferenciaban y se introduce una wildcard común. Por lo tanto, podemos concluir que el preproceso propuesto para el módulo SoCo-NG en el cual se incluyen los identificadores es el más conveniente, además de permitir que la aplicación del modelo sea independiente de los lenguajes de programación.

5.1.2 Detección de reutilización en un entorno masivo

En este apartado se presentan dos experimentos realizados sobre un entorno masivo de programación. En concreto, la competición de programación Google Code Jam descrita en el apartado 3.2.1. El objetivo del primer experimento consiste en analizar las distribuciones de similitud entre códigos fuente en los lenguajes C/C++, Java y Python. El propósito del segundo experimento consiste en detectar posibles casos de reutilización en la competición entre participantes. En ambos experimentos se ha realizado un cálculo exhaustivo requiriendo un elevado número de comparaciones entre pares de códigos fuente. Por ejemplo, solo para el problema B se calculó más de 34,6 millones de comparaciones entre códigos fuente.

Análisis de los rangos de similitud

En este experimento el objetivo consiste en observar cómo se distribuyen la similitudes de los pares de códigos fuente en intervalos e identificar si existen diferencias para cada lenguaje de programación. Para el cálculo de las similitudes se utilizará el modelo SoCo-NG dado que tiene que procesar un alto número de códigos fuente. En este experimento no hemos considerado el resto de modelos por los costes computacionales más elevados de estos, además de que algunos modelos (SoCo-LSA, SoCo-ESA...) requieren un entrenamiento previo. Estos intervalos de similitud pueden ayudar a establecer un umbral para distinguir entre los casos potenciales de reutilización de los que no lo son.

En la figura 5.1 se muestran las distribuciones de similitud de los pares de códigos fuente para cada problema y lenguaje de programación estudiado. En los problemas de complejidad media, como el problema B y C (véase el apartado 3.2.1), la mayoría de pares de códigos en Java tienen su similitud en el rango 0,4-0,6. En el problema de mayor complejidad, el problema D, la mayoría de códigos fuente están en el rango de similitud 0,2-0,3. Las soluciones de los problemas de menor dificultad tienden a tener una mayor similitud entre ellos. Los programas escritos en Python muestran esta misma tendencia. Los pares de códigos fuente más similares en Python están en el rango 0,2-0,4 para los problemas B y C, mientras que para el problema D están en el rango 0,1-0,3. Este comportamiento también se observa para el lenguaje C/C++ pero en menor grado: para los problemas B y C la mayoría de códigos están en el rango 0,2-0,5, mientras que para el problema D están en torno a 0,2-0,4.

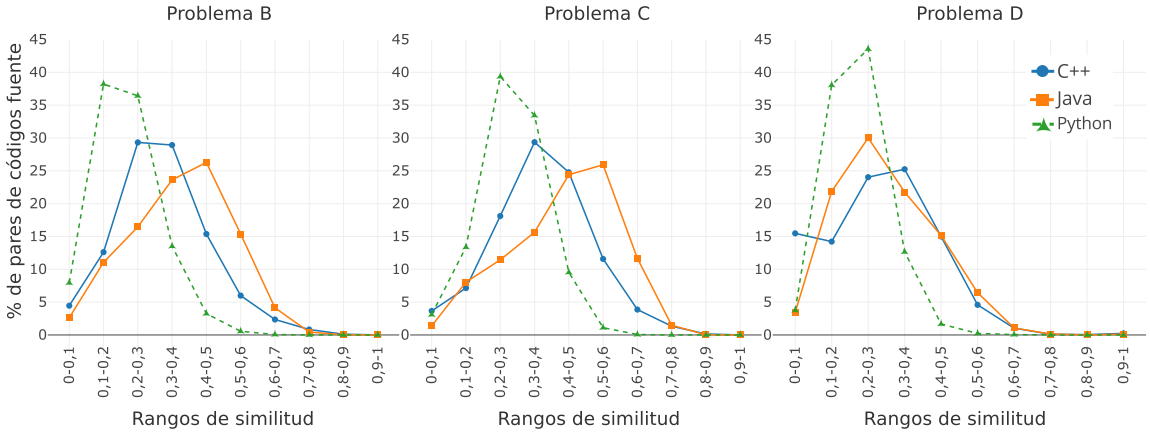


Figura 5.1: Distribución de la similitud de los pares de códigos fuente por problema y lenguaje de programación.

La tendencia muestra que el intervalo donde se aglutina la mayoría de códigos fuente se sitúa en torno a un valor de similitud menor cuando la complejidad del problema es alta. Los mayores valores de similitud ocurren entre códigos fuente escritos en Java, seguidos de C/C++ y finalmente Python. Por lo tanto, dos códigos fuente escritos en Java puede compartir más fragmentos en común sin implicar necesariamente reutilización. Esta información es particularmente útil para establecer umbrales de decisión para detectar los casos de reutilización para cada lenguaje de programación.

Búsqueda de casos de reutilización en código fuente

El objetivo de esta experimentación es comprobar si los modelos de detección de reutilización son útiles en entornos de programación a gran escala. Se han analizado códigos fuente de 6 tareas diferentes en los tres lenguajes más utilizados en la competición Google Code Jam (C/C++, Java y Python). La cantidad total de códigos fuente considerada fue de $34{,}3K$; dando como resultado más de 58,6 millones de comparaciones de pares de códigos fuente.

Después de obtener el ranking de las similitudes de los pares de códigos fuente para cada lenguaje de programación y tarea, se recuperaron los 20 pares más similares en cada uno de ellos y fueron analizados manualmente por tres programadores con fluidez en los tres lenguajes de programación (es decir, 360 instancias: 20 pares de códigos fuente $\times$ 6 tareas $\times$ 3 lenguajes de programación). El propósito del análisis manual fue para descubrir casos de reutilización.

En la tabla 5.6 se muestra el número de casos de reutilización detectados por SoCo-NG. Esta incluye el número total separado por cada tarea y lenguaje de programación detectados entre los 20 más similares. Es importante destacar que las tareas de mayor complejidad obtienen un número menor de casos de reutilización: se encontraron más casos en las tareas de menor complejidad (B_s, B_l y C_s)[3]. En general, se encontraron menos casos de pares de códigos fuente reutilizados en Python que

[3]Los problemas propuestos (B, C y D) se encuentran divididos en tareas (s y l) dependiendo del tamaño de entrada de datos que los problemas tienen que procesar.

en C/C++ y Java. Existe una diferencia grande entre las tareas C_s y C_l (exactamente el mismo problema, pero con mayor complejidad) en el lenguaje Python, que puede ser explicada por el alto coste temporal requerido para resolver la versión más compleja del problema. Como anteriormente se comentó, es posible que los participantes tengan que programar una versión más eficiente para poder cumplir con la restricción de tiempo en el envío de la tarea C_l (esto probablemente no fue necesario para los otros dos lenguajes en este problema). No se descarta la posibilidad de que explorando manualmente más allá de los 20 pares más similares se encuentren más casos de reutilización. Se ha comparado el desempeño de SoCo-NG en las 6 tareas frente a JPlag (Prechelt et al., 2002).

Lenguaje	B_s	B_l	C_s	C_l	D_s	D_l	Total
C++	20 (1,00-0,99)	18 (1,00-0,98)	18 (1,00-0,98)	18 (1,00-0,94)	8 (1,00-0,77)	0 (0,71-0,61)	82
Java	18 (1,00-0,99)	18 (1,00-0,99)	20 (1,00-0,99)	20 (1,00-0,96)	1 (0,72-0,65)	1 (0,70-0,57)	78
Python	17 (1,00-0,84)	15 (1,00-0,83)	20 (1,00-0,83)	4 (1,00-0,69)	0 (0,60-0,44)	13 (1,00-1,00)	56
Total por tarea	55	51	58	42	9	1	216

Tabla 5.6: Cantidad de pares reutilizados detectados entre los 20 más similares utilizando el modelo SoCo-NG. Entre paréntesis se muestra el intervalo de similitudes de los 20 pares analizados manualmente.

Como JPlag no soporta el lenguaje Python, se han comparado solo los códigos fuente escritos en C/C++ y Java. En la tabla 5.7 se muestra el número de casos reutilizados detectados por JPlag tras la revisión de los 20 pares de códigos fuente más similares. SoCo-NG ha sido capaz de detectar 37 pares de códigos fuente reutilizados más que JPlag en estos dos lenguajes. Cuando los códigos fuente tienen una longitud grande, es decir, en las tareas de mayor dificultad, ambos modelos muestran un desempeño similar. Los códigos fuente de mayor longitud contienen mayor información (por ejemplo, más funciones, más muestras del estilo de programación, etc.) que permiten una mejor caracterización de los códigos fuente. Ante códigos más cortos (relativamente con mayor similitud que los de mayor longitud), JPlag tiene dificultades para distinguir si dos códigos fuente fueron reutilizados entre sí o no.

Lenguaje	B_s	B_l	C_s	C_l	D_s	D_l	Total
C++	9 (1,00-1,00)	7 (1,00-1,00)	20 (1,00-1,00)	20 (1,00-0,93)	7 (1,00-0,41)	0 (0,32-0,19)	63
Java	13 (1,00-1,00)	13 (1,00-1,00)	16 (1,00-1,00)	16 (1,00-0,98)	1 (0,43-0,31)	1 (0,43-0,35)	60
Total por tarea	22	20	36	36	8	1	123

Tabla 5.7: Cantidad de pares reutilizados detectados entre los 20 más similares utilizando JPlag. Entre paréntesis se muestra el intervalo de similitudes de los 20 pares analizados manualmente.

Se ha realizado una revisión manual de los pares reutilizados detectados con el fin de clasificar las modificaciones introducidas por los participantes en los niveles según Faidhi et al. (1987) (véase el apartado 2.2.1). En la tabla 5.8 se muestran las distribuciones resultantes por niveles. Cabe destacar que los pares de códigos fuente reutilizados pueden contener más de un tipo de modificaciones y por lo tanto pertenecer a varios niveles (un fenómeno frecuente). Todos los casos de reutilización incluyen al menos un fragmento de código fuente exactamente igual a su par (nivel 0). Más del 65 % de los pares contienen cambios en los identificadores (nivel 2). Cambios en los comentarios y en la sangría ocurren con menor frecuencia, con valor cercano al 50 % (nivel 1). En este tipo de escenario, no es habitual que los programadores comenten sus códigos fuente. Sin embargo, deben respetar la sangría en el lenguaje Python. Estas razones causan que el nivel 1 ocurra con menor frecuencia que el nivel 2 para los lenguajes Java y Python, y con frecuencia similar en C/C++.

Nivel	0	1	2	3	4	5	6
C++	100,0	53,9	49,2	15,4	7,7	3,1	0,0
Java	100,0	50,0	82,4	25.7	8,1	5,4	0,0
Python	100,0	59,4	68,8	15,6	0,0	0,0	0,0
Global	100,0	53,2	67,3	19,9	6,4	3,5	0,0

Tabla 5.8: Distribución de las modificaciones realizadas en el código fuente según la clasificación de Faidhi et al. (1987) a partir de los casos de reutilización descubiertos (porcentajes).

Se encontraron en menor cantidad los niveles más altos de modificaciones: 19,9 % (nivel 3), 6,4 % (nivel 4), 3,5 % (nivel 5) y 0,0 % (nivel 6). Por una parte, si existen, estos niveles son más difíciles de detectar, debido a que la similitud entre dos fragmentos llega a ser extremadamente baja. Por otra parte, sigue siendo una cuestión abierta si estos casos se consideran como casos de reutilización. Diferenciar entre casos de reutilización sobre estos niveles altos de modificaciones de código fuente es prácticamente imposible, incluso a veces para un revisor humano.

En conclusión, los experimentos muestran una correlación inversa entre la complejidad de la tarea y la similitud entre los códigos fuente enviados: a menor dificultad del problema se encuentra una mayor similitud entre códigos fuente no relacionados. Por otra parte, los rangos de similitud varían según el lenguaje de programación: los fragmentos de código más similares se encuentran en el lenguaje Java. Esta evidencia se puede considerar para establecer un umbral de similitud y optimizar la recuperación de pares candidatos a posible caso de reutilización. Para cada tarea y lenguaje de programación se recuperaron los 20 pares de código fuente más similares (360 en total) y se revisaron manualmente. En el proceso de revisión se encontraron 216 casos de reutilización entre los recuperados con SoCo-NG (cerca del 60 %). Se encontraron más casos en el lenguaje Java, seguido de cerca por C/C++. SoCo-NG es un modelo independiente del lenguaje, una ventaja frente a herramientas como JPlag que no puede manejar códigos escritos en Python. Ambos modelos mostraron un desempeño similar sobre las tareas más complejas con códigos fuente más largos y, por lo tanto, con mayor información. Sin embargo, SoCo-NG obtiene mejores resultados, cuando los códigos fuente son altamente similares y la variabilidad es menor. La mayoría de fragmentos de código fuente detectados fueron reutilizados mediante copia exacta. Entre los fragmentos ofuscados, las operaciones más comunes aplicadas por los programadores implicaron cambios en los nombres de los identificadores, comentarios y sangría.

5.1.3 Ajuste, comparación y ensamble de modelos

En este apartado se resume una serie de experimentos realizados sobre el corpus A&T++ descrito en el apartado 3.1.3. En concreto, el corpus representa un escenario realista del ámbito académico, donde una tarea propuesta por el profesor debe ser resuelta por el conjunto de alumnos. Todos ellos deben demostrar sus destrezas sobre la misma tarea. Este hecho junto a la relación que pueden tener los estudiantes, hace que la tentación de reutilizar códigos fuente de un compañero sea elevada. A diferencia de la experimentación del apartado anterior (véase los experimentos con el corpus Google Code Jam del apartado 5.1.2), todos los estudiantes comparten profesor, aula, grupo de trabajo y pueden tener cierta relación entre ellos, mientras que en la ronda previa de la competición Google Code Jam, cada participante desconoce de la existencia de la mayoría de participantes. Si además de todos estos factores añadimos que todos los estudiantes han aprendido la misma metodología, la

utilizada por el profesor, resulta inevitable que aunque no exista reutilización, los trabajos de los alumnos vayan a contener un cierto grado de similitud.

Así pues, en estos experimentos ajustamos y comparamos los modelos propuestos para detección de reutilización monolingüe en código fuente. Primeramente, se describe el ajuste de parámetros realizado para el modelo SoCo-Sliding que presenta mejores resultados para, seguidamente realizar una comparación de este con el modelo a nivel de código fuente, SoCo-NG. Una primera experimentación se encuentra resumida en Flores (2012). A continuación, se muestra el ajuste del modelo SoCo-LSA a nivel monolingüe y la comparación de este con el modelo SoCo-NG. Finalmente, comparamos todos los modelos propuestos bajo las mismas medidas, y también se detalla el desarrollo de ensambles de estos modelos. En esta comparación se utilizaran tanto modelos que no requieren corpus de entrenamiento como modelos que sí los requieren.

Experimentación inicial

En este apartado se describe detalladamente la experimentación inicial que fue el origen de esta tesis. En este experimento comparamos los modelos basados en la representación de n-gramas y que tan solo requieren un ajuste de parámetros. En concreto, el modelo SoCo-NG, que ha reportado buenos resultados comparados con la herramienta JPlag en el experimento anterior (véase el apartado 5.1.2), y el modelo SoCo-Sliding, que se espera que con la detección de similitud a través de fragmentos concretos mejore la eficacia del modelo a nivel de código fuente.

El ajuste del parámetro n y la configuración del modelo SoCo-NG se encuentra detallado en el trabajo Flores et al. (2011). En este, se realizaron pruebas variando el tamaño del n-grama, en concreto de 2 a 5 caracteres, se probaron distintos tipos de pesado de los n-gramas, tf y $tf - idf$, y representaciones parciales del código fuente. Estas representaciones parciales consisten en tomar distintas partes del código: *(i)* el código fuente entero sin ser modificado; *(ii)* el código fuente sin considerar los comentarios; *(iii)* solo considerar las palabras reservadas del lenguaje; *(iv)* solo considerar los comentarios contenidos en el código fuente; *(v)* considerar el código fuente sin las palabras reservadas; y *(vi)* considerar el código fuente sin comentarios ni palabras reservadas. Como conclusión se extrajo que para conseguir el mejor rendimiento se debe considerar 3 caracteres como valor n de n-gramas, un pesado de frecuencia de términos tf y tomar todo el código fuente.

Para el ajuste de parámetros del modelo SoCo-Sliding se busca encontrar la combinación que obtenga el mejor rendimiento en la detección de reutilización monolingüe. Además, con el análisis a nivel de fragmento se pretende evitar que si la reutilización ha sido de un fragmento corto y el resto del código no ha sido reutilizado, la similitud global del documento no se vea afectada. Durante el ajuste del modelo SoCo-Sliding a nivel translingüe entre los lenguajes C++ y Java, se pensaba que al ser un par de lenguajes con un alto nivel de similitudes tanto léxicas como sintácticas, los mismos parámetros podrían aplicarse con eficacia en cualquiera de ambos lenguajes de programación a nivel monolingüe. Sin embargo, se ha comprobado que los parámetros estimados con el corpus SPADE a nivel translingüe entre los lenguajes Java–C++ (véase tabla 4.1), no sirven para ser aplicados a nivel monolingüe sobre el corpus CL-AT++ en el lenguaje Java, por lo que se han tenido que esti-

mar otra vez estos parámetros a nivel monolingüe. La mejor combinación de parámetros encontrada para el modelo SoCo-Sliding para el lenguaje Java a nivel monolingüe es de $s = d = 300$ y $t = 0{,}9$.[4]

Para comprobar la efectividad de estos parámetros, se ha realizado una comparación sobre un corpus más grande, en concreto el corpus A&T++, que contiene casos reales de reutilización entre alumnos como se indica en Arwin et al., 2006. Esta comparación se ha realizado entre los modelos SoCo-NG y SoCo-Sliding junto con la herramienta JPlag en el corpus escrito en el lenguaje Java. Debido a 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.

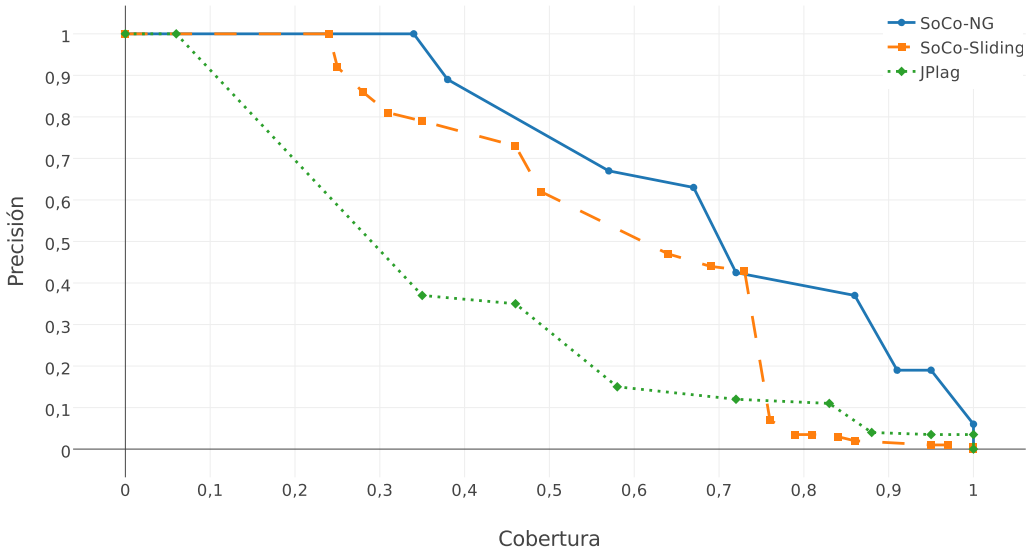


Figura 5.2: Curva de Precisión contra Cobertura comparando los modelos SoCo-NG, SoCo-Sliding y la herramienta JPlag a nivel monolingüe con el corpus A&T++. El modelo SoCo-NG se mantiene por encima del resto de modelos prácticamente en toda la curva.

En la figura 5.2 se muestra la comparación entre el SoCo-NG utilizando tf y trigramas, el modelo SoCo-Sliding con los parámetros estimados para el nivel monolingüe y la herramienta JPlag. Mientras que JPlag mantiene la precisión con valor 1,00 hasta el 0,05 de la cobertura, el modelo SoCo-Sliding y SoCo-NG lo mantienen hasta 0,24 y 0,33 respectivamente. Con el modelo SoCo-NG se obtiene mayor precisión que con los otros dos modelos. El modelo SoCo-Sliding es notablemente más preciso que JPlag hasta alcanzar una cobertura de 0,75. Las dos aproximaciones derivadas de este trabajo de investigación presentan mejores prestaciones que JPlag. Comparando las dos aproximaciones con un caso real de reutilización como es el corpus utilizado, el modelo SoCo-NG ha resultado obtener mejores resultados. Sin embargo, en el caso de querer localizar los fragmentos de código que han sido reutilizados, el modelo SoCo-Sliding será fácilmente adaptable a obtener fragmentos específicos dentro del código fuente. El modelo SoCo-Sliding no ha sido capaz de obtener mejores resultados que el modelo SoCo-NG cuando se trata de detectar reutilización a nivel de

[4]El parámetro s corresponde al tamaño de la ventana, d corresponde al desplazamiento de la ventana y t al umbral que debe superar la ventana para considerarse como reutilización.

código fuente. Además, el alto coste computacional del modelo SoCo-Sliding y la falta de recursos con reutilización a nivel de fragmentos descarta al modelo para experimentos con reutilización a nivel de código fuente.

Ajustes del modelo SoCo-LSA

Como parte de esta tesis, se ha desarrollado un nuevo modelo basado en la representación del código fuente en n-gramas, SoCo-LSA. El modelo SoCo-LSA a través de una fase de entrenamiento es capaz de reducir el espacio de dimensiones seleccionando los códigos fuente más representativos. Una vez representada la colección en el espacio vectorial reducido, la comparación de códigos fuente se realiza seleccionando un código fuente como consulta. Este código fuente consulta se representará en el espacio vectorial reducido con la misma transformación que los de entrenamiento. El resultado de este proceso es un vector de similitudes respecto a los códigos fuente representados en el espacio vectorial, que en este caso es la misma colección. Obviamente, la posición del vector correspondiente al propio código fuente consulta tendrá un valor de similitud sino de 1, muy cercano a este.[5] Este valor no se tiene en cuenta a la hora de la experimentación porque lo que realmente importa es el valor de similitud respecto al resto de la colección. Se espera que a través de la selección de códigos fuente representativos y de la reducción del espacio de características que ofrece LSA se obtengan mejores resultados que en el modelo SoCo-NG, donde cada n-grama se considera como una dimensión y su frecuencia como el valor de esta dimensión.

La primera prueba realizada con el modelo SoCo-LSA ha sido comprobar si la refactorización de funciones del código fuente afecta positiva o negativamente. Para ello, se realizó un pequeño experimento inicial donde se compara la eficacia del modelo con y sin incrustar los fragmentos de código fuente de las funciones en el lenguaje C considerando el corpus A&T++ (véase el apartado 3.1.3). Esta incrustación de código fuente consiste en sustituir la llamada a cada función por el cuerpo de estas. Para evitar casos de recursividad, solo se ha realizado la incrustación a las llamadas de dentro de la función principal, en este caso la función *main*. Debido al reducido número de funciones en esta colección, los resultados de incrustar no son significativos, por lo que solo se presentan los resultados sin realizar el preproceso de incrustación lo cual permite mantener el modelo independiente del lenguaje de programación.

Se tomarán los valores de similitud de cada código fuente contra el resto de códigos fuente y se ordenará por valor de similitud. Se espera que aquellos pares de códigos fuente con mayor valor de similitud se encuentren en lo alto del ranking. A diferencia del experimento anterior, donde se utilizó la curva de Precisión contra Cobertura, en este experimento se utilizará para comparar la eficacia de ambos modelos una métrica más cuantitativa. En concreto, esta métrica es la media de la precisión promedio (MAP, en inglés *Mean Average Precision*). La métrica MAP permite una evaluación del modelo teniendo en cuenta el orden de los resultados, penalizando aquellos pares que se consideran reutilizados pero se encuentran en una posición baja en el ranking de similitudes.

Para comparar este modelo con SoCo-NG debemos ajustar los diferentes valores de n-gramas, en concreto hemos considerado el rango $[1-3]$ tanto de caracteres como de palabras. Además de este ajuste en ambos modelos, en el modelo SoCo-LSA también debemos ajustar los diferentes

[5] Al convertir la representación de los n-gramas al espacio vectorial reducido se pierde cierta precisión en los números decimales.

valores de k, desde 10 hasta 400 dimensiones. En la tabla 5.9 se muestran los resultados al aplicar el modelo SoCo-LSA por cada valor de dimensiones k del nuevo espacio vectorial reducido. Los mejores resultados se han obtenido utilizando trigramas de caracteres y considerando 80 dimensiones. A partir de 80 dimensiones se obtiene el mismo valor de MAP, por lo que los códigos estarán mejor representados considerando las 80 dimensiones más significativas. La diferencia de eficacia entre utilizar caracteres o palabras para los n-gramas es debida a que el número de n-gramas extraídos utilizando palabras es muy reducido respecto al de caracteres. Así pues, los resultados con n-gramas de palabras son ligeramente inferiores respecto a utilizar n-gramas de caracteres.

	Caracteres			Palabras		
k	Unigramas	Bigramas	Trigramas	Unigramas	Bigramas	Trigramas
10	0,804	0,820	0,823	0,613	0,711	0,691
20	0,804	0,822	0,823	0,662	0,738	0,697
30	0,065	0,096	0,118	0,713	0,760	0,701
40	0,070	0,115	0,122	0,763	0,781	0,707
50	0,070	0,123	0,120	0,743	0,759	0,713
60	0,070	0,130	0,120	0,723	0,737	0,726
70	0,819	0,831	0,833	0,734	0,763	0,750
80	0,822	0,830	**0,836**	0,746	0,789	0,796
...	...	...	...	...	...	...
400	0,822	0,831	**0,836**	0,746	0,789	0,796

Tabla 5.9: Valores de *Mean Average Precision* para diferentes dimensiones K aplicando el modelo SoCo-LSA sin incrustación de código fuente para unigramas, bigramas y trigramas de caracteres y palabras sobre el corpus C de la colección A&T++.

En la tabla 5.10 se muestran los resultados del modelo SoCo-NG utilizando tanto caracteres como palabras para la extracción de n-gramas para valores de n del rango [1-3]. Considerando trigramas de caracteres se han obtenido resultados ligeramente mejores tal como ocurría en Flores et al. (2011). Además, se ha realizado un test estadístico t-Student para comparar los mejores resultados del modelo SoCo-LSA con el modelo SoCo-NG, donde las diferencias en los resultados resultaron ser significativas (p-value=0,045).

	Caracteres			Palabras		
	Unigramas	Bigramas	Trigramas	Unigramas	Bigramas	Trigramas
MAP	0,712	0,753	**0,774**	0,772	0,766	0,756

Tabla 5.10: Valores de *Mean Average Precision* para el modelo SoCo-NG considerando unigramas, bigramas y trigramas tanto de caracteres como de palabras.

En conclusión, ambos modelos han mostrado un mejor desempeño utilizando trigramas de caracteres. Aplicando el análisis semántico latente se ha conseguido una mejora de los resultados significativa desde un punto de vista estadístico. El modelo SoCo-LSA ha sido capaz de obtener información latente al reducir las dimensiones del espacio vectorial y considerando las dimensiones más representativas. Debido a los buenos resultados obtenidos con SoCo-LSA, y los nada despreciables del modelo SoCo-NG, el siguiente paso lógico es el de compararlos con otros modelos inspirados en técnicas del procesamiento del lenguaje natural y recuperación de información pero adaptados

para ser utilizados sobre códigos fuente. Este desarrollo de modelos y comparación de los mismos dará lugar al ensamble de los modelos construyendo un clasificador que aprovecha las bondades de cada uno de ellos.

Detección de reutilización mediante ensamble de modelos

En la mayoría de trabajos del estado de la cuestión, la detección de reutilización de código fuente se ha abordado generalmente desde la perspectiva de la compilación. Si consideramos el código fuente como un texto, podríamos aplicar sobre códigos fuente técnicas que normalmente son aplicadas sobre textos escritos en lenguaje natural. En esta experimentación se va a comparar el rendimiento de modelos utilizados para la recuperación de información textual y adaptados para poder aplicarlos a códigos fuente (Flores et al., 2016). Finalmente, se construirán clasificadores que ensamblen estos modelos. En concreto, los modelos propuestos son SoCo-LSA, SoCo-ESA, SoCo-COG, SoCo-NG y SoCo-WCR. No se ha considerado el modelo SoCo-Sliding porque ofrece resultados ligeramente inferiores al modelo SoCo-NG con un mayor coste computacional como se ha observado en el apartado anterior. Estos modelos se comparan con los ensambles para los lenguajes de programación C y Java. Los ensambles de modelos que se van a construir estarán compuestos por clasificadores comunes. Para ello, se tomará como entrada del clasificador el valor de similitud resultante de cada modelo.

El objetivo es conseguir un ensamble de modelos que obtenga mejores prestaciones en la detección de reutilización que los propios modelos actuando individualmente. La comparación de los modelos y de los ensambles se realiza utilizando el corpus A&T++ descrito en el apartado 3.1.3. Para comparar los modelos en términos de medidas como Precisión o Cobertura se ha establecido automáticamente un umbral para separar entre pares de códigos fuente reutilizados y no reutilizados. Este umbral de similitud se ha calculado utilizando el algoritmo C4.5 basado en el concepto de entropía de información[6]. Se ha elegido este algoritmo porque es capaz de establecer un umbral donde la ganancia de información sea máxima. En este caso solo existe una dimensión, el valor de similitud de cada modelo, por lo que solamente es necesario establecer un umbral para separar en dos clases (reutilizado y no reutilizado). La estimación del umbral de similitud con dicho algoritmo requiere de un corpus de entrenamiento balanceado para evitar el problema de que el clasificador siempre elija la clase mayoritaria (en nuestro caso los pares de códigos fuente no reutilizados). Para ello, se han considerado todas las muestras de la clase minoritaria con un subconjunto de muestras de la clase mayoritaria al azar del mismo tamaño que la partición minoritaria. Este método de balanceado se conoce por submuestreo. La entrada tanto del algoritmo C4.5 para determinar el resultado de clasificación de cada modelo como la entrada para el ensamble de modelos es un valor de similitud en el rango [0-1]. Se espera que el ensamble de modelos obtenga un mejor rendimiento que el obtenido por cada modelo por separado. Para realizar el ensamble de modelos se ha utilizado una variedad de clasificadores (utilizando los parámetros por defecto en Weka): *Random Forest*, *Hoeffding Tree*, *Adaboost M*1 y *Naive Bayes*. Estos clasificadores también se han entrenado utilizando un corpus balanceado. En concreto, se ha aplicado la técnica de submuestreo[7] del corpus A&T++. Esta técnica consiste en reducir las muestras de la clase mayoritaria, en nuestro caso la

[6]Dentro de la herramienta Weka (`http://www.cs.waikato.ac.nz/ml/weka/`) se corresponde con árboles J48.

[7]En el trabajo Yen et al. (2006) se hace una comparativa de los procedimientos más utilizados para balancear corpus desbalanceados.

clase de pares no reutilizados, eligiendo muestras al azar. Para la comparación de los resultados se utilizan las medidas Precisión, Cobertura y F_1[8].

	C			Java		
Modelos	Precisión	Cobertura	F_1	Precisión	Cobertura	F_1
SoCo-COG	0,695	0,500	0,582	**0,998**	0,914	0,953
SoCo-ESA	0,732	0,731	0,731	0,997	0,851	0,917
SoCo-WCR	0,712	0,663	0,687	0,997	0,720	0,835
SoCo-NG	0,835	0,436	0,573	0,997	0,691	0,817
SoCo-LSA	**0,992**	0,854	0,914	0,997	0,770	0,867
Ensambles de modelos						
Random Forest	**0,992**	**0,931**	**0,958**	**0,998**	**0,935**	**0,964**
Hoeffding Tree	0,851	0,765	0,806	0,997	0,883	0,937
Ada Boost	0,973	0,901	0,936	0,936	0,876	0,905
Naive Bayes	0,935	0,877	0,905	0,997	0,890	0,940

Tabla 5.11: Resultados de los modelos y de los ensambles de modelos en los corpus C y Java.

Los resultados obtenidos por los modelos y los ensambles sobre los corpus de C y Java se muestran en la tabla 5.11. Como se puede observar, el ensamble utilizando el clasificador *Random Forest* es el más eficaz en las tres medidas tanto para el lenguaje C como para Java.

En conclusión, en la mayoría de casos el ensamble de modelos ha mostrado un incremento del rendimiento respecto a los modelos trabajando individualmente en ambos lenguajes de programación, incluso sin ajuste de parámetros. El mejor rendimiento se obtiene con el ensamble que utiliza en el clasificador *Random Forest* en ambos lenguajes de programación, C y Java. Las aproximaciones SoCo-LSA y SoCo-COG también obtuvieron buenos resultados, incluso, comportándose con mejor rendimiento que algún ensamble. La utilización de ensamble mediante un clasificador muestra resultados prometedores para la detección monolingüe de reutilización en código fuente, mejorando las propuestas actuales.

5.2 Evaluación en escenarios translingües

En este apartado se va a resumir la experimentación realizada a nivel translingüe sobre detección de reutilización en códigos fuente. Todos los experimentos resumidos en este apartado están realizados sobre corpus de distinta naturaleza.

En el primer escenario (apartado 5.2.1) se describe la experimentación preliminar a nivel translingüe. En este escenario, se detecta reutilización translingüe dentro de un proyecto software (corpus SPADE) en diferentes lenguajes de programación. Se evalúa el modelo básico desde distintas aproximaciones permitiendo el mejor ajuste del mismo.

[8]La medida F_1 permite evaluar los resultados de un sistema. Considera tanto la Precisión como la Cobertura para calcular el valor, en concreto es la media harmónica de estas medidas $F_1 = 2 \cdot (Precisión \cdot Cobertura)/(Precisión + Cobertura)$.

En el segundo escenario se considera un corpus con casos reales de reutilización a nivel translingüe (corpus A&T++ translingüe). En este caso se evalúan las dos primeras propuestas de modelos utilizados (SoCo-NG y SoCo-Sliding) tal como se realizó a nivel monolingüe en el apartado 5.1.3. Este escenario añade mayor dificultad ya que todos los códigos fuente tienen un cierto grado de similitud por tener que resolver la misma tarea.

En el tercer escenario se desarrolla la evaluación de los modelos propuestos en esta tesis sobre un corpus multilingüe de códigos fuente de tamaño medio (corpus Rosettacode.org). En un primer experimento se observan los modelos que son capaces de recuperar pares de códigos fuente paralelos y códigos fuente comparables. Obviamente, al tener una mayor similitud por considerarse traducciones, los pares de códigos fuente paralelos tienen una dificultad menor. A continuación se simula un escenario de detección translingüe considerando los pares de códigos fuente paralelos como reutilizados y los comparables como no reutilizados. Finalmente, se establece un umbral basado en las distribuciones sobre los dos modelos que han mostrado una mayor eficacia, para llevar a cabo la comparación de los mismos.

5.2.1 Experimentación preliminar

En este experimento el objetivo consiste en determinar qué tipo de preproceso y estrategias de ponderación de las características de un código fuente permiten distinguir mejor los códigos fuente reutilizados entre distintos lenguajes de programación en un escenario real utilizando el modelo básico SoCo-NG (véase el apartado 4.2). El experimento trata de encontrar qué código fuente ha sido reutilizado desde un lenguaje de programación L a un lenguaje L'. Esta búsqueda se realiza sobre un conjunto de códigos fuente escritos en el lenguaje L' de los cuales se calcula un valor de similitud y posteriormente se ordena. El escenario ideal situaría el código fuente reutilizado en primer lugar del ranking ordenado por similitud.

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 el estilo del programador a través de distintos lenguajes de programación.

Para realizar este experimento preliminar se ha utilizado el corpus SPADE (véase el apartado 3.1.1). Este corpus está compuesto por una colección de códigos fuente escritos en tres lenguajes de programación: C++, Java y Python. Debido a las características del corpus, donde todos los códigos fuente son muestras positivas de reutilización de un lenguaje L a otro lenguaje L', el conjunto de códigos fuente consulta está formado por todos los códigos fuente en el lenguaje L, mientras que el corpus de referencia está compuesto por todos los códigos fuente escritos en el lenguaje L'. Como el corpus está compuesto por códigos fuente escritos en tres lenguajes de programación, se han extraído resultados para cada par de lenguajes. Con el objetivo de determinar qué partes de código fuente pueden indicar mejor la existencia de reutilización, se han realizado experimentos teniendo en cuenta distintos preprocesos, en concreto, se ha evaluado el modelo SoCo-NG bajo los siguientes supuestos: *(i)* considerando el código fuente entero sin modificar (*full code*); *(ii)* tomando el código fuente sin los comentarios (*fc-without comments*); *(iii)* considerando solo las palabras reservadas del lenguaje (*fc-reserved words only*); *(iv)* solo comentarios contenidos en el código fuente (*comments only*); *(v)* el código fuente sin las palabras reservadas (*fc-without rw*); *(vi)* y el código fuente sin

comentarios ni palabras reservadas (*fc-wc-wrw*). Además, también se han realizado experimentos variando el tamaño y el pesado de la frecuencia de los n-gramas.

La experimentación consiste en, dado un código fuente c escrito en un lenguaje L, encontrar la posición del código fuente c' en el ranking ordenado que se considera reutilizado en una colección de códigos fuente escritos también en L'. Por lo tanto, se considerará que detecta mejor la reutilización de código fuente la configuración de parámetros y preproceso que sitúe a c' en la primera posición del ranking.

En las siguientes tablas se muestran los resultados de los experimentos. De las seis tablas, las tres primeras (tablas 5.12, 5.13 y 5.14) corresponden al análisis utilizando tf para el pesado de términos, y las tres siguientes (tablas 5.15, 5.16 y 5.17) corresponden al análisis aplicando la técnica de pesado $tf - idf$. Cada tabla resume los resultados obtenidos entre cada par de lenguajes: Java–C++, Python→C++ y Python→Java . Los resultados se muestran en términos de la media y desviación típica de las posiciones de los códigos fuente que se consideran reutilizados respecto del código que está siendo analizado para distintos valores de n-gramas y los tipos de modificaciones comentados en el párrafo anterior.

En la tabla 5.12 se reflejan los resultados obtenidos con el modelo SoCo-NG considerando distintas partes del código fuente para el par de lenguajes Java–C++. Cada experimento se ha repetido variando el valor de n en el rango $[1 \dots 5]$. En la mayoría de experimentos se refleja como a partir de trigramas se encuentran los mejores resultados. También se aprecia que considerando solamente los comentarios o solo las palabras reservadas del lenguaje, la posición media empeora al subir el valor de n a 4 y 5. Esta tendencia se repite en todos los pares de lenguajes y usando ambos tipos de pesado de frecuencias (tf y $tf - idf$). Por otra parte, los mejores resultados se han obtenido considerando el código fuente completo y también 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.

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 ± 1,57	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 5.12: Resultados obtenidos comparando los códigos fuente del corpus SPADE escritos en C++ y en Java. Cada código es pesado utilizando tf. La tabla refleja la media y desviación típica de las posiciones de los códigos considerados reutilizados respecto del código que está siendo analizado.

Los resultados del estudio de reutilización en el corpus SPADE a nivel de documento entre los lenguajes Python→C++ se muestra en la tabla 5.13. En comparación con el par Java–C++, los resultados son ligeramente mejores al tener en cuenta valores de n de 3 o superior. Esto es debido a que entre el par Java–C++ 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++, es necesario adaptar y modificar más el código fuente. Este proceso de adaptación hace que exista menos similitud y los resultados no sean tan buenos como en el otro caso.

Así como ocurre en el par Java–C++, la mejor posición en el ranking se obtiene si consideramos el código fuente entero y también el código sin los comentarios, aunque esta vez solo considerando trigramas de caracteres.

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

Tabla 5.13: Resultados obtenidos comparando los códigos fuente del corpus SPADE escritos en Python y en C++. Cada código es pesado utilizando tf. La tabla refleja la media y desviación típica de las posiciones de los códigos considerados reutilizados respecto del código que está siendo analizado.

Entre el par de lenguajes Python→Java, el mejor resultado obtenido ha sido considerando trigramas de caracteres, el código fuente sin comentarios y sin palabras reservadas como se muestra en la tabla 5.14. Esto se debe a que el proceso de reutilización entre ambos lenguajes ha requerido utilizar librerías diferentes de las estándares. Además, al emplear estas librerías, parte de la funcionalidad que se realizaba en el código fuente ahora se realiza internamente en la librería. La similitud estimada proviene 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.

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

Tabla 5.14: Resultados obtenidos comparando los códigos fuente del corpus SPADE escritos en Python y en Java. Cada código es pesado utilizando tf. La tabla refleja la media y desviación típica de las posiciones de los documentos considerados reutilizados respecto del código que está siendo analizado.

Como se ha comentado al inicio del experimento, también hemos aplicado el método de pesado $tf-idf$ sobre los tres pares de lenguajes de programación. En la tabla 5.15 se muestran los resultados obtenidos al utilizar el método de pesado $tf-idf$ sobre el par de lenguajes Java–C++ y considerando los mismos experimentos que para tf. Como en el caso de utilizar tf entre este par de lenguajes, los mejores resultados se obtuvieron considerando el código fuente entero y también considerando el código sin los comentarios. El hecho que ambos lenguajes de programación comparten parte de su sintaxis, facilita al programador la reutilización del código, siendo esta más similar y por lo tanto más fácil de detectar por aproximaciones basadas en características léxicas como SoCo-NG.

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 5.15: Resultados obtenidos comparando los códigos fuente del corpus SPADE escritos en C++ y en Java. Cada código es pesado utilizando $tf - idf$. La tabla refleja la media y desviación típica de las posiciones de los documentos considerados reutilizados respecto del código 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 5.16. Se ha revisado manualmente porqué los valores de similitud obtenidos son tan cercanos cuando utilizamos 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 estos no influyan en el resultado de la detección. A diferencia de utilizar tf entre este par de lenguajes, los mejores resultados utilizando idf se obtienen además de con trigramas, con valores de n de 4 y 5.

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 5.16: Resultados obtenidos comparando los códigos fuente del corpus SPADE escritos en Python y en C++. Cada código es pesado utilizando $tf - idf$. La tabla refleja la media y desviación típica de las posiciones de los códigos considerados reutilizados respecto del código que está siendo analizado.

El estudio realizado entre el par de lenguajes Python→Java utilizando como método de pesado $tf - idf$ se muestra en la tabla 5.17. El 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, la diferencia en posición media es de 0,12. 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 estudio para determinar la relevancia de cada n-grama.

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 a nivel translingüe (Potthast et al., 2011). Por otra parte, como se puede apreciar en la tabla 5.18, los mejores resultados se obtienen utilizando el código entero

Experimentos/n-gramas	1	2	3	4	5
full code	$2{,}50 \pm 1{,}50$	$1{,}62 \pm 0{,}98$	$1{,}62 \pm 1{,}23$	$1{,}62 \pm 1{,}23$	$\mathbf{1{,}50 \pm 0{,}75}$
fc-without comments	$2{,}62 \pm 1{,}23$	$1{,}62 \pm 0{,}98$	$1{,}62 \pm 1{,}23$	$1{,}62 \pm 1{,}23$	$\mathbf{1{,}50 \pm 0{,}75}$
only comments	$2{,}00 \pm 2{,}00$	$2{,}33 \pm 1{,}56$	$3{,}00 \pm 0{,}67$	$2{,}33 \pm 1{,}56$	$4{,}00 \pm 0{,}00$
fc-only reserved words	$2{,}50 \pm 1{,}50$	$2{,}62 \pm 0{,}98$	$2{,}62 \pm 0{,}98$	$2{,}88 \pm 1{,}11$	$2{,}88 \pm 1{,}11$
fc-without rw	$2{,}50 \pm 1{,}50$	$2{,}00 \pm 1{,}50$	$2{,}50 \pm 1{,}25$	$1{,}75 \pm 0{,}19$	$3{,}50 \pm 0{,}75$
fc-wc-wrw	$2{,}50 \pm 1{,}50$	$2{,}00 \pm 1{,}50$	$2{,}50 \pm 1{,}25$	$1{,}75 \pm 0{,}19$	$4{,}00 \pm 0{,}83$

Tabla 5.17: Resultados obtenidos comparando los códigos del corpus SPADE escritos en Python y en Java. Cada código es pesado utilizando $tf - idf$. La tabla refleja la media y desviación típica de las posiciones de los códigos fuente considerados reutilizados respecto del código que está siendo analizado.

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/pares de lenguajes	Java–C++	Python→C++	Python→ Java
full code	$1{,}00 \pm 0{,}00$	$1{,}44 \pm 0{,}83$	$1{,}62 \pm 1{,}10$
fc-without comments	$1{,}00 \pm 0{,}00$	$1{,}44 \pm 0{,}83$	$1{,}62 \pm 1{,}10$
only comments	$2{,}29 \pm 1{,}57$	$2{,}83 \pm 1{,}38$	$3{,}00 \pm 0{,}67$
fc-only reserved words	$1{,}56 \pm 0{,}83$	$1{,}78 \pm 1{,}02$	$1{,}75 \pm 0{,}83$
fc-without rw	$1{,}44 \pm 0{,}83$	$1{,}78 \pm 1{,}13$	$2{,}00 \pm 1{,}32$
fc-wc-wrw	$1{,}44 \pm 0{,}83$	$1{,}67 \pm 0{,}94$	$1{,}44 \pm 0{,}69$

Tabla 5.18: Resultados obtenidos con trigramas de caracteres y tf como medida de pesado. Cada valor representa la media y la desviación típica de las posiciones de los códigos considerados como reutilizados respecto al código fuente que se está analizando.

En conclusión, tras los resultados se ha determinado 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 de tamaño pequeño y mediano 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.

5.2.2 Detección de reutilización en ámbito académico

En este apartado se resume una serie de experimentos realizados a nivel translingüe sobre el corpus A&T++ descrito en el apartado 3.1.3. En concreto, el corpus representa un escenario del ámbito académico, donde una tarea propuesta por el profesor debe ser resuelta por el conjunto de alumnos presentando casos reales de reutilización que se han traducido automáticamente. En estos experimentos se comparan los modelos SoCo-NG y SoCo-Sliding tal como se realizó a nivel monolingüe (véase el apartado 5.1.3). Tal como ocurrió en la experimentación a nivel monolingüe, se ha realizado una comparación inicial utilizando el corpus A&T++ (véase el apartado 3.1.3) que consta de una colección de 79 códigos fuente del ámbito académico escritos en C que han sido traducidos al lenguaje Java. El propósito de este experimento a nivel translingüe consiste en comparar el desempeño de los modelos en un escenario simulado de reutilización por traducción de códigos fuente. En este trabajo inicial, se comparan el modelo básico de n-gramas, SoCo-NG, con el modelo SoCo-Sliding.

Ambos modelos se evalúan bajo los mismos parámetros descritos en los apartados 4.2 y 4.3. Se divide en n-gramas de tres caracteres el código fuente sin eliminar comentarios, identificadores o palabras reservadas, estima su pesado en base a la frecuencia de términos tf y posteriormente calcula la similitud mediante la métrica del coseno. Por otro lado, en el modelo SoCo-Sliding la evaluación se realiza entre fragmentos del código fuente, en concreto fragmentos del tamaño de 300 caracteres. Para estimar si un par de fragmentos presenta reutilización o no, utiliza un umbral de 0,9 para el valor de similitud. La similitud entre dos códigos fuente se establece por la proporción de fragmentos que se han considerado reutilizados.

En la figura 5.3 se muestran las curvas de Precisión contra Cobertura de los modelos SoCo-NG y SoCo-Sliding. Ambos modelos mantienen el valor de precisión con valor 1 hasta valores de cobertura 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. Sin embargo, las diferencias no son significativas comparadas con el alto coste computacional que conlleva el calcular similitudes entre cada par de ventanas de dos códigos fuente. Además, no se disponen de recursos con casos de reutilización a nivel de fragmentos y los recursos que se han creado en el marco de esta tesis de investigación suponen casos de reutilización a nivel de documento. Por ello, en el resto de experimentos solo se considerará el modelo de comparación basado en n-gramas a nivel de documento.

5.2.3 Detección de reutilización en corpus comparable y paralelo

Para la detección de reutilización translingüe el propósito de este experimento consiste en aplicar modelos conocidos en el ámbito de recuperación de información y que ya han demostrado un buen comportamiento en detección de reutilización translingüe en textos (Potthast et al., 2011). En este experimento se utilizaran tanto modelos que no utilizan corpus de entrenamiento como modelos que sí los requieren. En esta evaluación se han comparado tanto modelos descritos en el capítulo 4 así como también combinaciones de estos. Modelos como SoCo-ASA (véase el apartado 4.7) están entrenados para detectar las traducciones más probables y modelos como SoCo-ESA (véase el

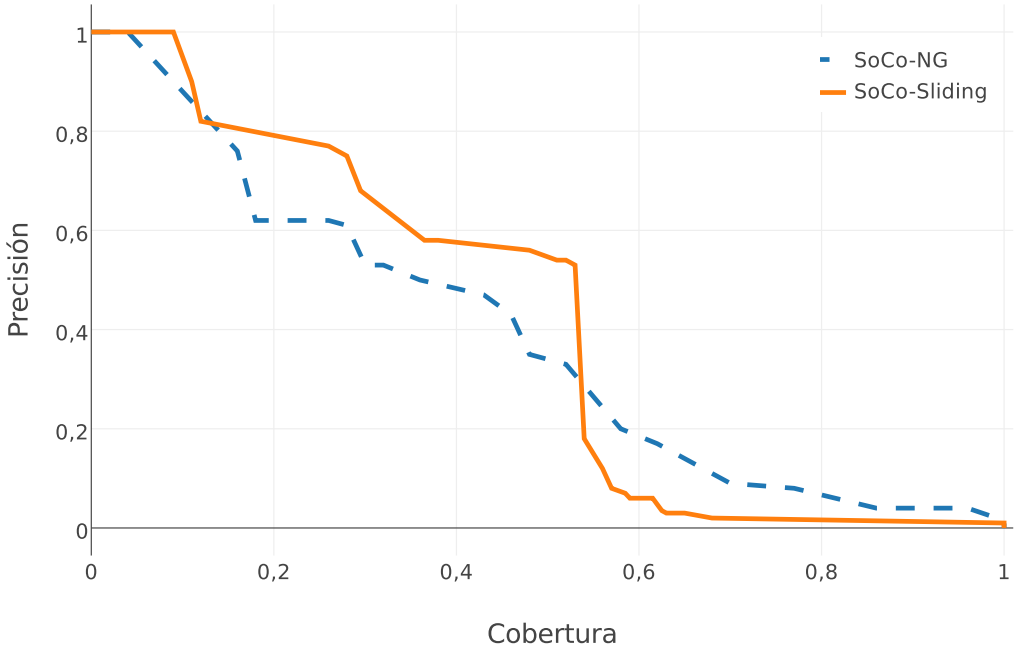


Figura 5.3: Resultados de los modelos SoCo-NG y SoCo-Sliding a nivel translingüe.

apartado 4.6) para detectar documentos de tópicos similares. Se han comparado todos los modelos en escenarios paralelos y escenarios comparables entre los tres lenguajes de programación, C, Java y Python, y en ambos sentidos de traducción (por ejemplo: de C a Java y de Java a C). Para poder comparar los modelos a nivel paralelo y comparable se ha utilizado el corpus Rosettacode.org. Este corpus contiene casos de códigos fuente comparables y paralelos entre los lenguajes de programación C, Java y Python. En total se han calculado más de cinco millones de similitudes por cada modelo.

Se ha simulado el escenario de reutilización translingüe de códigos fuente. Dado un código fuente origen, el objetivo es encontrar el código fuente correspondiente en otro lenguaje de programación dentro de una colección. En el escenario paralelo el objetivo consiste en encontrar el código fuente que ha sido traducido del original, mientras que en el escenario comparable, consiste en encontrar el código fuente solución al mismo problema que el código fuente origen.

La metodología seguida es la siguiente: c es un código fuente consulta de la colección K, K consiste en códigos fuente alineados con la colección K' ya sea de manera comparable o paralela, y c' que pertenece a K', es un código fuente alineado con c. Dado el código fuente c, todos los códigos fuente de la colección K' incluido c' se ordenan de mayor a menor valor de similitud translingüe con c. El modelo ideal debe situar a c' en la primera posición del ranking. Este proceso se ha repetido cinco veces utilizando validación cruzada y, posteriormente, promediando los resultados.

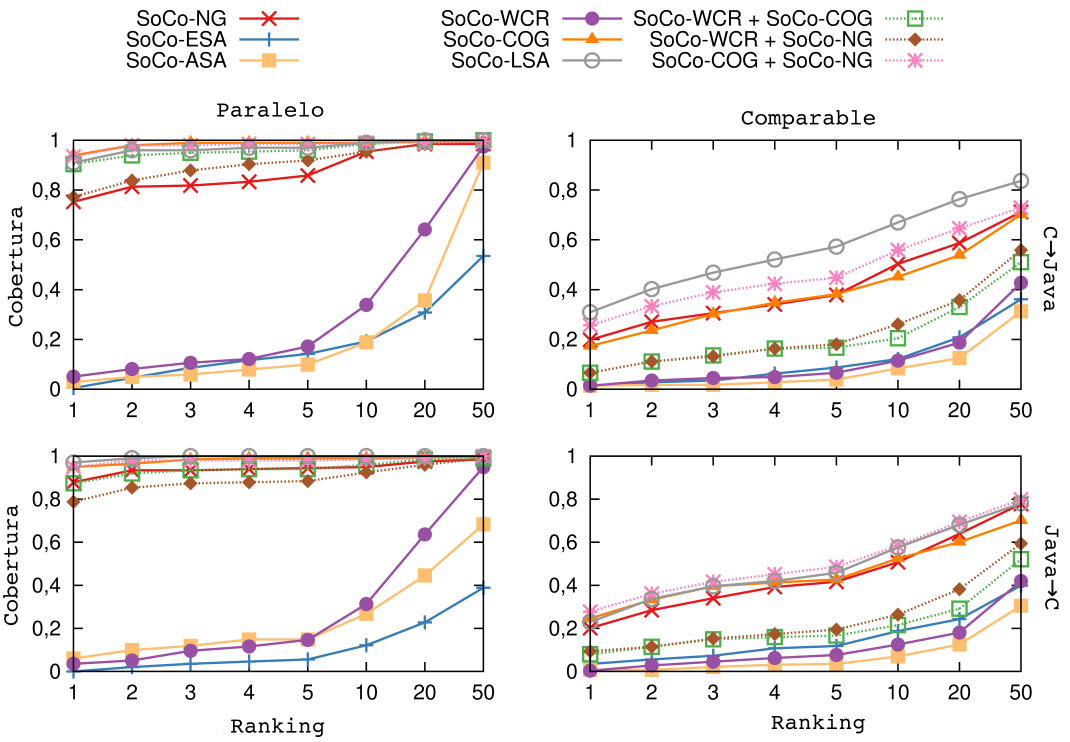


Figura 5.4: Comparación de los modelos en escenario paralelo y comparable en términos de Cobertura en el par de lenguajes C-Java.

Los resultados de la experimentación se muestran en las siguientes figuras: figura 5.4 para el par de lenguajes C-Java, figura 5.5 para Java-Python y figura 5.6 para C-Python en forma de curvas de cobertura contra el número de posiciones consideradas en el ranking. Se observa una diferencia notoria entre el escenario paralelo y comparable en términos de cobertura. Esta diferencia se debe a que los códigos fuente comparables que están alineados solo tienen en común que resuelven el mismo problema, pudiendo no compartir ningún fragmento de código fuente y, por lo tanto, no contener elementos para ser considerados similares.

El modelo SoCo-LSA y la combinación de los modelos SoCo-COG y SoCo-NG que no requieren un entrenamiento previo muestran los mejores resultados en la mayoría de pares de lenguajes y escenarios, también mejorando a la aplicación de los modelos por separado de la combinación. La combinación de estos dos modelos con el modelo SoCo-WCR reduce el rendimiento ligeramente.

Tanto el modelo SoCo-LSA como la combinación de SoCo-COG y SoCo-NG muestran un rendimiento casi perfecto entre el par de lenguajes Java–Python en el escenario paralelo. Este comportamiento puede ser debido a que el traductor utilizado (*java2python*) genera traducciones de código fuente prácticamente idénticas en cuanto a sintaxis. Estos modelos muestran un gran rendimiento cuando la sintaxis, estructura y vocabulario entre lenguajes es muy similar. Otra evidencia de esta situación

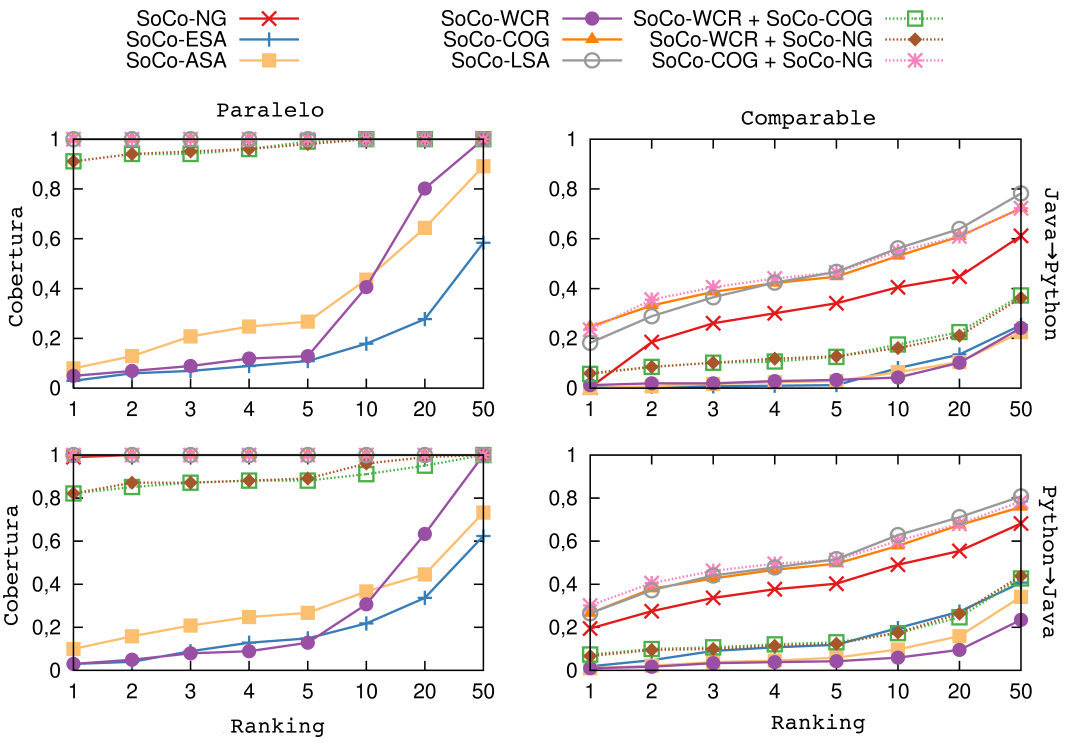


Figura 5.5: Comparación de los modelos en escenario paralelo y comparable en términos de Cobertura en el par de lenguajes Java-Python.

se muestra en el factor de longitud entre las traducciones generadas entre este par de lenguajes en la tabla 4.3 del apartado 4.7, donde el parámetro μ es cercano a 1 y la desviación típica es pequeña. Esto significa que las traducciones tienen un número de tokens muy similar al código que ha sido traducido.

En general, los modelos que requieren un entrenamiento previo, a excepción de SoCo-LSA, han mostrado un rendimiento más bajo que el resto de modelos. Este bajo rendimiento podría deberse a que la cantidad de códigos fuente en el proceso de entrenamiento es insuficiente. Para apoyar esta hipótesis se ha entrenado el modelo SoCo-ESA con distintas cantidades de códigos fuente comparables para comprobar si su rendimiento varía con el tamaño del corpus de entrenamiento. En la figura 5.7 se observa una ligera tendencia de crecimiento de la cobertura para todos los pares de lenguajes y escenarios (comparable y paralelo). Cuanto más grande es el corpus comparable de entrenamiento utilizado por SoCo-ESA, mejor es el rendimiento de este. Además, se ha estudiado si estos cambios son estadísticamente significativos comparando los resultados obtenidos considerando el 20 % del corpus de entrenamiento frente a utilizar el 100 %. En la tabla 5.19 se muestra que existen diferencias significativas entre la mayoría de escenarios comparables.

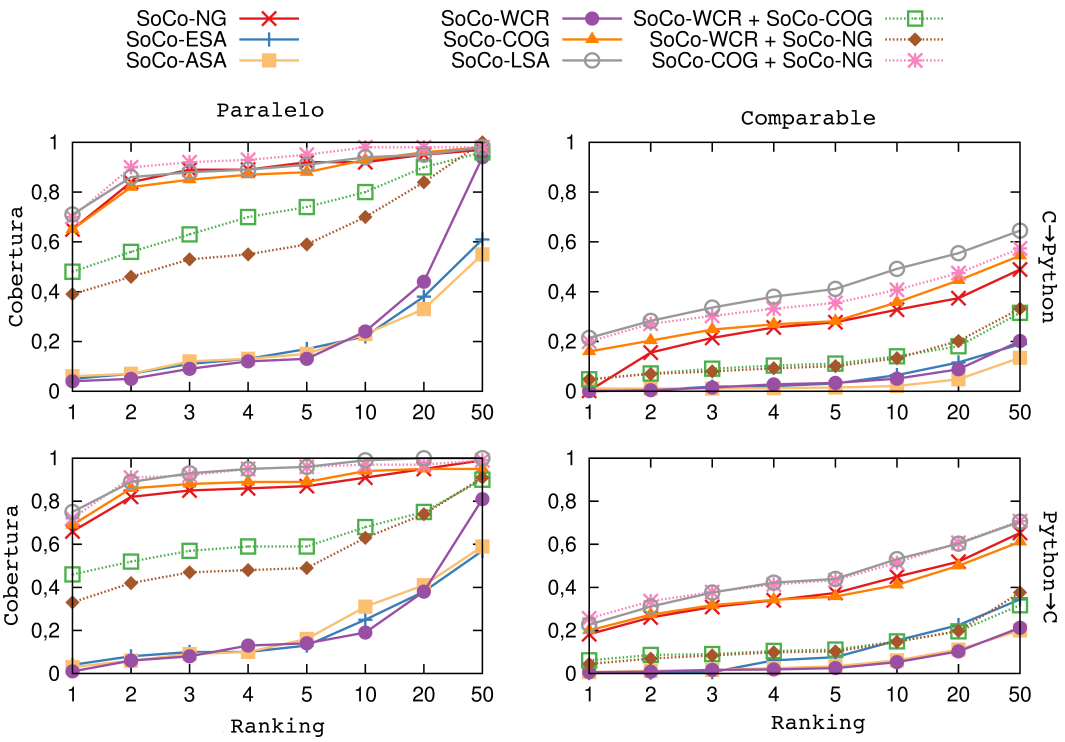


Figura 5.6: Comparación de los modelos en escenario paralelo y comparable en términos de Cobertura en el par de lenguajes C-Python.

También se realizaron ajustes sobre el modelo SoCo-ASA para tratar de mejorar la calidad de los datos de entrenamiento y así mejorar el rendimiento del modelo. Estos ajustes consisten en modificar la extracción de probabilidades entre términos durante el entrenamiento para obtener probabilidades más precisas. Debido a que el entrenamiento se ha realizado a nivel de documento y no a nivel de oración como se realizó en textos (Potthast et al., 2011), el diccionario estadístico aprendido del corpus paralelo de Rosettacode.org contiene para cada término 32 posibles términos como posibles traducciones con sus respectivos 32 valores de probabilidad. Este es el número por defecto que la herramienta de entrenamiento GIZA++ establece como número máximo de entradas del diccionario. Todas estas probabilidades provienen de términos que suelen aparecer en la mayoría de códigos fuente (por ejemplo la función *main*) y que no son realmente traducciones. Su probabilidad es baja pero introduce cierto ruido que se ha eliminado al considerar solo aquellas traducciones que se encuentren entre un cierto porcentaje más probable, es decir, se ordenan las traducciones aprendidas de mayor a menor probabilidad y solo se consideran aquellas probabilidades más altas que sumadas entren dentro del porcentaje. Se ha variado el porcentaje con incrementos del 20 % para encontrar el porcentaje que permite mejorar el rendimiento del modelo SoCo-ASA en ambos escenarios.

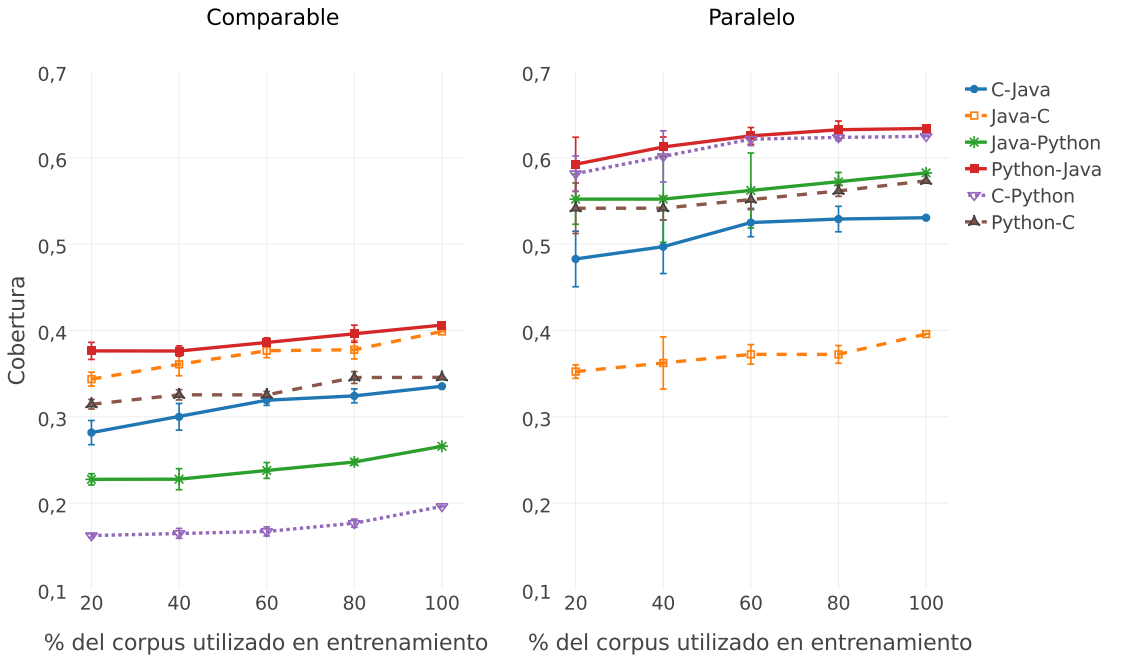


Figura 5.7: Comparación de diferentes tamaños del corpus de entrenamiento en el modelo SoCo-ESA en términos de cobertura en el top-50 para todos los pares de lenguajes de programación.

Otra modificación consiste en aprender las probabilidades utilizando una tabla de equivalencias entre las instrucciones básicas de los lenguajes de programación.[9] Considerando tanto el modelo SoCo-ASA original, como el modelo seleccionando las entradas de mayor probabilidad y el modelo entrenado a partir de tabla de equivalencias, se ha observado que utilizando el 40 % más probable en las entradas aprendidas a partir del corpus paralelo obtenemos un rendimiento ligeramente superior que el resto de variantes de SoCo-ASA estudiadas.

Observando cómo está creado el corpus se encuentran similitudes con un escenario de plagio translingüe. En un escenario de plagio translingüe, por ejemplo en el ámbito académico, donde todos los alumnos deben resolver un mismo problema, al realizar la comparación de todos sus códigos fuente contra todos, existen pares de códigos fuente que resuelven el mismo problema pero con estrategias distintas y por lo tanto son considerados como casos no reutilizados (así como lo son los pares de códigos fuente comparables). Por otra parte, los casos de pares de códigos reutilizados, siguen la misma estrategia y pueden contener modificaciones para evitar ser detectados (como los pares de códigos fuente paralelos donde el propio traductor de código puede refactorizar parte del código fuente si es necesario).

Dadas las similitudes encontradas entre los pares de códigos fuente paralelos y comparables con los de un escenario real, se ha experimentado cómo se comportan los modelos en un escenario

[9] En concreto la tabla utilizada ha sido extraída del artículo de la Wikipedia en `http://en.wikipedia.org/wiki/Comparison_of_programming_languages_(basic_instructions)` (Accedido en Marzo 2014).

	C→Java	Java→C	Java→Python	Python→Java	C→Python	Python→C
Paralelo	0,1141	**0,0003**	0,3863	0,2775	0,4201	0,2337
Comparable	**0,0015**	0,3581	**0,0442**	**0,0000**	**0,0000**	**0,0000**

Tabla 5.19: Resultados de significancia estadística con la prueba *t-Student* para el modelo SoCo-ESA entre los resultados considerando el 20 % y el 100 % del corpus de entrenamiento. Un valor p (*p-value* $< 0{,}05$ significa que las diferencias son significativas).

muy similar al de reutilización de códigos fuente translingüe. Para ello, se han tomado los pares comparables y paralelos como casos no reutilizados y reutilizados respectivamente. Cabe destacar que en la experimentación anterior se comparaba un código fuente (de la colección comparable o paralela) contra toda la colección (comparable o paralela), mientras que en este experimento solo se utilizan los pares ya definidos como comparables y paralelos, es decir, ya etiquetados como que resuelven el mismo problema. En la figura 5.8 se muestran las curvas de Precisión/Cobertura de todos los modelos y en los tres pares de lenguajes de programación estudiados: C–Java, Java–Python y C–Python. Además, en la tabla 5.20 se muestran los valores del área bajo la curva de cada modelo.

El modelo SoCo-LSA muestra un rendimiento similar a SoCo-NG en el escenario C–Java. Incluso sus valores de área bajo la curva son prácticamente iguales. En los otros dos pares de lenguajes, Java–Python y C–Python, la curva del modelo SoCo-LSA se muestra por encima del resto de modelos durante en la mayoría del gráfico. Así como en el caso monolingüe, SoCo-LSA muestra un desempeño mejor que SoCo-NG (Flores et al., 2014c). Como se observó en el experimento del análisis de los modelos en función de la posición en el ranking, el resto de modelos se comporta de manera similar a como lo hacían al recuperar códigos fuente comparables y paralelos por separado (Flores et al., 2014b), donde SoCo-NG y SoCo-COG obtenían buenos resultados comparados con los relativamente bajos de SoCo-WCR, SoCo-ESA Y SoCo-ASA.

Respecto a los distintos pares de lenguajes de programación, en general son mejores los resultados en los escenarios C–Java y Java–Python. SoCo-LSA incluso muestra un rendimiento casi perfecto en el par Java–Python. La razón de este comportamiento se debe a que el traductor de código fuente *java2python* produce traducciones casi idénticas respecto a la sintaxis del código original. Otra evidencia de este fenómeno son los parámetros de la proporción de longitud que se muestran en la tabla 4.3: μ tiene un valor cercano a 1, y la desviación típica es pequeña. Esto significa que las traducciones tienen un número casi igual de tokens que el código fuente original. Por otra parte, el traductor de código fuente *C++ to Java Converter*, necesita transformar el código fuente en código fuente refactorizado. Incluso si el traductor no reconoce una cierta llamada a función, este traductor genera una nueva función vacía para ser rellenada por el programador. En el escenario C–Python, los códigos fuente paralelos son el resultado de dos traducciones, de C a Java y de Java a Python, mientras que en los otros dos escenarios solo es necesaria una traducción.

Par de lenguajes	SoCo-LSA	SoCo-NG	SoCo-ESA	SoCo-ASA	SoCo-COG	SoCo-WCR
C–Java	**0,768**	**0,770**	0,603	0,470	0,708	0,690
Java–Python	**0,969**	0,797	0,411	0,399	0,523	0,459
C–Python	**0,365**	0,296	0,238	0,213	0,294	0,322

Tabla 5.20: Área bajo la curva de Precisión-Cobertura para cada modelo y lenguaje de programación.

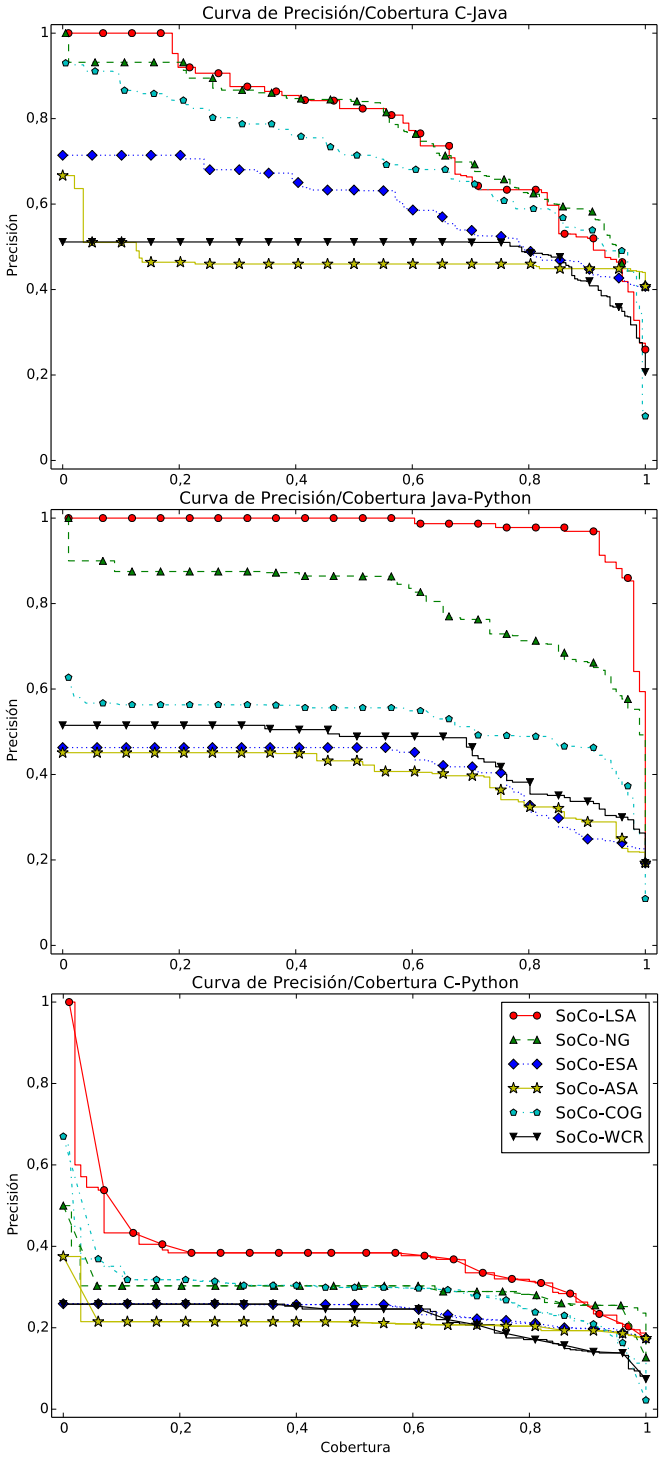


Figura 5.8: Curvas de Precisión/Cobertura para cada modelo y lenguaje de programación.

Cálculo del umbral de similitud

Finalmente, observando las grandes diferencias entre ambos escenarios, paralelo y comparable, con los modelos de mayor rendimiento, en este caso SoCo-LSA y SoCo-NG, se intuye que se pueda establecer un umbral de similitud que distinga entre códigos paralelos y comparables. Se considera el umbral de similitud el menor valor de similitud por el que un par de códigos fuente es considerado un caso de reutilización. Se espera que los pares de códigos fuente paralelos tengan una similitud mayor por el acierto mostrado en la experimentación anterior. La estimación del umbral entre los pares de códigos fuente paralelos y comparables se ha realizado considerando el 66 % de la partición de test. El 33 % restante se utiliza para medir las prestaciones de los umbrales. Para tratar de establecer un umbral que separe los códigos fuente comparables de los paralelos, se han generado histogramas por valor de similitud tanto para los pares de códigos fuente comparables como para los paralelos en intervalos de valor 0,05 como se muestra en la figura 5.9 utilizando los modelos que han mostrado mayor rendimiento, y para uno de los modelos de peor desempeño, en este caso SoCo-ASA entre el par de lenguajes C++ y Java. Para analizar las distribuciones, también se ha generado una línea de tendencia de la distribución de tipo polinómico de grado 6.

En los histogramas se aprecia que aquellos modelos que tienen mejor rendimiento en ambos escenarios distinguen mejor entre pares de códigos fuente paralelos y comparables. De hecho, la distribución generada para los pares de códigos fuente paralelos y la generada para los comparables con el modelo SoCo-ASA son muy similares resultando complicado distinguir qué códigos fuente son paralelos y cuáles son comparables. En los modelos SoCo-LSA y SoCo-NG se observa que a partir del punto de corte de las líneas que describen ambas distribuciones de similitud la cantidad de códigos fuente de una clase (comparable/paralelo) aumenta mientras que la cantidad de la otra clase disminuye. Por lo tanto, el punto de corte de las líneas de distribución puede ser un buen umbral para distinguir entre comparable y paralelo.

En la tabla 5.21 se detallan los valores de Precisión, Cobertura y la medida F_1 que combina la Precisión y Cobertura en una única medida para los umbrales establecidos en los modelos SoCo-LSA y SoCo-NG. En general, ambos modelos obtienen buenos resultados tanto en Precisión, Cobertura y F_1. Sin embargo, el modelo SoCo-LSA muestra un mejor comportamiento en términos del valor de F_1 en los tres pares de lenguajes. Ambos modelos muestran la misma tendencia que en las curvas de Precisión/Cobertura, disminuyendo su rendimiento en el par C–Python.

Modelo	Par de lenguajes	Umbral	Precisión	Cobertura	F_1
SoCo-LSA	C–Java	0,750	0,674	0,853	**0,753**
	Java–Python	0,770	0,756	1,000	**0,861**
	C–Python	0,550	0,304	0,824	**0,444**
SoCo-NG	C–Java	0,370	0,630	0,773	0,694
	Java–Python	0,410	0,526	0,882	0,659
	C–Python	0,270	0,280	0,618	0,385

Tabla 5.21: Evaluación de los modelos SoCo-LSA y SoCo-NG para distinguir entre pares de códigos fuente comparables y paralelos por cada par de lenguajes de programación.

Este experimento permite establecer un umbral para distinguir pares de códigos fuente paralelos entre lenguajes de programación. Es importante destacar que cuando se mezclan los códigos fuen-

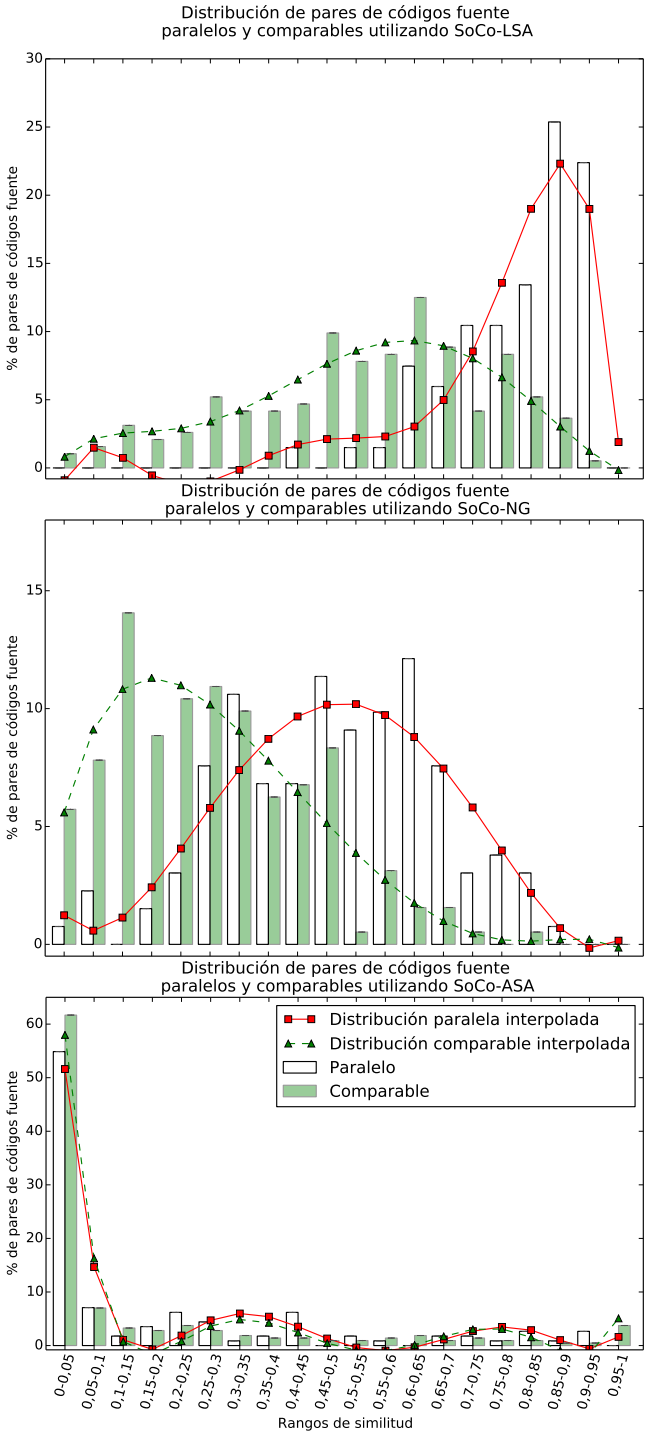


Figura 5.9: Histogramas por rango de similitud y líneas de tendencia utilizando los modelos SoCo-LSA, SoCo-NG y SoCo-ASA para los códigos fuente comparables y paralelos en el escenario C→Java.

te paralelos y comparables, es muy similar a un caso real de reutilización. Un claro ejemplo de este escenario se da en el ámbito académico. En este caso se puede producir reutilización entre un conjunto de códigos fuente escritos en diferentes lenguajes de programación que tratan de resolver el mismo problema. Los pares de códigos fuente que no han sido reutilizados se consideran como pares de códigos fuente comparables, mientras que los pares de códigos fuente reutilizados son considerados como paralelos. Tanto en la experimentación para comprobar la eficacia de los modelos separando pares de códigos fuente comparables y paralelos individualmente, así como en la simulación del escenario de reutilización considerando los comparables, los modelos SoCo-LSA y SoCo-NG tienen un alto rendimiento tanto en la recuperación translingüe como en la detección translingüe de reutilización de códigos fuente.

5.3 Conclusiones

En este capítulo hemos tratado la detección de reutilización en códigos fuente, en concreto siempre detección extrínseca, es decir, la búsqueda de evidencias de reutilización entre códigos de una colección cerrada. El capítulo está dividido en dos grandes apartados: experimentación realizada a nivel monolingüe y la realizada a nivel translingüe.

En la experimentación monolingüe se han realizado diversos experimentos sobre tres corpus distintos. Inicialmente, se trató de identificar si ciertas partes del código fuente permiten una mejor detección de la reutilización, así como de identificar las partes que la dificultan para descartarlas. En concreto, se ha identificado que aunque la modificación de identificadores es un recurso utilizado con frecuencia, gran parte de los identificadores permanecen exactamente igual en los códigos reutilizados (véase la tabla 3.3). Por lo tanto, la eliminación de identificadores o sustitución de estos solo beneficia cuando se han modificado muchos de ellos, mientras que cuando se han realizado otro tipo de modificaciones (por ejemplo, combinación de funciones,) afectará negativamente a la detección de reutilización. A continuación, se realizó una comparación de la herramienta más utilizada, JPlag, con el modelo SoCo-NG sobre un corpus de grandes dimensiones, realizándose la comparación entre el conjunto de pares de códigos fuente de mayor similitud. Se obtuvieron mejores resultados con el modelo SoCo-NG en los tres lenguajes de programación estudiados (C, Java y Python), siendo más notoria la diferencia en códigos fuente de poca longitud. Finalmente, se realizó una comparación de una batería de modelos de distinta naturaleza sobre un mismo corpus para una comparación justa. A partir de esta comparación de modelos distintos, se ha creado un ensamble de los modelos tratando de aprovechar las bondades de cada uno de ellos. A través de los distintos experimentos se ha comprobado que los modelos basados en la representación del código fuente en n-gramas obtienen buenos resultados en detección de reutilización monolingüe. Sin embargo, el ensamble de modelos ha mostrado una ligera mejora respecto a los modelos que lo componen.

En el escenario translingüe también hemos realizado distintos experimentos sobre tres corpus. Primero se realizaron ajustes que mejoren la detección con el modelo SoCo-NG. En concreto, se probaron cambios tanto en el pesado de los n-gramas (tf o $tf - idf$), como en el tipo (palabras o caracteres) y el tamaño (n=$[1-5]$). Se concluyó que se debe considerar el código fuente al completo sin eliminar comentarios ni palabras reservadas y usar un esquema de pesado de trigramas de caracteres con tf para obtener unos resultados óptimos en el modelo SoCo-NG. A continuación se realizó una comparación de modelos para comprobar su eficacia recuperando códigos fuente paralelos y comparables. Siguiendo los resultados obtenidos a nivel monolingüe, los modelos SoCo-NG

y SoCo-LSA obtuvieron los mejores resultados, siendo los del segundo ligeramente superiores. De esta experimentación derivó una simulación de reutilización translingüe considerando los pares de códigos fuente comparables como no reutilizados y los paralelos como reutilizados. A partir de las distribuciones por similitud de estas dos clases se obtuvo un umbral donde ambos modelos distinguen si los pares de códigos fuente han sido reutilizados. Tal como ocurre en el experimento de recuperación de pares comparables y paralelos, SoCo-LSA consigue un mejor desempeño. Finalmente, comparamos ambos modelos en una evaluación sobre un corpus translingüe con casos reales de reutilización. También en esta comparación el modelo SoCo-LSA obtiene mejores resultados que el modelo SoCo-NG.

Capítulo 6

Evaluación en la competición internacional de detección de reutilización en código fuente

En este capítulo se van a describir las dos competiciones internacionales sobre detección de reutilización en código fuente organizadas en el marco de este trabajo de tesis. En concreto, la primera edición (SOCO) se centra en detección a nivel monolingüe mientras que las segunda edición (CL-SOCO) lo hace a nivel translingüe. Ambas competiciones se encuentran descritas en los apartados 6.1 y 6.3. En ambos apartados se presenta la tarea propuesta, los corpus construidos en el marco de la tesis que se han puesto a disposición de las competiciones tanto para entrenar como para evaluar, una breve descripción de los modelos participantes y una comparación de los resultados. Este capítulo también incluye dos apartados que muestran una comparación, a nivel monolingüe (apartado 6.2) y a nivel translingüe (apartado 6.4), de los modelos propuestos en esta tesis con los modelos participantes en ambas competiciones.

6.1 Competición internacional monolingüe SOCO

Del mismo modo que ocurre en la detección de reutilización en textos, la mayoría de trabajos de investigación de detección de reutilización en códigos fuente se realiza sobre colecciones cerradas, generalmente por problemas de privacidad como por ejemplo (Rosales et al., 2008; Prechelt et al., 2002). Además, en la mayoría de casos no se puede realizar una comparación objetiva de distintos modelos por no tener acceso a estos (Clough, 2003). Estas restricciones dificultan la comparación de los modelos sobre un marco de evaluación común.

La competición internacional *Detection of SOurce COde Re-use (SOCO)*[1] (Flores et al., 2014a) pretende establecer un marco de evaluación común en la detección de reutilización de código fuente

[1] `http://www.dsic.upv.es/grupos/nle/soco/`

en el marco del laboratorio *PAN*[2]*: Uncovering Plagiarism, Authorship and Social Software Misuse.* Esta competición se ha celebrado como parte del foro internacional *Forum for Information Retrieval Evaluation (FIRE 2014)*[3] del 5-7 de diciembre 2014 en Bangalore (India).

La tarea SOCO PAN@FIRE se centra en la detección de códigos fuente que han sido reutilizados en un contexto monolingüe, es decir, entre códigos escritos en el mismo lenguaje de programación. Consiste en identificar los pares de códigos fuente reutilizados en una colección a gran escala. En los siguientes apartados se describirá la tarea propuesta, los modelos de los equipos participantes así como los resultados de la competición.

6.1.1 Tarea propuesta

La tarea propuesta en SOCO se centra en la detección de reutilización monolingüe, es decir, el par de código a comparar, original y sospechoso, están escritos en el mismo lenguaje de programación. En consecuencia, los participantes están provistos de un conjunto de códigos fuente escritos en C/C++ y Java, etiquetados por lenguaje de programación. La tarea consiste en la recuperación de pares de códigos fuente que se consideran reutilizados. Cabe destacar que la reutilización es a nivel de código fuente, así pues, no hay que identificar fragmentos de código fuente reutilizados, solo pares de códigos fuente completos. Por lo tanto, los sistemas participantes deben indicar qué códigos fuente presentan casos de reutilización.

La tarea está dividida en dos partes: entrenamiento y evaluación. Es importante destacar que no se exige señalar el sentido en que se ha realizado la reutilización[4]. Por ejemplo, si X es una reutilización de Y, se considerará igual de válido si se han recuperado tanto el par (X, Y) como (Y, X). En la fase de entrenamiento los equipos participantes conocen tanto los casos de reutilización en el corpus así como el lenguaje de programación al que pertenece cada código fuente, mientras que en la fase de evaluación, los equipos participantes solo conocen el lenguaje de programación de cada código.

6.1.2 Corpus

En este apartado se describen los dos corpus utilizados en la competición SOCO. En la fase de entrenamiento, el corpus está compuesto por una colección de códigos fuente escritos en C y Java procedente del ámbito académico. En la fase de evaluación, el corpus esta compuesto por códigos fuente escritos en C/C++ y en Java, procedientes de una competición de programación.

[2] `http://pan.webis.de/`

[3] `http://www.isical.ac.in/~fire/2014/`

[4] Determinar el sentido de la reutilización supone un desafío adicional en la detección de reutilización, es decir, identificar cuál es el documento original y cuál el reutilizado.

Corpus de entrenamiento

Para la construcción de esta colección se ha empleado el corpus A&T++ descrito en el apartado 3.1.3. En ambos lenguajes, C y Java, existen casos de reutilización pero solo a nivel monolingüe. La colección escrita en el lenguaje C consiste en 79 códigos fuente etiquetados desde 000.c a 078.c. La colección escrita en Java contiene 259 códigos fuente etiquetados desde 000.java a 258.java. Los juicios de relevancia representan casos de reutilización en ambos sentidos ($X \rightarrow Y$ y $Y \rightarrow X$). En la tabla 3.4 se muestra el número de pares de códigos fuente etiquetados como casos de reutilización por tres revisores. Con un acuerdo moderado entre anotadores para la colección escrita en C y sustancial en la colección Java según la propuesta de Fleiss (1971). Estos resultados señalan que el conjunto de datos proporcionado representa una colección confiable.

Corpus de evaluación

El corpus de test está compuesto por códigos fuente escritos en los lenguajes C/C++ y Java. Además, los códigos fuente están divididos en escenarios de diferentes temáticas/problemas. Este corpus consiste en un subconjunto de códigos fuente de la colección Google Code Jam descrita en el apartado 3.2.1. Cada lenguaje de programación está dividido en 6 escenarios (B_1, B_2, C_1, C_2, D_1 y D_2). Por lo tanto, el nombre de los códigos fuente consiste en el nombre del escenario al que pertenecen y un identificador. Por ejemplo, el código fuente "B10021" pertenece al escenario B1 y su número identificador es 0021. En la tabla 6.1 se muestra la cantidad de códigos fuente por escenario y lenguaje de programación. Los participantes solo tienen que encontrar casos de reutilización entre los códigos fuente de cada escenario. Por ejemplo, no se considera como caso de reutilización válido el par "B10021" y "B20013". El primer código fuente pertenece al escenario B1 mientras que el segundo al B2. Como se puede observar en la tabla 6.1, la cantidad de códigos fuente de la fase de evaluación es grande, así que los sistemas participantes están obligados en cierta medida a realizar una detección eficiente para resolver la tarea.

C/C++				
Escenario	# códigos	# líneas	# reut.	# reut.(2)
B1	5 408	63,68	99	28
B2	5 195	68,11	86	38
C1	4 939	69,97	86	37
C2	3 873	80,87	43	17
D1	335	180,40	8	4
D2	145	255,21	0	0
Total	19 895	92,70	322	124

Java				
Escenario	# códigos	# líneas	# reut.	# reut.(2)
B1	3 241	99,26	54	53
B2	3 093	107,87	47	45
C1	3 268	90,86	73	73
C2	2 266	102,46	34	35
D1	124	227,12	0	0
D2	88	361,00	14	14
Total	12 080	165,42	222	220

Tabla 6.1: Número de códigos fuente en el corpus de evaluación ordenados por lenguaje de programación y escenario. La tabla muestra el número de códigos fuente (# códigos), el promedio de líneas de código (# líneas) en el escenario respectivo y el número de casos de reutilización identificados *oficiales* (# reut.) y *no oficiales* (# reut. (2)).

Debido a la gran cantidad de códigos fuente de la partición de evaluación es prácticamente imposible el etiquetado manual. Por ello, con el fin de evaluar el desempeño de los sistemas participantes, se han determinado los juicios de relevancia de forma automática a partir de los resultados de los participantes (Sparck et al., 1975). Mediante esta técnica, para que un par de códigos fuente sea

considerado como un juicio de relevancia, es necesario que sea detectado en al menos el 66 % de los resultados de la competición. Por lo tanto, para la construcción de los juicios de relevancia, hemos considerado todos los envíos de los participantes junto con dos sistemas de referencia descritos en el siguiente apartado. Además, en la tabla 6.1 también se muestra el número promedio de líneas de código fuente para cada escenario y lenguaje de programación; este valor indica la dificultad (es decir, cantidad de líneas necesarias) de desarrollo de los programas para resolver una tarea asignada.

Finalmente, es importante mencionar que dos de los cinco participantes enviaron sus resultados posteriormente; por ello, hemos construido dos conjuntos de juicios de relevancia: los oficiales de la tarea y los que contemplan toda la participación. El segundo conjunto considera más envíos para calcular los juicios de relevancia y por lo tanto se considera una selección más robusta que la considerada en la tarea. La inclusión de dos resultados más en el etiquetado automático ha causado un cambio sustancial en los casos reutilizados del lenguaje C/C++. Este cambio es debido a que muchos pares de códigos fuente formaban parte de los juicios de relevancia con valores cercanos al 66 % requerido. Sin embargo, al considerar los nuevos envíos, se ha visto reducido su porcentaje de acuerdo y por lo tanto han quedado excluidos de los juicios de relevancia.

6.1.3 Evaluación

Todos los participantes entregaron de cada envío un archivo con todos los pares de códigos fuentes que sus sistemas consideraban como reutilizados. Los participantes podían entregar cómo máximo hasta tres archivos con sus detecciones. Los archivos de resultados siguen el formato XML que se muestra a continuación. El archivo debe contener una entrada del tipo "$<$ *reuse_case.../* $>$" para cada par de códigos fuentes considerados sospechosos de ser un caso de reutilización. La figura 6.1 muestra un ejemplo de la estructura del archivo XML.

```
<document>
<reuse_case source_code1="X1" source_code2="Y1" />
<reuse_case source_code1="X2" source_code2="Y2" />
...
</document>
```

Figura 6.1: Ejemplo del archivo XML para la entrega de los resultados.

Para evaluar la eficacia de los sistemas se utilizan las medidas Precisión, Cobertura y F_1. Los resultados han sido ordenados utilizando la medida F_1 para favorecer aquellos sistemas que son capaces de obtener valores "altos" y balanceados en Precisión y Cobertura.

Hemos considerado como sistemas de referencia para la competición SOCO los dos sistemas descritos a continuación:

- **Referencia 1.** Consiste en el sistema JPlag (Prechelt et al., 2002) utilizando sus parámetros por defecto. La mayoría de trabajos de investigación se comparan con este sistema debido a que se encuentra disponible vía Web[5]. En este sistema, el código fuente es analizado y convertido

[5]La Web oficial de JPlag es la siguiente: `http://jplag.ipd.kit.edu/`. Sin embargo, los nuevos usuarios deben dirigirse al repositorio (`https://github.com/jplag/`) para descargar y ejecutar localmente la aplicación.

en cadenas de palabras. El algoritmo *greedy-string-tiling* se utiliza para comparar e identificar las cadenas no solapadas comunes más largas. Como valor de similitud devuelve el porcentaje de código que comparten dos códigos fuente. Este modelo considera como reutilizados los primeros 20 pares de códigos fuente con mayor valor de similitud.

- **Referencia 2.** Consiste en el modelo de trigramas de caracteres propuesto en (Flores et al., 2011). En este modelo, el código fuente es considerado como texto y se representa en trigramas de caracteres, donde cada trigrama representa una dimensión en un espacio vectorial y cuyo valor de magnitud es su frecuencia. Como preproceso se eliminan espacios, tabulaciones y saltos de línea. Todos los caracteres se convierten a minúsculas y aquellos caracteres que se repiten en posiciones consecutivas más de tres veces son truncados. La similitud entre dos códigos fuente se calcula utilizando la similitud del coseno. Este modelo de referencia considera a dos códigos fuente como reutilizados si obtienen un valor de similitud mayor que 0,95. Este umbral se ha establecido manualmente sin realizar una prueba empírica. Con este valor de umbral se pretende ofrecer un sistema de referencia que no tenga una tasa elevada de falsos positivos.

6.1.4 Sistemas participantes

En total cinco equipos participantes enviaron 17 ficheros de resultados. En concreto, la *Universidad Autónoma del Estado de México* (`UAEM`) y la *Universidad Autónoma Metropolitana - Unidad Cuajimalpa* (`UAM-C`) enviaron tres ficheros de resultados en ambos lenguajes de programación mientras que la *Dublin City University* (`DCU`) envió tres en Java. El resto de equipos participantes, es decir, `Rajat` y `Apoorv` de la *Indian School of Mines, Dhanbad* enviaron sus resultados después del tiempo oficial de la competición. No obstante, los resultados obtenidos por estos dos últimos han sido considerados para realizar a posteriori el análisis completo que describimos a continuación.

`UAEM` (García-Hernández et al., 2014) utilizaron un modelo para detección de reutilización dividido en cuatro fases. En la primera fase solo se tiene en cuenta elementos léxicos (“”, “”, “(”, “)”, “+”, “*”, “;” etc.) y si hay más de un espacio en blanco contiguo, se sustituye por un único espacio. En la segunda fase se obtiene un valor de similitud entre cada par de códigos fuente. Este valor de similitud consiste en la suma de las longitudes de las subcadenas no contiguas más largas, siendo esta suma normalizada respecto a la longitud del código fuente más largo (García-Hernández et al., 2006). A partir de estos valores de similitud, en una tercera fase se obtendrá un conjunto de parámetros que permitirá identificar los casos de reutilización. Los parámetros obtenidos son: *(i)* el valor de *distancia* (1-similitud); *(ii)* el *ranking* ordenado de las distancias; *(iii)* la *diferencia* entre el siguiente en el ranking (solo se calcula para los diez pares más cercanos); y *(iv)* utilizando la diferencia máxima entre los diez pares más cercanos, se etiquetan los pares de códigos fuente según si están *antes* o *después* de esta máxima diferencia. La decisión de que un par de códigos fuente se considere como un caso de reutilización, se toma en la cuarta fase. Existe evidencia de reutilización en ambos sentidos, si tanto $X \rightarrow Y$ como $Y \rightarrow X$ son pares sospechosos. Un caso de reutilización existe si la *distancia* es menor que 0,45 y la *diferencia* mayor que 0,14, pero también es importante que se cumpla una de las dos condiciones siguientes: *(i)* el par de códigos fuente debe situarse en la primera o segunda posición en el *ranking* y, *(ii)* debe tener la etiqueta de *antes*. El primer sistema que presentaron consideró las condiciones anteriores. Sin embargo, en algunos casos la evidencia en un sentido es mucho mayor que en el otro, siendo estos dos pares un caso de reutilización. En

el segundo sistema, si no existe una evidencia de reutilización elevada en un sentido, el par puede considerarse reutilización si un código está situado primero en el *ranking* y la *diferencia* entre ambos códigos en *distancia* es mayor que 0,1.

`UAM-C` (Ramírez-de-la-Cruz et al., 2014b; Ramírez-de-la-Cruz et al., 2015) representaron el código fuente en tres vistas intentando remarcar distintos aspectos del código: vista *léxica*, vista *estructural* y vista *estilística*. En la vista léxica, tras un preproceso previo donde se eliminan las palabras reservadas del lenguaje de programación, espacios en blanco y todos los caracteres se convierten a minúsculas, se representa el código fuente mediante bolsas de 3-gramas de caracteres. Para la vista estructural, proponen dos medidas de similitud que tienen en cuenta las características de las funciones, por ejemplo, los tipos de datos y los identificadores de sus parámetros. La tercera vista consiste en características de estilo, como, el número de espacios, número de líneas con algún caracter en mayúscula, etc. Para cada par de códigos fuente del corpus de entrenamiento se calculan los valores de similitud de cada vista y, posteriormente, se establece un umbral donde se consigue el mayor valor de F_1. En el primer sistema presentado solo consideraron la vista léxica con un umbral de similitud establecido manualmente con valor 0,5. En el segundo sistema utilizaron la vista léxica y la vista estructural. De estas vistas se calculan tres similitudes: *(i)* similitud léxica (L), *(ii)* similitud de los tipos de datos (DT), y *(iii)* similitud de los nombres de los identificadores (IN). Estos valores de similitud se combinan mediante la siguiente fórmula: $0{,}5L \times 0{,}25DT \times 0{,}23IN$, según unos niveles de confianza establecidos manualmente. En el tercer sistema utilizaron 8 similitudes: una a partir de la vista léxica, 6 a partir de la vista estructural y otra a partir de la vista estilística. Finalmente, se utiliza una aproximación supervisada para determinar si existe reutilización, en concreto el modelo J48 de árboles de decisión de la herramienta Weka.

`DCU` (Ganguly et al., 2014; Ganguly et al., 2015) para la tarea de detección de reutilización en código fuente llevó a cabo una aproximación basada en la recuperación de información. Primero, se analiza el código fuente en Java para construir un árbol sintáctico abstracto (AST). A continuación, se extrae los contenidos de ciertos campos del AST guardándolos en un índice de Lucene[6]. En concreto, los nodos del AST de los que se extrae información son *declaraciones*, *nombres de clases*, *nombres de funciones*, *cuerpo de funciones*, *cadenas*, *vectores* y *comentarios*. Cada código se compara con el resto de la colección. Para las comparaciones solo se consideran los nombres de las clases, nombre de los métodos y parámetros, valores de las cadenas, nombres de los vectores, nombres de variables, nombre de los paquetes utilizados y los comentarios. De esta comparación se obtiene una lista ordenada de códigos fuente con sus respectivos valores de similitud. El modelo de recuperación utilizado es un modelo de lenguaje. El modelo recorre la lista en orden decreciente de similitud y se detiene cuando la diferencia relativa con el código fuente anterior es menor que un umbral predefinido (y no especificado por los autores). Los códigos fuentes recuperados se consideran códigos fuente reutilizados. En el primer sistema presentado consideraron por separado los nodos según el tipo, por ejemplo, los nombres de clases y los nombres de métodos. Se calculan frecuencias de términos para cada tipo de campo extraído del AST por separado. En el segundo sistema crearon el AST y, a partir de los nodos seleccionados, se construye una bolsa de palabras. En este caso los campos extraídos del AST no se consideran por separado, por lo que se construye un único índice. En el tercer sistema construyeron una simple bolsa de palabras para representar los códigos fuente sin tener en cuenta la estructura de estos.

[6] `https://lucene.apache.org/core/`

`Rajat` propuso un enfoque basado en la comparación de cadenas. El método propuesto busca una coincidencia exacta de líneas entre un par de códigos fuente. La métrica de similitud utilizada consiste en calcular una relación entre el número de líneas en común y el número total de líneas del archivo de código fuente más grande. Al final, el proceso de decisión para determinar si es o no un caso de reutilización se basa en el valor de similitud obtenido debiendo superar un umbral del 70 %. (no especificado el cálculo por sus autores).

`Apoorv` propuso un enfoque similar al método descrito por `Rajat`. En consecuencia, este enfoque considera los códigos fuente como documentos de texto, y sigue una estrategia de comparación de cadenas para medir similitudes. En concreto, busca líneas exactas que emparejan entre los códigos fuente y estima un valor de similitud en relación a las líneas que ambos códigos fuente tienen en común. Los pares de códigos fuente seleccionados como casos de reutilización son los que obtienen el máximo valor de similitud. Al final, para cada código fuente dentro del conjunto de evaluación, el enfoque enumera un solo caso de reutilización con el valor máximo de similitud.

6.1.5 Resultados

Los resultados obtenidos por los sistemas participantes se muestran para cada lenguaje de programación en cada escenario según los juicios de relevancia considerados (*oficiales* y *no oficiales*). Los juicios de relevancia *oficiales* consideran los casos de reutilización detectados por los sistemas que hicieron sus envíos dentro del plazo indicado. Los juicios de relevancia *no oficiales* consideran los casos de reutilización detectados por todos los participantes incluso los envíos fuera de plazo. Los resultados globales de cada sistema se presentan ordenados mediante el valor de F_1, dado que se valora un equilibrio entre los valores de Precisión P y Cobertura R.

Resultados considerando los juicios de relevancia *oficiales*

Equipo	Sistema	Escenarios				
		B1	B2	C1	C2	D1
UAEM	1	**0,382**	**0,372**	**0,587**	**0,531**	0,552
	2	0,321	0,309	0,581	0,521	0,485
UAM-C	1	0,010	0,009	0,024	0,019	0,762
	2	0,008	0,008	0,020	0,007	**0,800**
	3	0,010	0,009	0,024	0,019	0,737
Referencia	1	0,101	0,113	0,245	0,349	0,429
	2	0,294	0,255	0,326	0,323	0,333
Apoorv	1	0,035	0,023	0,022	0,014	0,029
Rajat	1	0,110	0,106	0,236	0,146	0,066

Resultados considerando los juicios de relevancia *no oficiales*

Equipo	Sistema	Escenarios				
		B1	B2	C1	C2	D1
UAEM	1	0,125	0,184	0,303	0,250	0,320
	2	0,103	0,150	0,300	0,245	0,276
UAM-C	1	0,003	0,004	0,011	0,007	0,471
	2	0,002	0,003	0,009	0,002	0,500
	3	0,003	0,004	0,011	0,007	**0,533**
Referencia	1	0,125	0,069	**0,386**	**0,486**	0,333
	2	**0,264**	**0,256**	0,340	0,388	0,500
Apoorv	1	0,023	0,015	0,014	0,009	0,024
Rajat	1	0,081	0,115	0,273	0,155	0,068

Tabla 6.2: Valor de F_1 por escenario considerando tanto los juicios de relevancia utilizados en la tarea SOCO (es, decir, los oficiales) como los calculados considerando los 5 equipos participantes (no oficiales) en el lenguaje de programación C/C++.

Para el lenguaje de programación C/C++ (tabla 6.2), observando los resultados oficiales, el equipo `UAEM` obtuvo un buen desempeño para todos los escenarios con excepción de D1. Específicamente para D1, que es la tarea más compleja, los mejores resultados fueron obtenidos por el equipo `UAM-C`. Este comportamiento señala, en cierta medida, que la metodología propuesta por `UAEM` es

adecuada para tareas fáciles y de complejidad moderada, mientras que para las tareas de alto nivel de dificultad, la aproximación de UAM-C tiene mejor desempeño. Considerando los juicios de relevancia no oficiales, el mejor rendimiento en todos los escenarios es obtenido por los modelos de referencia a excepción de la tarea D1, donde otra vez, UAM-C obtiene mejor rendimiento.

Resultados considerando los juicios de relevancia *oficiales*						
Equipo	Sistema	Escenarios				
		B1	B2	C1	C2	D2
UAEM	1	0,514	0,519	0,613	0,523	N/A
	2	0,250	0,234	0,324	0,248	0,500
UAM-C	1	0,474	0,452	0,616	0,447	0,824
	2	0,755	0,058	0,021	0,027	0,111
	3	**0,776**	**0,739**	**0,847**	**0,815**	**1,000**
DCU	1	0,600	0,564	0,652	0,581	0,596
	2	0,701	0,681	0,727	0,673	0,636
	3	0,667	0,676	0,702	0,687	0,667
Referencia	1	0,324	0,388	0,237	0,556	0,824
	2	0,529	0,537	0,559	0,568	0,667
Apoorv	1	0,025	0,021	0,038	0,025	0,275
Rajat	1	0,352	0,337	0,540	0,369	0,718

Resultados considerando los juicios de relevancia *no oficiales*						
Equipo	Sistema	Escenarios				
		B1	B2	C1	C2	D2
UAEM	1	0,507	0,503	0,613	0,534	N/A
	2	0,246	0,226	0,324	0,255	0,500
UAM-C	1	0,467	0,437	0,616	0,458	0,824
	2	0,746	0,055	0,021	0,028	0,111
	3	**0,797**	**0,769**	**0,847**	**0,829**	**1,000**
DCU	1	0,570	0,534	0,643	0,576	0,596
	2	0,693	0,662	0,727	0,686	0,636
	3	0,658	0,657	0,702	0,700	0,667
Referencia	1	0,301	0,369	0,237	0,545	0,824
	2	0,519	0,496	0,559	0,562	0,667
Apoorv	1	0,027	0,023	0,038	0,026	0,275
Rajat	1	0,384	0,376	0,560	0,404	0,718

Tabla 6.3: Valor de F_1 por escenario considerando tanto los juicios de relevancia utilizados en la tarea SOCO (es, decir, los oficiales) como los calculados considerando los 5 equipos participantes (no oficiales) en el lenguaje de programación Java.

Como se explicó anteriormente, ambos modelos de referencia y el propuesto por UAEM están basados en estrategias de PLN para identificar casos de reutilización. Por el contrario, la aproximación utilizada por UAM-C considera algunas características que no están basadas en el contenido (llamadas características estructurales en Rosales et al., 2008). Por lo tanto, parece que las aproximaciones basadas en PLN son suficientemente buenas para códigos fuente de tamaño reducido, mientras que para códigos más elaborados (tareas más complejas) las características estructurales permiten identificar con precisión casos de reutilización de código fuente.

Los resultados globales de la tabla 6.4 muestran que, los dos sistema del equipo UAEM recuperaron todos los casos de reutilización (alta Cobertura). La regla introducida para recuperar casos de reutilización menos obvios tiene un impacto negativo en su rendimiento en términos de Precisión y por lo tanto F_1. Los resultados del equipo UAM-C se vieron afectados negativamente en términos de Precisión por el elevado número de pares de códigos fuente recuperados (más de 50 000). Esto se debe al preproceso de eliminación de palabras reservadas y considerar los parámetros estimados sobre el corpus del lenguaje C. El lenguaje C++ incluye nuevas características como clases y métodos así como nuevas palabras reservadas, por ejemplo *cin* o *cout*.

Para el lenguaje Java, UAM-C obtuvo el mejor rendimiento manteniendo el equilibrio entre Precisión y Cobertura tanto en los resultados *oficiales* como en los resultados *no oficiales*. Según los resultados obtenidos, la combinación de todas sus medidas de similitud propuestas (léxica, estructural y estilística) utilizando un árbol de decisión supervisado ha resultado decisiva. Sin embargo, su segundo sistema se ha visto afectado por el mismo efecto que en el escenario C/C++: recupera más

Resultados considerando los juicios de relevancia *oficiales*				
Equipo	Sistema	Medidas de evaluación		
		F_1	P	R
UAEM	1	**0,440**	0,282	1,000
	2	0,387	0,240	1,000
UAM-C	1	0,013	0,006	1,000
	2	0,010	0,005	0,950
	3	0,013	0,006	0,997
Referencia	1	0,190	0,350	0,130
	2	0,295	0,258	0,345
Apoorv	1	0,022	0,011	0,543
Rajat	1	0,129	0,077	0,404

Resultados considerando los juicios de relevancia *no oficiales*				
Equipo	Sistema	Medidas de evaluación		
		F_1	P	R
UAEM	1	0,196	0,109	1,000
	2	0,169	0,092	1,000
UAM-C	1	0,005	0,002	1,000
	2	0,004	0,002	0,935
	3	0,005	0,002	1,000
Referencia	1	0,238	0,242	0,234
	2	**0,300**	0,193	0,669
Apoorv	1	0,014	0,007	0,903
Rajat	1	0,126	0,068	0,927

Tabla 6.4: Resultados globales correspondientes al lenguaje C considerando tanto los juicios de relevancia *oficiales* como los *no oficiales*.

Resultados considerando los juicios de relevancia *oficiales*				
Equipo	Sistema	Medidas de evaluación		
		F_1	P	R
UAEM	1	0,556	0,385	1,000
	2	0,273	0,158	1,000
UAM-C	1	0,517	0,349	1,000
	2	0,037	0,019	0,928
	3	**0,807**	0,691	0,968
DCU	1	0,602	0,432	0,995
	2	0,692	0,530	0,995
	3	0,680	0,515	1,000
Referencia	1	0,380	0,542	0,293
	2	0,556	0,547	0,712
Apoorv	1	0,029	0,015	0,811
Rajat	1	0,418	0,301	0,680

Resultados considerando los juicios de relevancia *no oficiales*				
Equipo	Sistema	Medidas de evaluación		
		F_1	P	R
UAEM	1	0,552	0,381	1,000
	2	0,271	0,157	1,000
UAM-C	1	0,513	0,345	1,000
	2	0,037	0,019	0,927
	3	**0,821**	0,701	0,991
DCU	1	0,688	0,525	0,995
	2	0,585	0,418	0,973
	3	0,676	0,510	1,000
Referencia	1	0,371	0,525	0,286
	2	0,544	0,445	0,700
Apoorv	1	0,031	0,016	0,855
Rajat	1	0,447	0,321	0,732

Tabla 6.5: Resultados globales correspondientes al lenguaje Java considerando tanto los juicios de relevancia *oficiales* como los *no oficiales*.

de 10 000 pares de códigos fuente como reutilizados, afectando a su valor de Precisión[7]. Los tres envíos del equipo `DCU` obtuvieron un buen rendimiento en general (véase la tabla 6.3). Su segundo sistema en concreto, la inclusión de la *bolsa de palabras* para seleccionar nodos del *AST* mejora ligeramente el rendimiento del primer sistema. Mientras que sus resultados empeoran ligeramente en el tercer sistema, donde no se utiliza la *bolsa de palabras* para seleccionar los nodos dentro del *AST*.

[7] En las tablas 6.4 y 6.5 que ofrecen valores de Precisión y Cobertura se observa el efecto de recuperar un gran cantidad de falsos positivos.

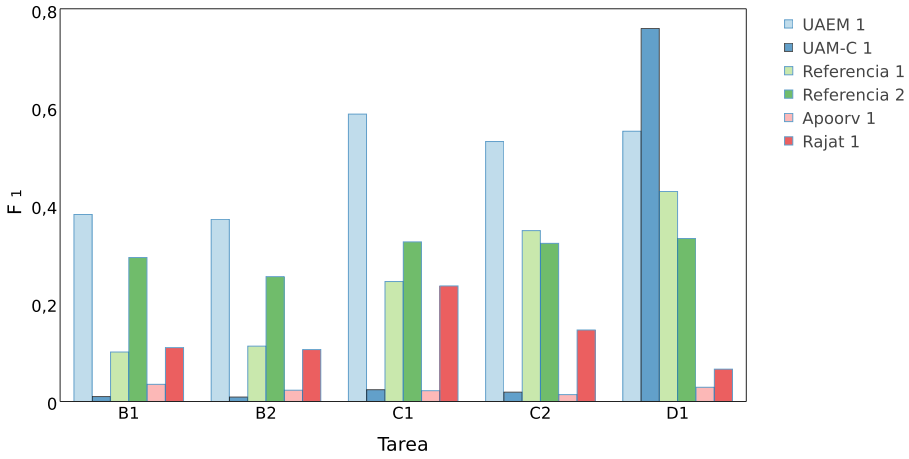


Figura 6.2: Comparación de los mejores sistemas en el lenguaje C/C++.

Como se observa en las figuras 6.2 y 6.3, la mayoría de las aproximaciones propuestas obtienen mejores valores de F_1 cuando la dificultad del problema es elevada, es decir, los códigos fuentes son más largos. Esto lleva a la conclusión que las tareas de dificultad menor, y que requieren menos líneas de código fuente para ser resueltas, muestran menos evidencias de que existe reutilización entre sus códigos. Por lo tanto, se considera las tareas de menor complejidad como escenarios más desafiantes para detectar reutilización de código fuente.

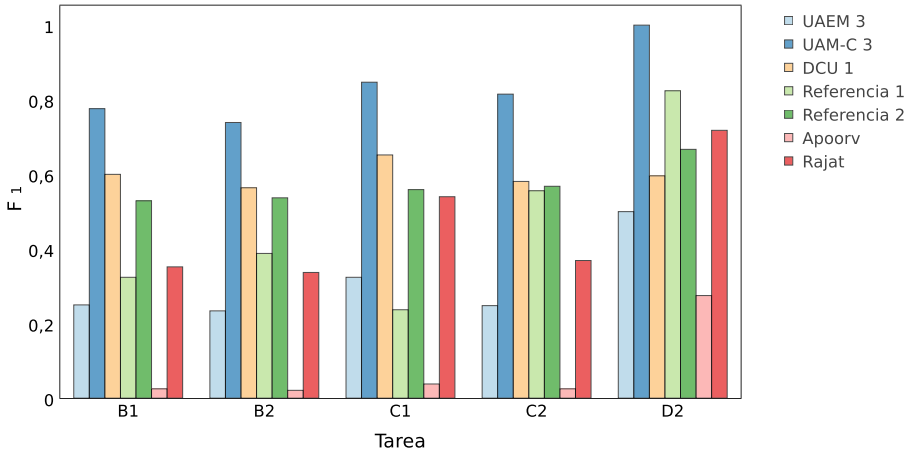


Figura 6.3: Comparación de los mejores sistemas en el lenguaje Java.

En conclusión, para resolver el problema de la detección de reutilización se aplicaron aproximaciones diferentes. Las propuestas varían desde comparación de cadenas de caracteres hasta sistemas basado en el árbol sintáctico creado por el compilador. Evaluar todas estas aproximaciones bajo

las mismas condiciones y utilizando las mismas colecciones de códigos fuente ha permitido realizar una comparación entre aproximaciones tan distintas. En la tabla 6.4 y 6.5 se presentan los resultados globales de cada aproximación para el lenguaje C/C++ y Java respectivamente. Considerando los juicios de relevancia oficiales, la aproximación con mejor desempeño en el lenguaje C/C++ ha sido el sistema 1 del equipo UAEM basado en la comparación de cadenas (García-Hernández et al., 2014), mientras que el sistema 3 que está basado en la combinación de características léxicas, estructurales y estilísticas lo ha sido para el lenguaje Java (Ramírez-de-la-Cruz et al., 2015). Sin embargo, considerando los resultados *no oficiales*, el modelo de referencia 2, basado en la comparación de trigramas de caracteres, obtuvo el mejor desempeño para el escenario C/C++, mientras que el sistema 3 del equipo `UAM-C` se mantiene como mejor sistema en el lenguaje Java, siempre realizando la comparación en términos de valores de F_1.

6.2 Comparación con la propuesta monolingüe

En este apartado se describe cómo se han preparado los modelos propuestos en esta tesis que obtuvieron los mejores resultados a nivel monolingüe con el fin de entrenar y evaluar en las mismas condiciones que los sistemas participantes de la tarea SOCO. En el primer apartado se detalla cómo se entrenaron los modelos SoCo-NG y SoCo-LSA, mientras que en el segundo ambos se comparan con los sistemas que participaron en la competición.

6.2.1 Entrenamiento y ajuste de los modelos para SOCO

Durante la experimentación monolingüe de esta tesis hemos comprobado cómo los modelos SoCo-NG y SoCo-LSA obtienen los mejores resultados en detección de reutilización de códigos fuente. Así pues, estos dos modelos se han comparado en pocas ocasiones con otros del estado de la cuestión (tan solo con JPlag en el entorno masivo) por dos razones: *(i)* los trabajos de otros autores se realizan sobre colecciones de códigos fuente no disponibles para realizar comparación; y *(ii)* los modelos no se encuentran perfectamente definidos para reproducir sus resultados, por ejemplo el del trabajo Marinescu et al. (2013) donde realizan un filtrado de palabras reservadas del lenguaje, patrones, librerías que no se comentan explícitamente cuáles son. Por ello, en el marco de esta tesis surgió la iniciativa de crear una competición internacional para solventar ambos problemas para la comunidad científica. Para poder realizar una comparación de los modelos propuestos en esta tesis de investigación con los modelos de los participantes de la competición, en este apartado detallamos la fase de entrenamiento y ajuste de los modelos SoCo-NG y SoCo-LSA, y posteriormente en el apartado 6.2.2, se mostrarán los resultados obtenidos. Para la fase de entrenamiento se ha utilizado del corpus A&T++ descrito en el apartado 3.1.3 así como se realizó en la competición. En concreto los códigos fuente que contienen reutilización monolingüe en los lenguajes C y Java.

Para entrenar el modelo SoCo-NG como si de un modelo participante se tratase, es necesario establecer un criterio que separe los pares de códigos fuente entre reutilizados y no reutilizados. Dado que el modelo ofrece un valor numérico de similitud, vamos a establecer un umbral empírico a través de una validación cruzada de diez particiones. En cada partición se establecerá un umbral donde el valor de la medida F_1 se maximice. En la figura 6.4 se muestra el valor de Precisión, Cobertura y F_1 en función del valor de umbral establecido en el corpus de entrenamiento Java.

Finalmente, para la evaluación final, se utilizará como umbral la media de los umbrales establecidos en las diez particiones.

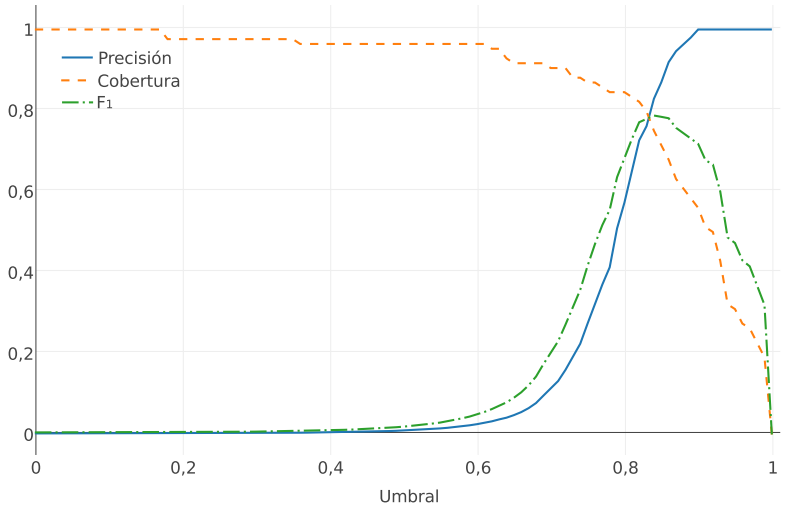


Figura 6.4: Curvas de Precisión, Cobertura y F_1 para el modelo SoCo-NG en la fase de entrenamiento con el corpus Java.

El entrenamiento del modelo SoCo-LSA también se ha realizado mediante validación cruzada de diez particiones. En este modelo se han utilizado los pares de códigos fuente de nueve particiones para ser representados en el espacio vectorial reducido y se ha calculado la similitud entre los pares de la partición de validación. Del mismo modo que con el modelo SoCo-NG, en cada validación se ha estimado un umbral de similitud para separar los reutilizados y no reutilizados, siendo la media de estos umbrales, el umbral que se utilizará para la comparación con los modelos de la competición. En la figura 6.5 se muestra el valor de Precisión, Cobertura y F_1 según el valor de umbral establecido.

Durante la fase de entrenamiento se ha establecido como el mejor umbral para el modelo SoCo-NG en el lenguaje Java de un valor 0,84 mientras que para el modelo SoCo-LSA un umbral de valor 0,93. Para el lenguaje C se han establecido los umbrales 0,97 y 0,92 para los modelos SoCo-NG y SoCo-LSA respectivamente. No se muestran las curvas de Precisión, Cobertura y F_1 para el lenguaje C porque ambas tienen un comportamiento similar a las del lenguaje Java y no aportan más información que las ya mostradas.

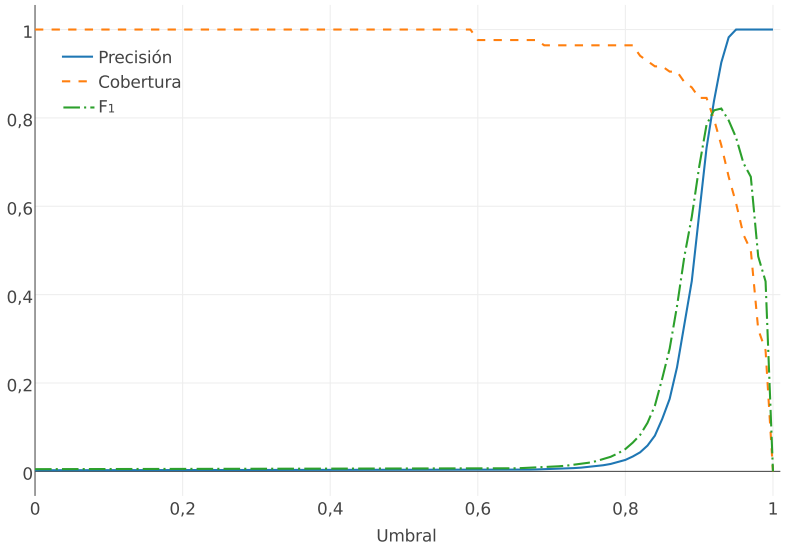


Figura 6.5: Curvas de Precisión, Cobertura y F_1 para el modelo SoCo-LSA en la fase de entrenamiento con el corpus Java.

6.2.2 Comparación de los modelos propuestos con la competición

En este apartado comparamos los modelos propuestos en esta tesis con los de la competición SOCO. Esta comparación se realiza empleando las mismas medidas que se utilizaron en la competición: la Precisión, la Cobertura y F_1. Para realizar esta comparación hemos considerado los dos modelos que han mostrado mejores prestaciones durante la experimentación realizada, SoCo-NG y SoCo-LSA. Tanto en los experimentos sobre los modelos SoCo-NG y SoCo-LSA así como en la competición, el objetivo es obtener buenas prestaciones en términos de la medida F_1. Ambos modelos están diseñados para estimar la similitud entre todos los pares de códigos fuente de una colección, mientras que en la competición SOCO se requiere determinar si un par de códigos fuente es considerado como reutilizado o no. La adaptación de los dos modelos se ha realizado empleando el corpus de entrenamiento (véase el apartado 6.1.2).

En el caso del modelo SoCo-NG, hemos considerado el código fuente al completo junto con la utilización de trigramas de caracteres y tf como método de pesado. Además, se ha realizado una validación cruzada para establecer un umbral de similitud que maximice el valor de F_1. El umbral para el modelo SoCo-NG queda establecido en el valor de 0,97 y 0,84 para el corpus C/C++ y Java respectivamente.

En el modelo SoCo-LSA también hemos utilizado la configuración estimada como óptima durante esta tesis. Esta consiste en dividir cada código fuente en trigramas de caracteres, pesar estos trigramas con tf y proyectar cada código fuente en un espacio vectorial reducido de 256 dimensiones (ver apartado 4.5). También hemos utilizado validación cruzada para estimar el umbral de similitud maximizando el valor de F_1. El umbral para el modelo SoCo-LSA queda establecido en el valor de 0,92 y 0,93 para el corpus C/C++ y Java respectivamente.

Debido a la variabilidad de los juicios de relevancia en el corpus de evaluación de la competición al considerar nuevos envíos, se ha optado por realizar esta comparación sobre el corpus de entrenamiento que se etiquetó manualmente. Además, todos los equipos participantes disponían de antemano de los juicios de relevancia, ajustando sus modelos sobre estos. Así pues, los resultados obtenidos en este apartado tanto por los modelos propuestos así como los resultados de los modelos de la competición han sido entrenados y evaluados bajo las mismas condiciones.

En las tablas 6.6 y 6.7 se muestran los resultados obtenidos sobre los corpus de entrenamiento de la competición SOCO de los modelos SoCo-NG y SoCo-LSA comparados con el resto de aproximaciones de la competición ordenados según el valor de F_1. Respecto al corpus escrito en el lenguaje C/C++, el modelo SoCo-LSA se sitúa con un valor de F_1 de 0,838 por encima de las aproximaciones de la competición superando por 0,088 a la mejor aproximación de la competición. Por otra parte, el modelo SoCo-NG obtiene buenos resultados consiguiendo un valor de F_1 de 0,762 también superando ligeramente la mejor aproximación de la competición, en concreto con una diferencia de 0,012 de valor de similitud.

	Precisión	Cobertura	F1	Umbral
SoCo-LSA	0,923	0,768	**0,838**	0,92
SoCo-NG	0,941	0,640	**0,762**	0,97
UAM-C	0,818	0,692	0,750	—
UAEM	0,656	0,808	0,724	—
Referencia 2	0,800	0,615	0,696	0,95

Tabla 6.6: Resultados del corpus de entrenamiento escrito en C de la competición SOCO junto con los modelos SoCo-NG y SoCo-LSA. El ranking se encuentra ordenado según el valor de F_1.

	Precisión	Cobertura	F1	Umbral
SoCo-LSA	0,925	0,738	**0,821**	0,93
SoCo-NG	0,829	0,750	**0,788**	0,84
UAM-C	0,723	0,714	0,719	—
DCU	0,684	0,619	0,650	—
UAEM	0,404	0,857	0,550	—
Referencia 2	1,000	0,310	0,473	0,95

Tabla 6.7: Resultados del corpus de entrenamiento escrito en Java de la competición SOCO junto con los modelos SoCo-NG y SoCo-LSA. El ranking se encuentra ordenado según el valor de F_1.

En el corpus escrito en el lenguaje Java, el modelo SoCo-LSA también se sitúa por encima de las aproximaciones de la competición con un valor de F_1 de 0,821 superando por 0,102 a la mejor aproximación. También el modelo SoCo-NG obtiene buenos resultados consiguiendo un valor de F_1 de 0,788 así superando ligeramente la mejor aproximación de la competición, con una diferencia de 0,069 de valor de similitud.

En este apartado se han comparado los modelos SoCo-LSA y SoCo-NG bajo las mismas condiciones que las aproximaciones presentadas a la competición SOCO. Ambos modelos han demostrando un buen desempeño, obteniendo el modelo SoCo-LSA los mejores resultados y el modelo SoCo-NG

superando ligeramente a la mejor aproximación presentada en la competición para cada lenguaje de programación. Es importante destacar que ambos modelos han obtenido unos valores de Precisión elevados junto a una Cobertura medianamente elevada. Esto indica que los modelos consiguen detectar con precisión los pares reutilizados y obtienen una gran cantidad de pares reutilizados.

6.3 Competición internacional translingüe CL-SOCO

La competición CL-SOCO (Flores et al., 2015a) surge de la idea de llevar un paso más allá la anterior edición, SOCO (Flores et al., 2014a; Flores et al., 2015d), descrita en el apartado 6.1. Así pues, si la anterior edición se enfocó en la detección de reutilización de código fuente a nivel monolingüe, CL-SOCO se centra en la detección a nivel translingüe. El estudio de este tipo de reutilización responde a un interés reciente (Arwin et al., 2006; Flores et al., 2015b). Se puede producir cuando en un trabajo de programación a evaluar por el profesor, un alumno encuentra la solución a un problema que debe resolver implementada en un lenguaje de programación distinto al requerido. Dicho profesor necesitará herramientas capaces de detectar una copia fraudulenta entre lenguajes de programación por parte de su estudiante. Otra posible situación a la detección de plagio, es la de recuperación de información, donde un programador dispone de un código en un lenguaje concreto pero lo necesita en uno distinto. La búsqueda automática de códigos fuente entre distintos lenguajes de programación es una solución para resolver esta problemática costosa tanto en tiempo como económica comparado con la traducción manual de códigos fuente. Este escenario supone un reto mayor que en la detección de reutilización monolingüe porque los lenguajes de programación no comparten las mismas palabras reservadas, ni las librerías, ni tampoco la sintaxis.

Al igual que ocurre a nivel monolingüe, los trabajos de detección translingüe se realizan sobre colecciones cerradas (Arwin et al., 2006) con dificultad en el acceso a estas colecciones por problemas de privacidad. Por ello, la competición internacional *Cross-Language Detection of SOurce COde Re-use (CL-SOCO)*[8] pretende establecer un marco de evaluación común en la detección de reutilización translingüe de código fuente como una actividad de laboratorio *PAN: Uncovering Plagiarism, Authorship and Social Software Misuse*. Esta competición se celebra como parte del foro internacional *Forum for Information Retrieval Evaluation (FIRE 2015)* del 4-6 de Diciembre 2015 en Gandhinagar (India).

La tarea CL-SOCO PAN@FIRE se centra en la detección de códigos fuente que han sido reutilizados en un contexto translingüe, es decir, entre códigos escritos en distintos lenguajes de programación, en concreto C y Java por ser dos de los lenguajes más utilizados actualmente. Consiste en identificar y distinguir los pares de códigos fuente más similares entre una colección de códigos fuente. En los siguientes apartados se resumirán los objetivos, se describirá la tarea propuesta, las aproximaciones de los equipos participantes así como los resultados durante la competición.

[8] `http://www.dsic.upv.es/grupos/nle/clsoco/`

6.3.1 Tarea propuesta

La tarea CL-SOCO se centra en la detección translingüe de código fuente. Esto significa que las aproximaciones participantes deben tratar casos donde el código fuente sospechoso y el original están escritos en distintos lenguajes de programación. Los participantes están provistos de un conjunto de códigos fuente escritos en ambos lenguajes de programación, C y Java, los cuales se han separado por lenguaje de programación para facilitar la detección. Así pues, la tarea consiste en recuperar pares de códigos fuente que se consideran reutilizados. Es importante destacar que la detección se debe realizar a nivel de documento, no se espera detectar fragmentos específicos dentro del código fuente. Por lo tanto, las aproximaciones deben reportar qué pares de códigos fuente consideran reutilizados (e indirectamente los que se consideran como no reutilizados).

La tarea está dividida en dos fases: entrenamiento y evaluación. Para la fase de entrenamiento se dispone de un corpus anotado en los lenguajes C y Java. Esta anotación incluye información sobre qué pares son considerados como reutilizados y, en este caso, el código fuente original es el escrito en el lenguaje C. Aunque se conozca el sentido en el que se ha cometido la reutilización al igual que para la tarea monolingüe SOCO, solo se considera relevante detectar los pares reutilizados sin importar el sentido, por ejemplo, si el código fuente X ha sido reutilizado por Y, se considera como válido reportar tanto el par (X, Y) como el par (Y, X). En cambio, en la fase de evaluación, los participantes solo conocen cuál es el lenguaje de programación en que está escrito cada código fuente.

6.3.2 Corpus

En este apartado se describen las dos colecciones utilizados en la competición CL-SOCO. Tanto en la fase de entrenamiento así como en la de evaluación, el corpus está compuesto por códigos fuente escritos en los lenguajes C y Java. En ambas fases, la reutilización translingüe de códigos fuente fue realizada automáticamente por medio de un traductor automático de código fuente *C++ to Java Converter*. La herramienta *C++ to Java Converter* es capaz de refactorizar el código fuente si fuera necesario.

Corpus de entrenamiento

La colección de entrenamiento consiste en códigos fuente escritos en el lenguaje de programación C y posteriormente traducidos a Java. Para la construcción de esta colección se ha empleado el repositorio Rosettacode.org ya comentada en el apartado 3.2.2. `Rosettacode.org` es un sitio Web que presenta soluciones para la misma tarea en tantos lenguajes de programación como sea posible. En concreto, Rosettacode.org cuenta actualmente con implementaciones en casi 600 lenguajes de programación. Se extrajeron 599 códigos fuente de tareas resueltas en el lenguaje de programación C. En la tabla 6.8 se resumen las características del corpus de entrenamiento.

Corpus de entrenamiento				
Leng. de progr.	# códigos fuente	Tokens	Líneas	# casos reut.
C	599	86 001	25 226	599
Java	599	102 239	31 713	

Tabla 6.8: Características del corpus de entrenamiento. Se muestran valores de la cantidad de tokens tras eliminación de símbolos de puntuación, el número de líneas de código fuente y la cantidad de casos de reutilización entre las colecciones C y Java.

Corpus de evaluación

El corpus de evaluación proporcionado se ha creado a partir del corpus C descrito en el apartado 3.1.3 utilizado en la fase de entrenamiento de la edición de 2014 de la competición SOCO, y también en el trabajo de Arwin et al. (2006). A partir de 79 códigos fuente en el lenguaje de programación C, se generó un conjunto de 79 códigos fuente Java automáticamente traducidos. Cada uno de los 79 códigos fuente en C y su traducción suponen un caso de reutilización translingüe, que a partir de ahora será llamado *reutilización traducida*. La colección de códigos fuente en C tiene una particularidad de origen, contiene reutilización monolingüe entre sus códigos. Por ello, esta reutilización se ha extendido entre lenguajes cuando se han traducido los códigos fuente. Este tipo de reutilización será llamado *reutilización propagada*. Por ejemplo, si el par $A.c \leftrightarrow B.c$ era un caso de reutilización monolingüe en la colección original en C, este caso generará dos nuevos casos de reutilización translingüe después del proceso de traducción $A.c \rightarrow B.java$ y $B.c \rightarrow A.java$ además de los casos de *reutilización traducida* que se generaban con la traducción automática $A.c \rightarrow A.java$ y $B.c \rightarrow B.java$. En la figura 6.6 se muestra un ejemplo de los tipos existentes de reutilización descritos anteriormente. Para la tarea de detección de reutilización translingüe solo se considerarán los casos de reutilización *traducida* y *propagada*. En total 131 casos de reutilización son considerados para la tarea CL-SOCO, 79 instancias de *reutilización traducida* y 52 de *reutilización propagada*. En la tabla 6.9 se resumen las características del corpus de evaluación.

Leng. de prog.	# códigos fuente	Tokens	Líneas	# casos reut.
C	79	13 171	7 169	131
Java	79	15 829	7 144	

Tabla 6.9: Características del corpus de evaluación. Se muestran valores de la cantidad de tokens tras eliminación de símbolos de puntuación, el número de líneas de código fuente y la cantidad de casos de reutilización entre las colecciones C y Java.

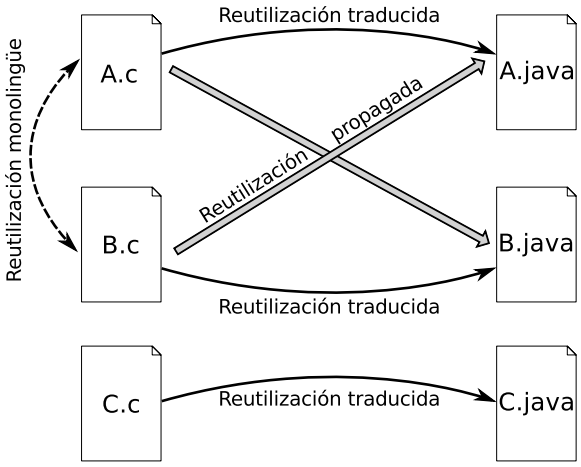


Figura 6.6: Ejemplo de los tipos de reutilización de código fuente.

6.3.3 Evaluación

Todos los participantes entregaron de cada envío un archivo con todos los pares de códigos fuentes que sus sistemas consideraban como reutilizados. Cómo máximo pudieron entregar hasta tres archivos con sus detecciones. Los archivos de resultados siguen el formato XML que se muestra a continuación. El archivo debe contener una entrada del tipo "< *reuse_case.../* >" para cada par de códigos fuentes considerados sospechosos de ser un caso de reutilización. La figura 6.7 muestra un ejemplo de la estructura del archivo XML.

```
<document>
<reuse_case source_codeC="X1" source_codeJ="Y1" />
<reuse_case source_codeC="X2" source_codeJ="Y2" />
...
</document>
```

Figura 6.7: Ejemplo del archivo XML para la entrega de los resultados.

Para evaluar la eficacia de los sistemas se utilizan las medidas Precisión, Cobertura y la medida F_1. Los resultados han sido ordenados utilizando la medida F_1 para favorecer aquellos sistemas que son capaces de obtener valores "altos" y balanceados tanto en Precisión y Cobertura.

6.3.4 Sistemas participantes

En total cinco equipos participantes enviaron 12 ficheros de resultados. En concreto, el departamento de informática de la Universidad de Gujarat, India (`CLSCR`), el SkyLine LLC de la Universidad del Estado de Zhytomyr State, Ucrania (`Palkovskii`), el instituto tecnológico PES, India (`PES_BSec`), la Universidad Autónoma del Estado de México (`UAEM`) y la Universidad Autónoma Metropolitana - Unidad Cuajimalpa (`UAM-C`). El equipo `CLSCR` solo realizó un envío, el equipo `Palkovskii` realizó dos envíos mientras que el resto de equipos presentaron tres.

`CLSCR` (Shah et al., 2014) principalmente utilizaron dos componentes: *(i)* un compilador para analizar y "traducir" a un lenguaje intermedio específico definido por los autores del trabajo; y *(ii)* posteriormente se calcula la similitud entre las representaciones de un par de códigos fuente utilizando la herramienta Sherlock[9] ya descrita en el apartado 2.2.1. Esta herramienta genera huellas digitales de fragmentos de texto y mediante la comparación de estas huellas es capaz de encontrar fragmentos de texto iguales.

`Palkovskii` (Palkovskii, 2014) utilizó inicialmente un tokenizador de secuencias estándar para dividir el código fuente en n-gramas de palabras de tamaño 5, tratando el mismo código fuente como si de un texto se tratase. También se aplica eliminación de palabras frecuentes y diferentes estructuras de n-gramas, n-gramas estructurales, n-gramas con saltos de palabras (en inglés *skip-words n-grams*), n-gramas basados en entidades (en inglés *NER-based n-grams*). A continuación, se aplica el algoritmo de agrupamiento DB-scan (Ester et al., 1996) para encontrar coincidencias, y diferentes post-procesamientos heurísticos incluyendo combinaciones y saltos. Además también emplea la técnica de la ventana deslizante con las frecuencias estimadas utilizando tf-idf para detectar segmentos altamente ofuscados.

`PES_BSec` (Heblikar et al., 2014) utilizaron un modelo dividido en cuatro pasos: *(i)* preproceso: todos los caracteres se convierten a minúsculas, los espacios en blanco son eliminados y los caracteres acentuados sustituidos por el caracter sin acentuar. La salida de este paso es una secuencia de tokens para cada código fuente; *(ii)* ponderación de la frecuencia: se crean vectores *tf-idf* a partir de los códigos fuente tokenizados; *(iii)* estimación del umbral de similitud: se establece un umbral de similitud considerando el promedio de la similitud del coseno entre los códigos fuente del corpus de entrenamiento; y *(iv)* decisión: se considera que un par de códigos fuente es un caso translingüe de reutilización si su similitud del coseno es mayor que el umbral establecido en el paso *(iii)* . El primer sistema consideró los cuatro pasos anteriormente descritos. En el segundo sistema se elimina el conjunto de los n tokens más frecuentes en el corpus de entrenamiento C y Java tras el paso de preproceso. En el tercer sistema se reemplazan instrucciones C y Java similares por códigos de operación. Por ejemplo, *println* y *printf* por *op1* o *argv* y *argValue* por *op2*. Estos reemplazos se realizaron manualmente tras seleccionar los n más frecuentes del corpus de entrenamiento y posteriormente asignar un código de operación.

`UAEM` (García-Hernández et al., 2015) emplearon la misma aproximación que en la edición monolingüe con ligeras modificaciones. En concreto utilizaron un modelo para detección de reutilización dividido en cuatro fases. En la primera fase solo se tienen en cuenta elementos léxicos (como , , (,), +, *, ; etc.) y si hay más de un espacio en blanco contiguo, se sustituye por un único espacio. Además, añadieron un preproceso léxico que traduce entre lenguajes de programación ciertas

[9] `http://sydney.edu.au/engineering/it/~scilect/sherlock/`

instrucciones, por ejemplo `System.out.print` por `print`. En la segunda fase se obtiene un valor de similitud entre cada par de códigos fuente. Este valor de similitud consiste en la suma de las longitudes de las subcadenas no contiguas más largas, siendo esta suma normalizada respecto a la longitud del código fuente más largo (García-Hernández et al., 2006). En la tercera fase se obtienen un conjunto de parámetros que permite identificar los casos de reutilización posteriormente a partir de los valores de similitud obtenidos en la segunda fase. Los parámetros obtenidos son: *(i)* el valor de *distancia* (1-similitud); *(ii)* el *ranking* ordenado de las distancias; *(iii)* la *diferencia* entre el siguiente en el ranking (solo se calcula para los diez pares más cercanos); y *(iv)* utilizando la diferencia máxima entre los diez pares más cercanos, se etiquetan los pares de códigos fuente según si están *antes* o *después* de esta máxima diferencia. La decisión de que un par de códigos fuente se considere como un caso de reutilización, se toma en la cuarta fase. Si existe evidencia de reutilización en ambos sentidos, es decir, si tanto $X \rightarrow Y$ como $Y \rightarrow X$ son pares sospechosos. Un caso de reutilización existe si la *distancia* es menor que 0,45 y la *diferencia* mayor que 0,14, pero también es importante si se cumple una de las dos condiciones siguientes: *(i)* el par de códigos fuente debe situarse en la primera o segunda posición en el *ranking* y, *(ii)* debe tener la etiqueta de *antes*. El primer sistema consideró las condiciones anteriores. Sin embargo, en algunos casos la evidencia en un sentido es mucho mayor que en el otro, siendo estos dos pares un caso de reutilización. En el segundo sistema si no existe una evidencia de reutilización elevada en un sentido, el par puede considerarse reutilización si un código está situado primero en el *ranking* y la *diferencia* entre ambos códigos en *distancia* es mayor que 0,1. En el tercer sistema se propuso una solución intermedia entre los dos primeros envíos, es decir, consideraron un valor de *distancia* mayor que 0,05.

`UAM-C` (Ramírez-de-la-Cruz et al., 2014a) el método presentado usa cinco características de alto nivel: *(i)* similitud léxica: trigramas de caracteres sin considerar las palabras reservadas; *(ii)* similitud estilística: considerando un conjunto de once características como el número de líneas del código fuente, número de espacios en blanco, número de tabulaciones, número de líneas vacías, número de letras minúsculas, etc.; *(iii)* similitud en los comentarios: similitud del coseno entre los textos dentro de las secciones de comentarios; *(iv)* similitud de los identificadores: similitud entre los identificadores ya sean nombres de variables, funciones o parámetros; y *(v)* similitud estructural: considerando un conjunto de nueve características tales como el número de operadores relacionales, el número de asignaciones, número de llamadas a función, número de bucles, número de sentencias de retorno, etc. En consecuencia, cada par de códigos fuente se representa utilizando estas cinco características, y posteriormente se clasifican como reutilización o no reutilización mediante un clasificador *Random Forest*. El primer sistema se entrenó considerando los pares de códigos fuente reutilizados de SOCO 2014, en concreto la partición C de entrenamiento, utilizando la representación descrita anteriormente. El segundo sistema se entrenó utilizando los códigos fuente reutilizados de la partición de entrenamiento de CL-SOCO, la edición de 2015 también utilizando la representación anterior. En el tercer sistema, solo la similitud léxica fue tomada en cuenta, estableciendo manualmente el umbral de similitud en el 20 %. Es decir, cada par de códigos fuente en la partición de test con similitud léxica de 20 % o superior se considera como un caso de reutilización.

6.3.5 Resultados

En la tabla 6.10 se muestran los resultados obtenidos por los equipos participantes en términos de la medida F_1, la Precisión (P) y la Cobertura (R). Como se ha comentado anteriormente, los resultados a destacar son los obtenidos con la medida F_1, dado que se prefieren sistemas capaces de obtener valores equilibrados entre Precisión y Cobertura. También se considera importante mostrar resultados por tipo de reutilización detectada. En la tabla 6.11 se muestran los resultados de la *reutilización traducida*, mientras que en la tabla 6.12 se muestran los de *reutilización propagada*.

Los mejores resultados en CL-SOCO de acuerdo al valor de la medida F_1 los obtuvo el equipo `UAM-C`. Sin embargo, un análisis de la varianza (ANOVA[10]) muestra que no existe diferencias significativas entre el primer sistema del equipo `UAM-C` y los tres envíos siguientes (primer sistema del equipo `Palkovskii`, el segundo sistema del equipo `PES_BSec` y el primer sistema del equipo `UAEM`). De hecho, la diferencia entre el primer y cuarto clasificado tan solo es de 0,033 en valor de F_1. Los resultados globales de la competición se muestran en la tabla 6.10. En general, todos los envíos obtuvieron un buen rendimiento con valores de F_1 mayores que 0,6. La mayoría de envíos obtuvieron mejores valores de Precisión que de Cobertura a excepción de `UAM-C` en los envíos 2 y 3. En estos dos envíos, su aproximación recuperó más pares de códigos fuente pero con un alto impacto en la Precisión. Este fenómeno ya se encontró en el equipo `UAM-C` en la edición monolingüe de esta competición descrita en el apartado 6.1.

Equipo	Sistema	Medidas de evaluación		
		F_1	*Precisión*	*Cobertura*
UAEM	1	0,739	0,975	0,595
	2	0,709	1,000	0,550
	3	0,703	1,000	0,542
UAM-C	1	**0,772**	0,988	0,634
	2	0,687	0,620	0,771
	3	0,655	0,496	0,962
Palkovskii	1	0,752	1,000	0,603
	2	0,724	0,962	0,580
PES_BSec	1	0,683	1,000	0,519
	2	0,740	1,000	0,588
	3	0,697	1,000	0,534
CLSCR	1	0,611	0,952	0,450

Tabla 6.10: Evaluación global de los resultados de la tarea CL-SOCO.

Como la partición de evaluación contiene distintos tipos de reutilización translingüe, *traducida* y *propagada*, es importante analizar las detecciones por separado. Los resultados de detección de

[10] `https://en.wikipedia.org/wiki/Analysis_of_variance`

reutilización traducida se resumen en la tabla 6.11. Este tipo de reutilización se generó tomando cada código fuente de la colección escrita en C y automáticamente traducido a Java. Además, puede incluir algunos cambios respecto a la traducción exacta del código fuente. Por ejemplo, si una función de cierta librería es desconocida para el traductor, este trata dicha función como desconocida y crea una nueva función con el mismo nombre y argumentos pero con el cuerpo vacío para ser completado por el programador. Este proceso de refactorización hace que este escenario sea un poco más complicado que la simple copia literal entre lenguajes de programación. Los sistemas de los equipos `Palkovskii` y `UAEM` muestran un alto rendimiento en este escenario. También los equipos `PES_BSec` y `CLSCR` obtuvieron valores de F_1 superiores a 0,8. Considerando el valor de Cobertura, la mayoría de los sistemas fueron capaces de recuperar un alto porcentaje de casos de *reutilización traducida* (10 de 12 obtuvieron más de 0,8 de valor de Cobertura). El valor de Precisión no es tan importante como la Cobertura en esta tabla porque estamos considerando los 52 casos de *reutilización propagada* como casos no reutilizados en estos resultados. Por lo tanto, solo proporciona una referencia de la cantidad de casos de *reutilización traducida* recuperada entre todos los casos reportados. Por ejemplo, los resultados del primer sistema del equipo `Palkovskii` muestra que prácticamente todos sus casos recuperados son pares de códigos fuente de *reutilización traducida*, mientras que el valor del sistema 3 del equipo `UAM-C` muestra que el 31,1 % de los casos reportados corresponden a *reutilización traducida* siendo el resto de casos reportados o bien, de *reutilización propagada* o de falsos positivos.

Equipo	Sistema	Medidas de evaluación		
		F_1	*Precisión*	*Cobertura*
UAEM	1	0,893	0,887	0,899
	2	0,927	0,972	0,886
	3	0,933	0,986	0,886
UAM-C	1	0,736	0,714	0,759
	2	0,628	0,466	0,962
	3	0,474	0,311	1,000
Palkovskii	1	**0,975**	0,975	0,975
	2	0,924	0,924	0,924
PES_BSec	1	0,871	0,941	0,810
	2	0,910	0,922	0,899
	3	0,859	0,914	0,810
CLSCR	1	0,823	0,935	0,734

Tabla 6.11: Resultados de los envíos de los participantes *solo* considerando la *reutilización traducida*.

El segundo caso contemplado es el de *reutilización propagada*. Este tipo de reutilización consiste en casos de reutilización monolingüe que han sido traducidos a otro lenguaje de programación. Este supone un escenario más desafiante que el de *reutilización traducida* porque tiene en cuenta modificaciones realizadas al código fuente a nivel monolingüe. Como se puede observar en la tabla 6.12, la

mayoría de envíos obtienen resultados pobres si consideramos el valor de Cobertura. Solo el tercer sistema del equipo `UAM-C` fue capaz de recuperar más de la mitad de casos que existen en este tipo de reutilización translingüe (0,904 de valor de Cobertura). Sin embargo, este sistema ha considerado un alto número de pares como reutilizados siendo su Cobertura la más alta frente a su Precisión, el segundo valor más bajo como se ve en sus resultados globales de la tabla 6.10. Así como ocurre en el otro tipo de reutilización translingüe, los valores de Precisión no son tan importantes como la Cobertura al no considerar todos los casos de reutilización translingüe, *reutilización traducida* y *reutilización propagada*, en estos resultados. Los valores de Precisión tan solo nos indican que los casos reportados de *reutilización propagada* suponen una minoría del total de casos reportados en todos los envíos.

Equipo	Sistema	Medidas de evaluación		
		F_1	*Precisión*	*Cobertura*
UAEM	1	0,106	0,087	0,135
	2	0,032	0,028	0,038
	3	0,016	0,014	0,019
UAM-C	1	**0,338**	0,274	0,442
	2	0,233	0,153	0,481
	3	0,307	0,185	0,904
Palkovskii	1	0,031	0,025	0,038
	2	0,046	0,038	0,058
PES_BSec	1	0,067	0,059	0,077
	2	0,093	0,078	0,115
	3	0,098	0,086	0,115
CLSCR	1	0,018	0,016	0,019

Tabla 6.12: Resultados de los envíos de los participantes *solo* considerando la *reutilización propagada*.

La tarea CL-SOCO ha proporcionado una tarea que supone particularmente un reto para los equipos participantes. La tarea está enfocada en la recuperación de casos translingües de pares de códigos fuente reutilizados a partir de una colección de programas. Al mismo tiempo, CL-SOCO también ofrece un marco de evaluación donde todos los participantes pueden comparar sus resultados bajo las mismas condiciones y el mismo corpus.

En general, se aplicaron aproximaciones de distinta naturaleza para resolver el problema de detección de reutilización translingüe. Las aproximaciones propuestas varían desde las basadas en el emparejamiento de cadenas hasta las basadas en compiladores. Además, el hecho de comparar todas estas aproximaciones bajo las mismas condiciones utilizando las mismas colecciones y medidas en todas ellas, hace posible realizar una correcta comparación de sus resultados. En consecuencia, la aproximación con mejor desempeño fue la combinación de características léxicas y estructurales (Ramírez-de-la-Cruz et al., 2014a). El equipo `UAM-C` ha mostrado un rendimiento más robusto en ambos escenarios que se ve reflejado en los resultados globales. Esta aproximación no obtiene

los mejores resultados al detectar *reutilización traducida*, pero si unos resultados balanceados que sumados al moderado valor de Cobertura en el escenario de *reutilización propagada* causa que obtenga los mejores resultados. Es en el tipo de *reutilización propagada* donde se deben centrar los esfuerzos futuros para mejorar en trabajos futuros.

Finalmente, es importante destacar que tanto el corpus de entrenamiento y el de evaluación representan unos recursos valiosos para ampliar trabajos futuros en el área de la detección translingüe de reutilización en códigos fuente.

6.4 Comparación con la propuesta translingüe

En este apartado se describe cómo se han preparado los modelos propuestos en esta tesis que obtuvieron los mejores resultados a nivel translingüe con el fin de entrenar y evaluar en las mismas condiciones que los sistemas participantes de la tarea CL-SOCO. En el primer apartado se detalla cómo se entrenaron los modelos SoCo-NG y SoCo-LSA a nivel translingüe, mientras que en el segundo ambos se comparan con los sistemas que participaron en la competición.

6.4.1 Entrenamiento y ajuste de los modelos para CL-SOCO

A continuación, se describe la comparación cross-corpus de los modelos SoCo-LSA y SoCo-NG entrenándose con el corpus Rosettacode.org y posteriormente evaluándose en el corpus A&T++. Este experimento cross-corpus se realiza a raíz de la competición internacional CL-SOCO de detección de reutilización translingüe en códigos fuente descrita en el apartado anterior. En esta competición se ofrecieron estos mismos corpus para entrenamiento y evaluación respectivamente.

Durante la experimentación translingüe (también monolingüe) de esta tesis se ha comprobado como los modelos SoCo-NG y SoCo-LSA obtenían los mejores resultados en detección de reutilización de códigos fuente. Para poder realizar una comparación de los modelos propuestos en esta tesis de investigación con los modelos de los participantes de la competición en este apartado se va detallar la fase de experimentación de los modelos SoCo-NG y SoCo-LSA, y posteriormente, se mostrarán los resultados obtenidos. Para la fase de entrenamiento se ha utilizado del corpus Rosettacode.org descrito en el apartado 3.2.2, en concreto los códigos fuente del par C-Java.

Para entrenar el modelo SoCo-NG como si de un modelo participante se tratase, es necesario establecer un criterio que separe los pares de códigos fuente entre reutilizados y no reutilizados. Dado que el modelo ofrece un valor numérico de similitud, se va a establecer un umbral empírico a través de una validación cruzada de diez particiones. En cada partición se establecerá un umbral donde el valor de la medida F_1 se maximice. En la figura 6.8 se muestra el valor de Precisión, Cobertura y F_1 dependiendo del valor de umbral establecido. Finalmente, se establecerá como umbral definitivo la media de los umbrales establecidos en las diez particiones.

El entrenamiento del modelo SoCo-LSA también se ha realizado mediante validación cruzada de diez particiones. En este modelo se han utilizado los pares de códigos fuente de nueve particiones para ser representados en el espacio vectorial reducido y se ha calculado la similitud entre los pares de la partición de validación. Del mismo modo que con el modelo SoCo-NG, en cada validación se ha estimado un umbral de similitud para separar los reutilizados y no reutilizados, siendo la media

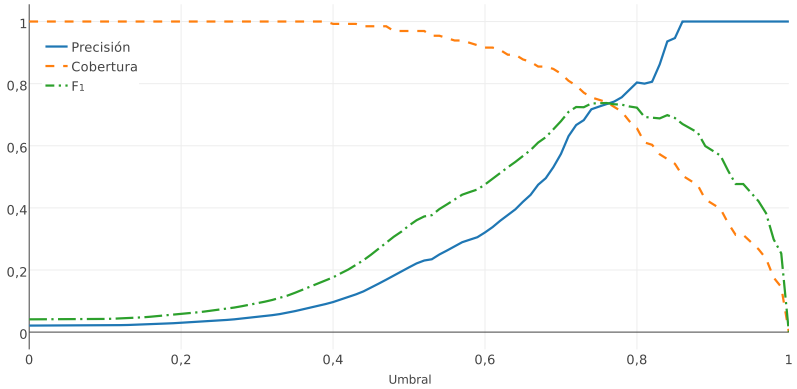


Figura 6.8: Curvas de Precisión, Cobertura y F_1 para el modelo SoCo-NG en la fase de entrenamiento.

de los umbrales, el umbral definitivo que se utilizará para la comparación con los modelos de la competición. En la figura 6.9 se muestra el valor de Precisión, Cobertura y F_1 según el valor de umbral establecido.

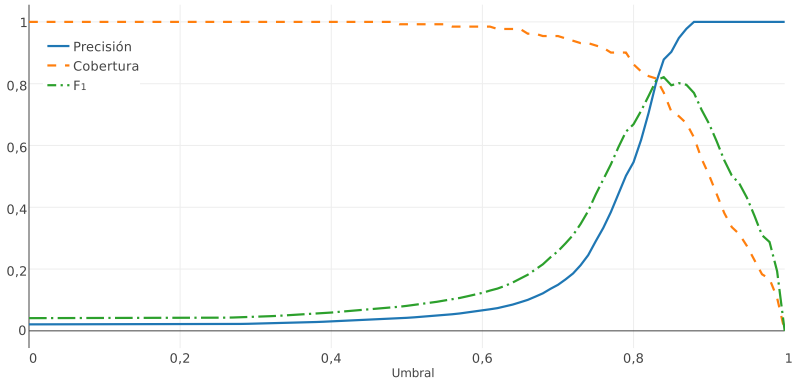


Figura 6.9: Curvas de Precisión, Cobertura y F_1 para el modelo SoCo-LSA en la fase de entrenamiento.

Durante la fase de entrenamiento se ha establecido como el mejor umbral para el modelo SoCo-NG entre el par de lenguajes C-Java de un valor 0,76 mientras que para el modelo SoCo-LSA un umbral de valor 0,84.

6.4.2 Comparación de los modelos propuestos con la competición

En este apartado se pretende realizar una comparación de los modelos propuestos en esta tesis con los de la competición CL-SOCO. Esta comparación se realiza empleando las mismas medidas que se utilizaron en la competición: la Precisión, la Cobertura y F_1. Para realizar esta comparación se han considerado los dos modelos de mejores prestaciones durante la experimentación realizada, SoCo-NG y SoCo-LSA. Tanto la competición como los experimentos sobre los modelos SoCo-NG y SoCo-LSA se centran en maximizar el valor de F_1. Ambos modelos están diseñados para estimar la similitud entre todos los pares de códigos fuente de una colección, mientras que en la competición CL-SOCO se requiere determinar si un par de códigos fuente es considerado como reutilizado o no.

Para adaptar estos dos modelos se ha utilizado el corpus de entrenamiento (véase el apartado 6.3.2). En el caso del modelo SoCo-NG, se considera el código fuente al completo junto con la utilización de trigramas de caracteres y tf como método de pesado. Además, se ha realizado una validación cruzada para establecer un umbral de similitud que maximice el valor de F_1. El umbral para el modelo SoCo-NG queda establecido en el valor 0,76.

En el modelo SoCo-LSA, al igual que en el modelo SoCo-NG, se utiliza la configuración estimada como óptima durante esta tesis. Esta consiste en dividir cada código fuente en trigramas de caracteres, pesar estos trigramas con tf y proyectar cada código fuente en un espacio vectorial reducido de 256 dimensiones. También se ha utilizado validación cruzada para estimar el umbral de similitud maximizando el valor de F_1. El umbral para el modelo SoCo-LSA queda establecido en el valor 0,84.

Sistema	F_1	Precisión	Cobertura
SoCo-LSA	**0,821**	0,878	0,771
UAM-C #1	0,772	0,988	0,634
Palkovskii #1	0,752	1,000	0,603
PES_BSec #2	0,740	1,000	0,588
UAEM #1	0,739	0,975	0,595
SoCo-NG	0,738	0,735	0,740
Palkovskii #2	0,724	0,962	0,580
UAEM #2	0,709	1,000	0,550
UAEM #3	0,703	1,000	0,542
PES_BSec #3	0,697	1,000	0,534
UAM-C #2	0,687	0,620	0,771
PES_BSec #1	0,683	1,000	0,519
UAM-C #3	0,655	0,496	0,962
CLSCR #1	0,611	0,952	0,450

Tabla 6.13: Resultados globales de la competición CL-SOCO junto con los modelos SoCo-NG y SoCo-LSA. El ranking se encuentra ordenado según el valor de F_1.

En la tabla 6.13 se muestran los resultados obtenidos sobre el corpus de evaluación de la competición CL-SOCO de los modelos SoCo-NG y SoCo-LSA comparados con el resto de aproximaciones de la competición ordenados según el valor de F_1. El modelo SoCo-LSA se sitúa con un valor de

F_1 de 0,821 por encima de las aproximaciones de la competición superando por 0,049 a la mejor aproximación de la competición. Por otra parte, el modelo SoCo-NG obtiene buenos resultados consiguiendo un valor de F_1 de 0,738 viéndose superado ligeramente por cuatro aproximaciones de la competición además del modelo SoCo-LSA. Es importante destacar que ambos modelos han obtenido unos valores de Precisión y Cobertura equilibrados. Esto indica que los modelos han sido capaces de detectar con precisión los pares reutilizados y de obtener la mayor cantidad de estos.

Reutilización traducida				*Reutilización propagada*			
Sistema	F_1	Precisión	Cobertura	Sistema	F_1	Precisión	Cobertura
Palkovskii #1	0,975	0,975	0,975	SoCo-LSA	0,462	0,346	0,692
UAEM #3	0,933	0,986	0,886	SoCo-NG	0,375	0,278	0,577
UAEM #2	0,927	0,972	0,886	UAM-C #1	0,338	0,274	0,442
Palkovskii #2	0,924	0,924	0,924	UAM-C #3	0,307	0,185	0,904
PES_BSec #2	0,910	0,922	0,899	UAM-C #2	0,233	0,153	0,481
UAEM #1	0,893	0,887	0,899	UAEM #1	0,106	0,087	0,135
PES_BSec #1	0,871	0,941	0,810	PES_BSec #3	0,098	0,086	0,115
PES_BSec #3	0,859	0,914	0,810	PES_BSec #2	0,093	0,078	0,115
CLSCR #1	0,823	0,935	0,734	PES_BSec #1	0,067	0,059	0,077
UAM-C #1	0,736	0,714	0,759	Palkovskii #2	0,046	0,038	0,058
SoCo-LSA	0,734	0,663	0,823	UAEM #2	0,032	0,028	0,038
SoCo-NG	0,657	0,534	0,848	Palkovskii #1	0,031	0,025	0,038
UAM-C #2	0,628	0,466	0,962	CLSCR #1	0,018	0,016	0,019
UAM-C #3	0,474	0,311	1,000	UAEM #3	0,016	0,014	0,019

Tabla 6.14: Resultados de los sistemas participantes junto con los modelos SoCo-NG y SoCo-LSA considerando la *reutilización traducida* y la *reutilización propagada*. Los resultados están ordenados según valor de F_1.

Así como se realizó en la competición, se ha analizado la eficacia de los modelos según los tipos de reutilización encontrados en este corpus, *reutilización traducida* y *reutilización propagada*. En la tabla 6.14 se muestran los resultados para ambos tipos de reutilización juntos a los obtenidos por los equipos participantes en CL-SOCO. Así como ya se explicó en los resultados de la competición (véase el apartado 6.3.5), la Cobertura ofrece más información acerca del comportamiento del modelo según el tipo de reutilización, mientras que la Precisión muestra el porcentaje que representa el tipo de reutilización entre los pares de códigos fuente reportados. Por ello, se enfoca el análisis de los tipos de reutilización más en la Cobertura que en Precisión y F_1.

Respecto a la *reutilización traducida*, ambos modelos se encuentran en una posición relativamente baja en el ranking pero siendo pequeñas las diferencias en términos de Cobertura respecto al promedio de todos los modelos (el valor promedio de Cobertura es 0,872). En concreto, esta diferencia es de 0,049 y 0,024 para los modelos SoCo-LSA y SoCo-NG respectivamente.

Por otro lado, los modelos SoCo-LSA y SoCo-NG obtienen las mejores prestaciones detectando *reutilización propagada*. Si comparamos la Cobertura de ambos modelos con el promedio de todas las aproximaciones (incluidos estos dos modelos), se encuentra una diferencia de 0,427 y 0,312, siendo el promedio de la Cobertura 0,265. Estos resultados junto con los obtenidos por el equipo `UAM-C` demuestran que las aproximaciones que utilizan n-gramas de caracteres para representar el código (o parte del código) son más eficaces para detectar *reutilización propagada*.

Debido a la poca diferencia entre las aproximaciones de la competición, en esta se realizó un estudio para comprobar si existen o no diferencias significativas entre los modelos. Así pues, se ha realizado conjuntamente el mismo estudio incluyendo los modelos propuestos en esta tesis, SoCo-LSA y SoCo-NG. Mediante un análisis de la varianza de los modelos (ANOVA) se encontró que existen diferencias entre los modelos pero sin especificar cuáles. Por ello, se ha procedido a calcular qué modelos son significativamente distintos mediante las diferencias mínimas significativas (LSD, en inglés *Least Significant Difference*) de Fisher (Fisher et al., 1960). En concreto, las diferencias diferencias mínimas significativas LSD se consideran ciertas con un 95 % de confianza. En la figura 6.10 se muestra para cada modelo su valor medio con los intervalos marcados por la diferencia LSD. Se puede observar que el modelo SoCo-LSA es estadísticamente superior respecto al resto de modelos, situando su intervalo al completo por encima de 0,81. Por otra parte, el intervalo mínimo de significancia del modelo SoCo-NG intersecta con los modelos que obtenían un valor de F_1 ligeramente superior. Esto implica que no existe diferencia significativa respecto al primer sistema del equipo `UAM-C`, los dos sistemas del equipo `Palkovskii`, el segundo sistema del equipo `PES_BSec` y el primer sistema de `UAEM`. En la tabla 6.15 se muestran los valores de las medias, los límites de los intervalos LSD y la distribución de grupos formada a partir de estos intervalos.

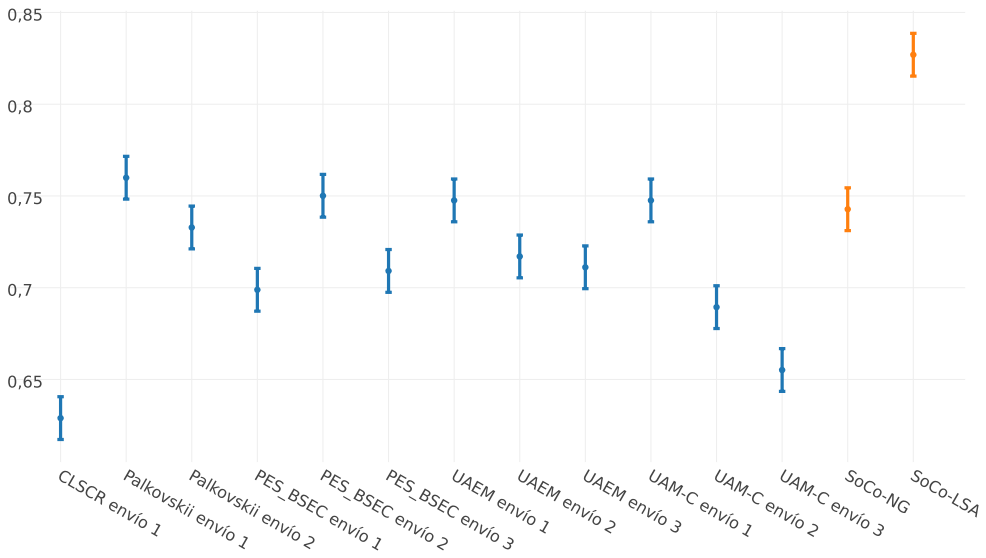


Figura 6.10: Estudio de la significancia estadística de los modelos de la competición CL-SOCO con los modelos SoCo-NG y SoCo-LSA. Se muestran las medias y los intervalos de mínima diferencia significativa de Fisher con una confianza del 95 %.

Sistema	Media	Error estadístico	Límite superior	Límite inferior	Grupos homogéneos
SoCo-LSA	**0.82697**	0,01166	0,83863	0,81531	X
Palkovskii #1	0,75994	0,01166	0,77160	0,74828	XX
PES_BSEC #2	0,75010	0,01166	0,76176	0,73843	XXX
UAEM #1	0,74758	0,01166	0,75924	0,73592	XXX
UAM-C #1	0,74758	0,01166	0,75924	0,73592	XXX
SoCo-NG	0,74278	0,01166	0,75444	0,73112	XXX
Palkovskii #2	0,73282	0,01166	0,74448	0,72116	XXXX
UAEM #2	0,71704	0,01166	0,72870	0,70538	XXXX
UAEM #3	0,71113	0,01166	0,72279	0,69947	XXXXX
PES_BSEC #3	0,70915	0,01166	0,72081	0,69749	XXXX
PES_BSEC #1	0,69887	0,01166	0,71053	0,68720	XXXX
UAM-C #2	0,68941	0,01166	0,70107	0,67775	XXX
UAM-C #3	0,65515	0,01166	0,66681	0,64349	X
CLSCR #1	0,62893	0,01166	0,64060	0,61727	X

Tabla 6.15: Resumen del estudio de los intervalos LSD. En la tabla se muestra el valor de la media, el error estadístico con un 95 % de confianza según la prueba de Fisher, los límites de los intervalos de confianza y la distribución grupal de estos intervalos.

6.5 Conclusiones

En este capítulo hemos descrito las dos competiciones, SOCO y CL-SOCO, organizadas en el marco esta tesis para la promoción y propuesta de un marco de evaluación disponible y abierto para trabajos de investigación actuales y futuros en detección de reutilización en códigos fuente monolingüe y translingüe. Sirviéndonos de este marco de evaluación, hemos realizado una comparativa de los modelos propuestos que obtuvieron mejores resultados en los experimentos llevados a cabo en esta tesis, SoCo-NG y SoCo-LSA, con los sistemas que se presentaron a la competición. En ambos modelos utilizamos la configuración que hemos estimado como óptima en esta tesis. En el modelo SoCo-NG consiste en dividir cada código fuente en trigramas de caracteres tras un preproceso previo, estimar la frecuencia de los trigramas con tf y finalmente comparar los códigos fuente mediante la medida de similitud del coseno. Mientras que en el modelo SoCo-LSA, dividimos cada código fuente en trigramas de caracteres, pesamos estos trigramas con tf y proyectamos cada código fuente en un espacio vectorial reducido de 256 dimensiones. Tanto a nivel monolingüe como a nivel translingüe el modelo SoCo-LSA ha mostrado los mejores resultados siendo significativa la diferencia con respecto a los resultados obtenidos por los otros modelos. El modelo SoCo-NG ha mostrado un buen rendimiento situándose a la altura de los mejores sistemas presentados en ambas competiciones pero en este caso sin diferencias significativas.

En la competición a nivel monolingüe SOCO hemos comprobado que aquellos casos donde es más difícil la detección de reutilización, es decir los códigos fuente de menor longitud, las aproximaciones basadas en características léxicas, como SoCo-NG y SoCo-LSA, son más eficaces en la detección, mientras que en los códigos de mayor longitud propuestas basadas en la estructura muestran un rendimiento ligeramente mejor. En la competición translingüe CL-SOCO hemos detectado que las aproximaciones léxicas obtuvieron mejor rendimiento en el caso de mayor dificultad, es decir, aquellos que no se tratan solo de una traducción (*reutilización propagada*). Cabe destacar la gran

diferencia de los modelos SoCo-NG y SoCo-LSA en este tipo de reutilización translingüe, viéndose mucho menos afectados a los cambios realizados en el código fuente.

Capítulo 7

Conclusiones y trabajos futuros

El objetivo principal de esta tesis ha sido desarrollar modelos eficaces para la detección automática de reutilización en códigos fuente. Para ello, hemos tenido en cuenta distintos escenarios donde es posible que suceda la reutilización de códigos fuente. La reutilización/colaboración entre compañeros en el ámbito académico es tan importante como la reutilización en recursos externos como Webs, respositorios, etc. Se han realizado experimentos tanto a nivel monolingüe como translingüe para comprobar la validez de los modelos propuestos en situaciones distintas. Así pues, el trabajo de investigación se estructuró de la siguiente manera:

- Recopilación de colecciones que contienen casos de reutilización en códigos fuente tanto a nivel monolingüe como translingüe
- Desarrollo y adaptación de modelos de detección de similitud en textos para detección de reutilización en códigos fuente
- Modificación de estos modelos para escenarios translingües
- Estudio de las principales modificaciones que realizan los programadores para evadir ser detectados en la reutilización tanto monolingüe como translingüe
- Análisis de la detección tanto a nivel académico (colecciones medianas) como a gran escala (colecciones de gran tamaño)
- Comparación del rendimiento de los modelos propuestos con distintos ensambles de estos modelos
- Estudio de la dificultad de detección de reutilización entre los pares de lenguajes de programación más comunes
- Comparación de los modelos en escenarios de recuperación translingüe, códigos fuente que resuelven el mismo problema (códigos fuente comparables) y códigos que son considerados traducciones (códigos fuente paralelos)

- Simulación de un escenario de reutilización translingüe a gran escala
- Contribución a la creación de marcos de evaluación para la detección de reutilización en códigos fuente mediante la organización de tareas, así como, la recopilación y difusión de colecciones
- Comparación de los modelos desarrollados con propuestas del estado de la cuestión tanto a nivel monolingüe como translingüe (comparaciones con los sistemas participantes en SOCO y CL-SOCO)

Las publicaciones generadas en el marco de este trabajo de investigación se incluyen en el apéndice B.

7.1 Aportaciones de la tesis

La principal dificultad para la investigación en detección automática de reutilización en códigos fuente es la escasez de colecciones de códigos fuente que estén etiquetadas con casos de reutilización. Nuestros esfuerzos se enfocaron en la creación de colecciones variadas y que estas estén disponibles para futuros trabajos de investigación.[1] Siguiendo la línea marcada en el estado de la cuestión, el ámbito académico es un escenario donde sucede con frecuencia este fenómeno.

- El corpus SPADE (véase apartado 3.1.1) fue recopilado a partir de un trabajo académico que intentaba reproducir el mismo comportamiento de una librería escrita originalmente en Python, a los lenguajes Java y C++. Este corpus se construyó con la intención de que la reutilización fuese lo más fiel posible al original, con la intención de ser integrado dentro del proyecto, siendo este un caso claro de no plagio. Por lo tanto, supone un recurso translingüe ideal para pruebas iniciales de detección translingüe en códigos fuente.
- El corpus ILN (véase apartado 3.1.2) representa un conjunto de tareas de programación dentro la misma asignatura. Los profesores de la asignatura determinaron los casos de reutilización manualmente tras la corrección y revisión de estos. Cada tarea contiene una cantidad de códigos fuente suficiente para analizar las modificaciones más comunes que se realizan para evadir la detección.
- El corpus A&T++ (véase apartado 3.1.3) supone una colección de códigos fuente de tamaño medio que contiene casos de reutilización monolingüe en C y Java, y que hemos enriquecido a posteriori con nuevos casos de reutilización translingüe. Como parte de esta tesis, hemos etiquetado manualmente los corpus monolingües C y Java, además de crear casos simulados de reutilización translingüe de los códigos escritos en lenguaje C al Java. El hecho de que la colección C contiene reutilización monolingüe ha dado lugar a un caso de reutilización translingüe a priori con mayor nivel de dificultad en su detección: la *reutilización propagada*.
- El corpus Google Code Jam (véase apartado 3.2.1) supone el mayor corpus disponible de códigos fuente con casos de reutilización conocidos. Este corpus tiene un alto nivel de similitud con los recopilados del entorno académico dado que un grupo de participantes debe de resolver los mismos problemas y que, además, tienen mayor libertad para reutilizar códigos fuente

[1] Los recursos se encuentran disponibles en: `www.dsic.upv.es/grupos/nle`

entre participantes. El valor añadido que supone este corpus es la enorme cantidad de códigos fuente que contiene la colección y esto supone una gran cantidad de pares de códigos fuente que hay que descartar como casos de reutilización. Sobre esta colección hemos realizado un filtrado de códigos fuente que contienen caracteres no pertenecientes al estándar UNICODE. Además, solo seleccionamos los tres lenguajes más populares, C/C++, Java y Python, por lo que es posible ampliarlo a más lenguajes que se utilizaron en la competición. Por otra parte, hemos enriquecido con 360 pares de códigos fuente etiquetados manualmente como juicios de relevancia, sean casos de reutilización o no, más similares según el modelo SoCo-NG. También se han calculado dos conjuntos de juicios de relevancia automáticamente mediante votación entre las aproximaciones de la competición SOCO.

- El corpus Rosettacode.org (véase apartado 3.2.2) es el corpus más diferente entre los recopilados en este trabajo de investigación. Se ha extraído de un repositorio Web que consiste en un "banco" de códigos fuente con implementaciones de algoritmos conocidos cuyo objetivo es ayudar al aprendizaje de otros lenguajes de programación. Para ello, cada algoritmo disponible en el repositorio se ha desarrollado/traducido a todos los lenguajes de programación posibles por los colaboradores. Hemos recopilado los códigos fuente presentes en los lenguajes más populares, de nuevo C, Java y Python, para crear una colección multilingüe de códigos fuente que resuelven el mismo problema. Tras un etiquetado manual se encontraron casos de reutilización translingüe en este corpus pero no era una cantidad lo suficientemente grande para poder realizar experimentos. Es por esto que hemos enriquecido el corpus con más casos de reutilización translingüe simulándola mediante herramientas de traducción de códigos fuente.

La creación de los recursos descritos anteriormente ha permitido realizar experimentos donde hemos comprobado que la eliminación de partes de código fuente como palabras reservadas, identificadores, comentarios, etc. empeoran el rendimiento de los modelos basados en características léxicas como SoCo-NG (véase apartado 5.2.1 y Flores (2012)). También hemos realizado experimentos sustituyendo los identificadores (véase apartado 5.1.1) por *wildcards*, debido a que estos son los primeros que un programador modifica para evitar ser detectado. Sin embargo, el uso de *wildcards* tan solo favorece la detección de aquellos códigos fuente en que únicamente se han modificado los identificadores, empeorando la detección de otros tipos de modificaciones como cambios de posición de instrucciones, cambios por funciones equivalentes o la combinación de funciones. Por lo tanto, consideramos que es importante tener en cuenta el código fuente completo para realizar las comparaciones.

En la experimentación con la colección Google Code Jam (véase apartado 5.1.2) hemos realizado más de 34 millones de comparaciones entre códigos fuente escritos en los tres lenguajes de programación más populares: C/C++, Java y Python. Dada la gran cantidad de códigos fuente disponible, inicialmente estudiamos las distribuciones de similitud por dificultad de las tareas a resolver y por lenguaje de programación para comprobar si existen diferencias. Es importante destacar que hemos observado una correlación inversa entre la complejidad de resolver un problema y la similitud entre los códigos fuente de los participantes; cuanto menor es la dificultad, mayor es la similitud. Al analizar por cada lenguaje de programación encontramos diferencias: la distribución de similitudes de Java se sitúa en rangos mayores, seguida de C/C++ y finalmente de Python. Esta información es útil para tratar de establecer umbrales de decisión para detectar los casos de reutilización para cada lenguaje de programación. Otro experimento realizado consiste en comprobar si es posible

aplicar y detectar casos de reutilización en un corpus a gran escala con nuestra propuesta. Para ello, comparamos el modelo SoCo-NG con la herramienta JPlag analizando manualmente los 360 pares de códigos fuente con mayor similitud devueltos por ambos. En líneas generales, se observa un desempeño similar por ambos modelos en aquellas tareas que tienen más cantidad de líneas de código fuente, donde se encuentran menos casos reutilización, mientras que en las tareas de menos líneas de código SoCo-NG detecta más casos de reutilización. También hemos realizado un análisis manual sobre los casos de reutilización detectados con el fin de estudiar los distintos tipos de modificaciones realizadas sobre el código fuente. Las modificaciones más comunes en los casos de reutilización analizados son los cambios en los identificadores, comentarios y de formato (sangrías, saltos de línea, etc.).

Sobre el corpus A&T++ hemos realizado diversos experimentos por etapas. En primer lugar, hemos comparado los modelos propuestos SoCo-NG y SoCo-Sliding porque ambos están basados en la misma aproximación que SoCo-Sliding realiza una comparación más exhaustiva a nivel de fragmento de código fuente. Además, en esta comparación también incluimos la herramienta JPlag. Los resultados de ambos modelos muestran un desempeño superior a JPlag, siendo los de SoCo-NG ligeramente superiores a SoCo-Sliding. Debido a que la reutilización de este corpus (y del resto de corpus propuestos) está realizada a nivel de código fuente y que no se dispone de corpus con reutilización etiquetada a nivel de fragmento, hemos decidido no seguir considerando el modelo SoCo-Sliding. Además, las comparaciones exhaustivas que este realiza aumentan significativamente el coste computacional para detectar casos de reutilización en una colección. A continuación, comparamos el modelo SoCo-NG con SoCo-LSA para comprobar si la reducción de dimensiones que el segundo aplica mejora los resultados del primero. Tras comparar con unigramas, bigramas y trigramas de caracteres y de palabras observamos una mejora significativa respecto al modelo SoCo-NG. Finalmente, comparamos todos los modelos propuestos (a excepción de SoCo-Sliding) bajo las mismas condiciones: establecemos un umbral automáticamente mediante un clasificador para distinguir entre los casos reutilizados y los que no están reutilizados según su valor de similitud. También añadimos a la comparación un ensamble de estos modelos utilizando clasificadores conocidos. El ensamble de los modelos, concretamente con el clasificador *Random Forest*, muestra una ligera mejoría en rendimiento respecto al obtenido individualmente por cada modelo.

A nivel translingüe hemos recorrido una línea experimental similar a la monolingüe donde primeramente comprobamos que es importante considerar el código fuente completo para obtener el mejor rendimiento en los pares de lenguajes de programación estudiados: Java–C++, Python–C++ y Java–Python. También hemos comparado los dos modelos SoCo-NG y SoCo-Sliding, los cuales mostraron un rendimiento muy similar para descartar a SoCo-Sliding así como hicimos a nivel monolingüe. Su alto coste computacional y el hecho de que todas las colecciones de códigos fuente se encuentren etiquetadas a nivel de código fuente hacen que no aporte mejoras respecto a SoCo-NG en este trabajo de investigación.

Hemos realizado una experimentación con el corpus Rosettacode.org de gran tamaño comparando los modelos propuestos en dos escenarios de recuperación de información: recuperando códigos fuente que consideramos como traducciones (códigos fuente paralelos) y recuperando códigos fuente que resuelven la misma tarea (códigos fuente comparables). Cabe destacar que realizamos más de cinco millones de comparaciones de códigos fuente por cada modelo de un total de nueve modelos. En general, los modelos muestran un mejor rendimiento en la recuperación de códigos fuente paralelos que en los comparables. Esto es debido a que los códigos fuentes paralelos comparten la misma

estrategia, estructura, puede que coincidan comentarios, identificadores o cadenas de texto mientras que en los códigos fuente comparables tan solo comparten el problema que resuelven.[2] Los modelos SoCo-LSA y la combinación de SoCo-COG y SoCo-NG muestran el mejor rendimiento en todos los pares de lenguajes y ambos escenarios. Es importante destacar el rendimiento casi perfecto en el escenario paralelo del par de lenguajes Java-Python. Esto es debido a que el traductor utilizado, *java2python*, genera traducciones de código fuente prácticamente idénticas en cuanto a sintaxis mientras que el otro traductor utilizado, del lenguaje C al Java, altera ligeramente la estructura y los identificadores.

Otro experimento que hemos realizado con el corpus Rosettacode.org es simular un escenario de reutilización translingüe. Para ello consideramos los pares de códigos fuente paralelos como casos de reutilización y los comparables como no reutilizados. Los pares de códigos fuente comparables, están escritos en diferentes lenguajes de programación y resuelven el mismo problema. Estas dos condiciones son las mismas que se dan en un escenario real. Comparando los modelos en este escenario de reutilización simulado, los modelos SoCo-LSA y SoCo-NG muestran mejores resultados. Este comportamiento sigue el comportamiento observado en las comparaciones a nivel monolingüe. Así como hemos sugerido en la experimentación monolingüe, la distribución de similitud nos puede ayudar a establecer un umbral de decisión para separar los casos reutilizados de los que no lo son. Establecer un umbral de decisión de manera automática nos ha permitido obtener buenas prestaciones de los modelos SoCo-LSA y SoCo-NG, siendo los resultados de SoCo-LSA significativamente mejores que los obtenidos con SoCo-NG.

Como trabajo paralelo a esta investigación, y con la intención de crear un marco de evaluación común para otros trabajos de investigación, hemos organizado dos competiciones de detección de reutilización automática en códigos fuente. En estas competiciones se han ofrecido tanto recursos para entrenar como para evaluar a disposición de los participantes para el desarrollo y mejora de sus sistemas propuestos.

Finalmente, hemos realizado una comparación de los modelos de mejor rendimiento propuestos en esta tesis con los sistemas propuestos en las competiciones aprovechando los marcos de evaluación que se facilitan en SOCO y CL-SOCO. Para ello, hemos entrenado los modelos SoCo-NG y SoCo-LSA utilizando los corpus de entrenamiento de la competición mediante validación cruzada. En el modelo SoCo-LSA, al igual que en el modelo SoCo-NG, se utiliza la configuración estimada como óptima durante esta tesis. Esta consiste en dividir cada código fuente en trigramas de caracteres, pesar estos trigramas con tf y proyectar cada código fuente en un espacio vectorial reducido de 256 dimensiones. Comparando los resultados de SOCO con los que se han obtenido con los dos modelos propuestos, hemos podido apreciar que ambos modelos obtienen mejores resultados en términos de la medida F_1 aunque solo los resultados de SoCo-LSA son significativamente superiores tanto para el lenguaje C como Java. Del mismo modo, en la comparativa translingüe con los sistemas presentados en CL-SOCO, el modelo SoCo-LSA obtiene los mejores resultados siendo estos significativamente mejores, mientras que el modelo SoCo-NG obtiene resultados similares a los mejores de la competición. Analizando en detalle los tipos de reutilización translingüe tal como se hizo en el análisis de la competición, SoCo-NG y SoCo-LSA detectan la mayoría de casos de *reutilización traducida*, y en la *reutilización propagada* superan notablemente a las otras aproximaciones. Estos

[2] Al compartir el problema a resolver, los códigos fuente comparables pueden compartir nombres de identificadores, cadenas o comentarios relacionados con la problemática que permitan a los modelos encontrar similitudes entre los códigos fuente.

resultados demuestran que las aproximaciones que utilizan n-gramas de caracteres para representar el código (o parte del código) son más eficaces para detectar *reutilización propagada*.

7.2 Respuestas de investigación

En este apartado contestamos a cada una de las preguntas de investigación del capítulo 1. En primer lugar respondemos a las cuestión sobre la detección automática de reutilización seguido de cuestiones relativas a la creación de colecciones.

Respuestas sobre detección automática

1. ¿Qué modificaciones realiza un programador para evitar la detección?

   En estudios del estado de la cuestión (véase Faidhi et al. (1987) y Whale (1990)) se destacaban cambios en el estilo del código fuente (sangrías, saltos de línea, tabulaciones, etc.), cambios en los identificadores, comentarios del código fuente y reemplazo por funciones equivalentes. A lo largo de esta tesis de investigación se han observado las modificaciones encontradas por parte de los programadores para que su reutilización no sea detectada a través de distintas colecciones. En colecciones del ámbito académico (corpus ILN y A&T++) encontramos como modificaciones más comunes los cambios de identificadores, cambios en la posición de fragmentos. Por otra parte, en colecciones con grandes cantidades de códigos fuente y no ligadas estrictamente a las aulas es más común la modificaciones de comentarios, cambios en los identificadores o sustitución de instrucciones/sentencias del código por equivalentes. En menor medida también encontramos casos de reutilización donde se alteraba la posición de partes del código fuente y otros donde se combinaban funciones.

2. ¿Se pueden aplicar modelos de detección de reutilización en textos sobre casos de reutilización en códigos fuente?

   Durante la fase de experimentación de esta tesis se han realizado adaptaciones de los modelos de detección de reutilización que demostraron ser eficaces en textos. A nivel monolingüe hemos comparado con la herramienta JPlag mostrando nuestro modelo propuesto, SoCo-NG, un mejor rendimiento además de poderse aplicar en cualquier lenguaje de programación.[3] También se compararon los modelos propuestos con otras aproximaciones presentadas a la competición SOCO: aproximaciones basadas en la estructura interna del código, aproximaciones basadas en características léxicas y estilísticas. En esta ocasión, mientras el modelo SoCo-NG mostró un buen desempeño en la línea de las aproximaciones participantes, el modelo SoCo-LSA que ya mostró un mejor desempeño entre los propuestos en la tesis, también obtuvo mejores prestaciones que los presentados a la competición. A nivel translingüe comparamos ambos modelos propuestos sobre un escenario de recuperación de información de códigos fuente comparables, paralelos y, también, en un escenario simulado de detección de reutilización translingüe obteniendo los mejores resultados los modelos SoCo-LSA y SoCo-NG en este orden.

[3] La herramienta JPlag solo puede aplicarse sobre códigos fuente escritos en lenguajes de la familia C/Java, no pudiéndose aplicar por ejemplo en otros lenguajes populares como Python o Ruby.

3. ¿Qué modelos obtienen mejores resultados para la detección de reutilización?

   En la experimentación/evaluación monolingüe comparamos tanto modelos basados en características léxicas (SoCo-NG, SoCo-LSA, etc.), modelos basados en características estructurales (`DCU` y `UAM-C` en SOCO) o basados en patrones (JPlag o `UAEM` en SOCO). Los modelos basados en características léxicas SoCo-NG y SoCo-LSA han mostrado un mejor desempeño, siendo los resultados de SoCo-LSA significativamente mejores. En el apartado 5.1.2 comprobamos que los modelos basados en características léxicas eran más eficaces en aquellos casos de reutilización que suponen un mayor reto para ser detectados, es decir, los códigos fuente de menor longitud.

   En la experimentación translingüe comparamos los modelos SoCo-LSA y SoCo-NG con los presentados a la competición CL-SOCO entre los cuales se encuentran aproximaciones de distinta naturaleza: basados en la estructura que se genera en tiempo de compilación, basados en la detección de patrones, etc. El modelo SoCo-LSA fue capaz de mejorar significativamente los resultados de los mejores sistemas participantes, mientras que el modelo SoCo-NG obtuvo resultados en la línea de estos. A la vista de los resultados, los modelos propuestos de detección de reutilización en código fuente han mostrado buenos resultados en los escenarios reales/realistas presentados en esta tesis.

4. ¿Cómo podemos abordar la detección translingüe?

   Hasta el inicio del trabajo de esta tesis doctoral pocos trabajos abordaban la detección translingüe de reutilización en códigos fuente. Estos trabajos se basaban en la estructura del código fuente que genera el compilador, o bien comparando lenguajes prácticamente iguales (versión de Java para Symbian y versión Java para Android) o bien empleando lenguajes que un mismo compilador es capaz de manejar (GCC). Estos sistemas son únicamente aplicables entre los lenguajes estudiados por depender estrictamente de las herramientas que manejan un reducido número de lenguajes de programación. En la tesis hemos propuesto modelos que dependen del lenguaje de programación (SoCo-NG, SoCo-Sliding, SoCo-COG y SoCo-WCR) y otros modelos que tan solo necesitan de un corpus de entrenamiento alineado entre los dos lenguajes de programación que se estudian (SoCo-LSA, SoCo-ESA y SoCo-ASA). Tan solo la herramienta DeSoCoRe (véase el apéndice A) requiere de un analizador léxico por cada lenguaje de programación tratado para distinguir las funciones en las que se divide un código fuente. Los modelos propuestos han demostrado su buen rendimiento en la comparación realizada con los participantes de la competición CL-SOCO.

5. ¿Existen distintos tipos de reutilización translingüe?

   En la literatura se menciona la reutilización translingüe como un único tipo de reutilización. Sin embargo, en esta tesis identificamos dos subtipos de reutilización translingüe: *reutilización traducida* y *reutilización propagada*. La *reutilización traducida* corresponde a aquellos casos en los que un (fragmento de) código fuente se ha traducido de un lenguaje de programación a otro distinto ya sea mediante un proceso manual o automático. La *reutilización propagada* consiste en aquellos casos en los que un (fragmento de) código fuente que contenía reutilización a nivel monolingüe es traducido a otro lenguaje de programación. Este segundo tipo de reutilización se considera más cercano a un caso real debido a que un mismo código fuente puede ser traducido de más de una forma por un programador siendo todas igualmente válidas. Si

un par de códigos fuente reutilizados a nivel monolingüe es traducido a otro lenguaje de programación, dispondremos de dos códigos fuente probablemente distintos con el mismo comportamiento.

6. ¿Podemos detectar del mismo modo estos distintos tipos de reutilización translingüe?

   En la comparación del apartado 6.4.2 encontramos que la *reutilización propagada* es más difícil de detectar que la *reutilización traducida* por la reutilización monolingüe adicional que contiene. La mayoría de modelos presentados a la competición CL-SOCO sufren una pérdida grande de rendimiento al considerar este escenario más complejo. La eficacia de modelos propuestos en la tesis con mejor desempeño, SoCo-LSA y SoCo-NG, también disminuye en el escenario de *reutilización propagada* pero en menor medida, obteniendo los mejores resultados en este tipo de reutilización translingüe más difícil de detectar.

Respuestas sobre recursos

Consideramos relevante haber creado y/o colecciones con casos de reutilización en códigos fuente. En el apartado 1.2 identificamos tres grandes dificultades para la detección automática de reutilización en códigos fuente: la falta de colecciones disponibles con casos de reutilización, el tamaño pequeño de las colecciones generalmente extraídas del ámbito académico, la escasez de trabajos en detección de reutilización translingüe de códigos fuente.

La gran mayoría de trabajos de investigación utilizan corpus del ámbito académico donde los profesores encontraron casos de reutilización manualmente. La falta de consentimiento por parte de los estudiantes, especialmente por aquellos que cometieron plagio, hace imposible la difusión de estas colecciones por problemas de privacidad. Por otra parte, se encuentran las colecciones recompiladas a partir de recursos de libre disposición, pero estas recopilaciones no representan situaciones realistas de un escenario de reutilización en códigos fuente. Como parte de este trabajo de investigación se han propuesto varias alternativas como colecciones con reutilización realistas. A nivel monolingüe se proponen los corpus A&T++ con casos reales de reutilización en C y Java, y el corpus Google Code Jam también con casos reales en los lenguajes C/C++, Java y Python. A nivel translingüe se proponen la colección Rosettacode.org con casos comparables y paralelos en los lenguajes C, Java y Python que representan un escenario realista, y la colección translingüe A&T++ con casos reales (*reutilización propagada*) y casos realistas (*reutilización traducida*). Todos los corpus mencionados han formado parte o bien de la fase de entrenamiento o de evaluación de las competiciones SOCO y CL-SOCO descritas en el capítulo anterior.

7. ¿Cómo construir colecciones de códigos fuente para el estudio y desarrollo de herramientas de detección automática de reutilización?

   Para la construcción de colecciones realistas de reutilización en códigos fuente se han considerado las dos condiciones siguientes: que las colecciones se encuentren disponibles para otros estudios/comparaciones de detección de reutilización en códigos fuente y que estas colecciones representen un entorno lo más real posible. Para cumplir la primera condición, hemos explorado repositorios de libre acceso como son Rosettacode.org o la competición Google Code Jam, solicitamos autorización de uso de la parte cedida del corpus A&T++. En la segunda condición, hemos buscado escenarios donde ya se ha dado reutilización entre códigos fuente como es el corpus A&T++ y también hemos buscado escenarios propicios a contener casos

de reutilización como en Google Code Jam donde en la ronda inicial todos los participantes disponen de 25 horas para resolver los problemas desde cualquier lugar del mundo. También se han simulado escenarios de reutilización translingüe empleando herramientas de traducción de códigos fuente. En el corpus Rosettacode.org hemos utilizado el traductor para crear los casos de reutilización paralela en códigos fuente, mientras que en el corpus A&T++ se crearon los casos de *reutilización propagada* y *reutilización traducida*.

8. ¿Qué condiciones debe cumplir una colección de códigos fuente para considerar que contiene casos de reutilización?

   La respuesta a esta pregunta depende de en que contexto se formula, por ejemplo, desde el ámbito de la recuperación de información tan solo es necesario disponer del código/conjunto de códigos que deben encontrarse al realizar una consulta. La colección de códigos fuente entre los que se realiza la búsqueda solo debe cumplir el requisito de ser muestras sin reutilización. Sin embargo, centrándose en la detección de reutilización de códigos fuente, se considera una escenario realista si la colección de códigos fuente en la cual se realiza la búsqueda comparten problemática y por lo tanto un cierto grado de similitud. Por esta razón la búsqueda de colecciones de códigos fuente para la detección de reutilización además de centrarse en entornos propensos a la reutilización también la hemos enfocado en que un colectivo de programadores trate de resolver un mismo problema.

9. ¿Es posible crear colecciones de códigos fuente con casos simulados de reutilización?

   En esta tesis se han descrito diversos tipos de modificaciones del código fuente que puede realizar el programador para evadir su detección que son fácilmente de realizar como de detectar, por ejemplo sustituir/modificar los identificadores del código fuente o cambiar de posición fragmentos de código fuente sin alterar el funcionamiento de este. Sin embargo, hemos recurrido a simular casos de reutilización por no disponer de la cantidad suficiente de casos reales de reutilización translingüe (véase el apartado del corpus Rosettacode.org). En concreto hemos simulado los casos de reutilización translingüe empleando herramientas de traducción de códigos fuente. Para el corpus A&T++ creamos los casos de reutilización entre los lenguajes C y Java dando lugar a los dos tipos de reutilización descritos en la tesis: *reutilización propagada* y *reutilización traducida*, mientras que en el corpus Rosettacode.org creamos los pares de códigos paralelos entre los lenguajes C, Java y Python.

10. ¿Son útiles los casos simulados para la detección de casos reales de reutilización?

    La utilidad de los casos simulados se aprecia en como de probable es que un programador reutilice un código fuente del mismo modo que se ha realizado de forma artificial. En nuestro caso, solo se han simulado casos de reutilización translingüe. Cuando un programador necesita traducir un código fuente de un lenguaje de programación L a otro L', generalmente tiene un cierto grado de conocimiento y manejo de ambos lenguajes. Con este conocimiento es capaz de encontrar equivalencias entre los lenguajes de programación que le permitan reproducir el comportamiento del código fuente escrito en L, en otro código fuente escrito en L'. Los traductores de códigos fuente disponen de mecanismos muy similares a los que utilizaría un programador al traducir código fuente, mediante una serie de patrones o reglas son capaces de traducir un código fuente. Cuando se realiza una traducción de una colección de códigos fuente, el traductor siempre está utilizando el mismo conjunto de reglas, mientras que distintos

programadores pueden emplear distintos mecanismos para obtener traducciones igualmente válidas. Por ello, en esta tesis se le da un valor importante a la *reutilización propagada* porque en parte simula este comportamiento; una reutilización manual a nivel monolingüe más un proceso de traducción automático generará una traducción igualmente válida.

11. ¿Es posible crear colecciones de códigos fuente con casos de reutilización translingüe?

    Durante el etiquetado manual del corpus Rosettacode.org encontramos 62 pares de códigos fuente paralelos, que consideramos como reutilización translingüe, entre los lenguajes C++ y Java. Sin embargo, esta cantidad resultaba insuficiente para realizar una experimentación con entrenamiento y evaluación por lo que se construyó un corpus paralelo mediante herramientas de traducción automática de códigos fuente. Estas herramientas de traducción de códigos fuente están basadas en reglas/patrones que permiten convertir un código escrito en un lenguaje de programación a otro equivalente en un lenguaje de programación distinto del mismo modo que lo realizaría un programador. Además, en el corpus A&T++ llevamos la reutilización translingüe un paso más allá al traducir entre lenguajes de programación pares de códigos fuente que habían sido reutilizados a nivel monolingüe (*reutilización propagada*). Este proceso permite recrear la situación donde un programador conoce más de una forma en la que puede traducir un fragmento de código fuente siendo así un caso de reutilización más realista.

7.3 Lineas de investigación abiertas

Aún siguen abiertas muchas líneas en la investigación de la reutilización de códigos fuente. En este apartado remarcamos las más interesantes según nuestro punto de vista.

Identificación del origen de la reutilización

En este trabajo de investigación nos hemos centrado en detectar casos de reutilización dentro de colecciones de códigos fuente. Sin embargo, en un caso de plagio es tan importante la detección del acto deshonesto cometido como identificar cual es el contenido origen y cual es el que contiene el contenido plagiado, incluso si ha sido una colaboración mutua. En estos casos, el análisis forense y la atribución de autoría son esenciales para encontrar las evidencias pertinentes. Por ejemplo, en un escenario académico el orden de entrega y la fecha de creación de los trabajos de programación pueden indicar en ciertos casos cual es el original y cual el plagiado (Hellyer et al., 2009). Otra forma de abordar el problema es tratando de identificar el estilo de cada programador para determinar si el estilo de un (fragmento de) código fuente corresponde al del autor o, si por lo contrario, ha habido reutilización.

Detección intrínseca de reutilización

En la misma línea que la propuesta anterior se encuentra la detección intrínseca de reutilización en código fuente. Se trata de dado un código fuente escrito por un único programador, encontrar fragmentos donde haya evidencias de que el fragmento haya sido reutilizado identificando cambios en el estilo. Cuando un código fuente ha sido escrito por múltiples programadores la situación es más complicada. Una posible solución es tratar de identificar que fragmentos han sido escrito por cada programador y, dentro de cada fragmento, tratar de encontrar partes que no se correspondan en estilo al del fragmento, es decir, reducir el escenario de múltiples autores al de un único autor.

Recopilación de nuevas colecciones con reutilización en códigos fuente

La falta de colecciones de códigos fuente con reutilización disponibles es una gran barrera para investigar en la reutilización en códigos fuente. A lo largo de la tesis se han recopilado colecciones del ámbito académico, colecciones de gran tamaño a partir de competiciones de programación, colecciones translingües a partir de repositorios Web. También se han creado colecciones simuladas para aumentar el número de casos de reutilización translingüe. Es necesario destacar el proceso posterior de análisis y etiquetado manual de los casos de reutilización sobre estas colecciones para formar los juicios de relevancia con el que poder evaluar los modelos. Aunque las colecciones derivadas de esta tesis son un conjunto valioso para futuras investigaciones en el área, es necesario crear más colecciones (y si es posible) con mayor número de casos de reutilización para formar otros bancos de pruebas externos a este trabajo de investigación. Consideramos que la divulgación y libre disposición de este tipo de recursos para la investigación es esencial para diseñar mejores sistemas para la detección de reutilización en código fuente.

Por otra parte, también consideramos interesante que se creen colecciones (reales o simuladas) de códigos fuente con reutilización a nivel de fragmento. En proyectos académicos, o para resolver competiciones, es común la reutilización a nivel de código fuente, pero en proyectos de gran envergadura[4] la reutilización más común es la de fragmentos de código.

Detección de fragmentos reutilizados

Durante la investigación de esta tesis se ha planteado el modelo SoCo-Sliding que es capaz de analizar los códigos fuente a nivel de fragmento (ventana). Sin embargo, todas las colecciones de códigos fuente recopiladas y analizadas contienen casos de reutilización a nivel de código fuente por lo que la detección resultante del modelo SoCo-Sliding era a nivel de código fuente. Se propone como trabajo futuro que junto a futuras colecciones de códigos fuente etiquetadas con casos de reutilización a nivel de fragmento se realicen detecciones a nivel de fragmento como realiza la herramienta JPlag.[5] Así como ocurre en la competición internacional de detección de plagio en

[4] Se entiende por proyecto de gran envergadura aquellos proyectos software que contienen miles de líneas de código fuente.

[5] JPlag además de proporcionar un valor global de porcentaje de reutilización, también muestra los fragmentos comunes entre cada par de códigos fuente.

textos (Stamatatos et al., 2015), sería posible utilizar medidas de acierto sobre fragmentos como *plagdet*[6] para medir el acierto en detección de fragmentos en códigos fuente.

Detección de casos de plagio

Durante todo el documento de tesis hemos utilizado mayoritariamente el concepto de reutilización en detrimento de plagio por no realizar la comprobación de si se estaba cometiendo una acción deshonesta. El propósito de esta investigación no se focaliza en detectar casos de plagio sino en proporcionar herramientas que sean capaces de detectar la reutilización y por lo tanto ofrecer indicios de posibles casos de plagio. Una posible línea de investigación es la de Identificación de casos de plagio en códigos fuente. Esta línea es muy cercana a la de detección de citas en textos donde se busca y comprueba si un fragmento de texto sospechoso de ser plagio, está citado, y además la cita se corresponde con el documento original.

Recuperación de códigos fuente

En la experimentación se ha observado que los modelos propuestos tienen un buen rendimiento al recuperar códigos fuente traducidos a otro lenguaje de programación y aceptable al recuperar problemas que resuelven el mismo problema. La búsqueda de códigos fuente similares/traducidos (independientemente de que sea a nivel monolingüe o translingüe) entre colecciones de códigos fuente que no mantienen relación o similitud alguna es un campo interesante y poco explorado. La búsqueda de códigos fuente dentro de repositorios de libre distribución y/o dentro de repositorios internos en una empresa puede ser reducida en costes temporales y monetarios con herramientas de este tipo.

Detección de reutilización con ofuscación

En la experimentación realizada en esta tesis no se ha tenido en cuenta el fenómeno de la ofuscación de código fuente. Nos hemos centrado en el proceso de modificación de los códigos fuente que un programador puede realizar por sí mismo sin recurrir a herramientas de ofuscación. En los escenarios donde hemos estudiado el fenómeno de la reutilización en códigos fuente la ofuscación no está permitida. Por ejemplo, en el ámbito académico, el profesor debe comprender el código fuente para evaluarlo, o en una competición de programación, en caso de reclamación el jurado puede analizar el código fuente para comprobar que es genuino.

A pesar de que existen herramientas capaces de alterar el código fuente de tal forma que sea completamente ilegible por una persona, creemos que el proceso realizado por estas herramientas se puede revertir porque están basadas en un conjunto finito de reglas de transformación. Si conseguimos identificar la secuencia de reglas aplicadas sobre un código fuente ofuscado, seremos capaces de conseguir el código fuente original. La Identificación de la secuencia de reglas realizadas en el proceso de ofuscación es una línea interesante de investigación.

[6] *Plagdet* es una medida compuesta por la micro Precision, micro Cobertura y la granularidad. Para mas detalles consultar Potthast et al. (2010b)

Combinación de aproximaciones estructurales con las léxicas

En la competición SOCO hemos observado como la herramientas basadas en características léxicas propuestas en esta tesis eran más efectivas en los códigos fuente de menor longitud, y que por lo tanto, es más difícil identificar los casos de reutilización por la alta similitud entre los casos no reutilizados. Sin embargo, en los problemas que necesitan más líneas de código fuente para resolverlos, se mostró un desempeño ligeramente mejor que las propuestas basadas en la estructura del código fuente. Por lo tanto, se propone emplear soluciones mixtas que ofrezcan un mayor peso a distintas propuestas, léxicas, estructurales, etc., dependiendo de la longitud de los códigos fuente a comparar.

Extensión de la herramienta DeSoCoRe

La herramienta DeSoCoRe actualmente es capaz de analizar y detectar reutilización entre los lenguajes C, Java y Python. Consideramos que esta herramienta puede extenderse a multitud de lenguajes de programación dado que tan solo requiere de un analizador léxico para identificar el alcance de las funciones. Otra posible extensión es la de analizar fragmentos de un determinado tamaño establecido por el usuario. Al analizar fragmentos de código fuente, no se necesita la estructura del código fuente por lo que se podría analizar cualquier par de códigos fuente sin depender del lenguaje de programación en que estén escritos. Como tercera extensión posible se plantea el manejo de colecciones de códigos fuente. Actualmente, DeSoCoRe realiza comparaciones entre dos códigos fuente mientras que lo que se plantea es que sea capaz de comparar todos los códigos fuente de una colección contra el resto de códigos de esta colección independientemente del lenguaje de programación en que estén escritos.

Apéndices

Apéndice A

Herramienta DeSoCoRe

La herramienta DeSoCoRe (Flores et al., 2012) (*Detection of Source Code Re-use*) permite la detección de reutilización entre lenguajes de programación. El principal objetivo de esta herramienta es facilitar la revisión de código fuente con el fin de decidir si existe reutilización. DeSoCoRe es la primera herramienta online capaz de detectar reutilización entre códigos fuente escritos en diferentes lenguajes de programación. Se ha desarrollado una metodología para detectar reutilización entre lenguajes de programación y se muestra su funcionalidad a través de la herramienta DeSoCoRe. Esta herramienta permite comparar códigos fuente escritos en Python, Java y lenguajes de la familia sintáctica de C: C, C++ o C#. La similitud entre códigos fuente se calcula de manera independiente del lenguaje de programación mediante técnicas de PLN. De hecho, los lenguajes de programación tienen cierta similitud con el lenguaje natural: ambos se representan mediante cadenas de símbolos (caracteres, palabras, sentencias, etc.). El objetivo de DeSoCoRe es el de apoyar al revisor en el proceso de detectar funciones dentro de códigos fuente con un grado alto de similitud. Esta herramienta permite visualizar las coincidencias detectadas entre dos códigos fuente. Los códigos fuente se representan como un grafo. El código fuente está representado como un nodo, así como también sus funciones. Cada código fuente está conectado a sus funciones mediante una arista. También existe una arista entre las funciones sospechosas de ser objeto de reutilización. El contenido de las funciones del código fuente se muestra al usuario para su posterior revisión. Con la información proporcionada, el revisor puede decidir si una función de un código ha sido reutilizado.

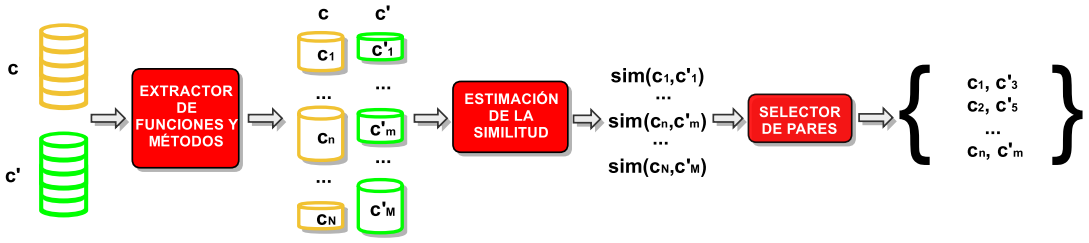


Figura A.1: Arquitectura de la herramienta DeSoCoRe. El código fuente c tiene N funciones, y c' tiene M funciones. Cada función del código c se compara contra todas las funciones de c'.

Como se muestra en la figura A.1, DeSoCoRe consiste en tres módulos generales. El usuario proporciona como entrada un par de códigos fuente (c y c'). El *Extractor de funciones y métodos* es el encargado de dividir los códigos fuente en funciones. Para dividir cada código fuente en funciones se han desarrollado en este trabajo de investigación analizadores sintácticos para Python, C y Java.

El módulo *Estimación de la similitud* es el encargado de comparar las funciones del código fuente c con las de c'. Para realizar la comparación el módulo está dividido en tres submódulos: *(i)* Preproceso: saltos de línea, tabulaciones y espacios se eliminan, así como también los caracteres se convierten a minúsculas; *(ii)* Extracción de características: el fragmento de código fuente se divide en n-gramas y se estima su frecuencia con tf; y *(iii)* Comparación: se estima la similitud entre las funciones de los códigos fuente utilizando la similitud del coseno. La salida de este módulo es un valor de similitud en el rango $[0 - 1]$ para cada par de funciones de dos códigos fuente. Se han realizado diversos experimentos con el fin de encontrar la mejor combinación para detectar reutilización entre códigos. Estos experimentos se inspiran en los niveles de modificaciones propuestos en (Faidhi et al., 1987). Los niveles están descritos según la dificultad por parte del programador para esconder la reutilización del código fuente. Para una descripción detallada de los niveles véase la figura 2.5. Como resultado de esta experimentación se obtuvo la mejor configuración utilizando el código fuente completo y utilizando trigramas de caracteres para medir la frecuencia (véase sección 5.2.1).

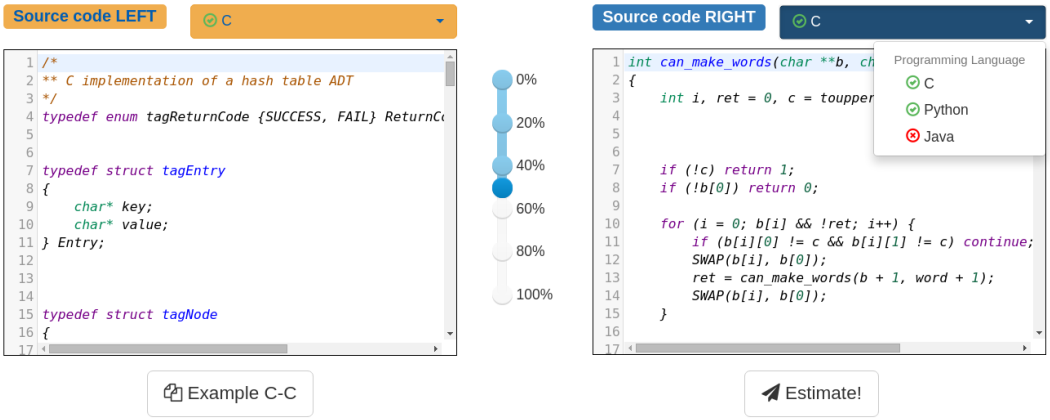
Una vez se han calculado todos los valores de similitud entre los pares de funciones, el *Selector de Pares* decide los pares de funciones candidatos a ser un caso de reutilización en base a un umbral de similitud establecido por el usuario. Este módulo es el encargado de descartar los pares de funciones inferiores a dicho umbral y emparejar aquellas con mayor valor de similitud. Como resultado, DeSoCoRe devuelve los pares de funciones sospechosas de haber sido objeto de reutilización.

La herramienta DeSoCoRe está disponible en forma de aplicación Web.[1] Está dividida en dos interfaces: *(i)* Pantalla de entrada: permite al usuario introducir los dos códigos fuente, elegir sus lenguajes de programación y, adicionalmente, seleccionar un valor de umbral de similitud; y *(ii)* Pantalla de salida: muestra los resultados en dos secciones: *(a)* una representación gráfica de los dos códigos fuente y sus funciones candidatas a ser casos de reutilización; y *(b)* una representación textual de las funciones. Se ha utilizado la librería *vis.js*[2] para dibujar el grafo que representa los códigos fuente y los posibles casos de reutilización entre estos. El grafo dibujado consiste en dos nodos naranja y azul que representan cada código fuente. Sus funciones se representan con nodos de su mismo color y conectados al código fuente mediante ejes. Si alguna función ha sido seleccionada por el sistema como posible caso de reutilización, el nodo que representa la función estará conectado con la función correspondiente del otro código fuente. También se muestra el contenido de las funciones seleccionadas como reutilizadas para facilitar la revisión manual de ambas. La propia herramienta dispone de un ejemplo para familiarizarse con su utilización. La figura A.2 muestra un ejemplo de dos códigos fuente sospechosos en C.

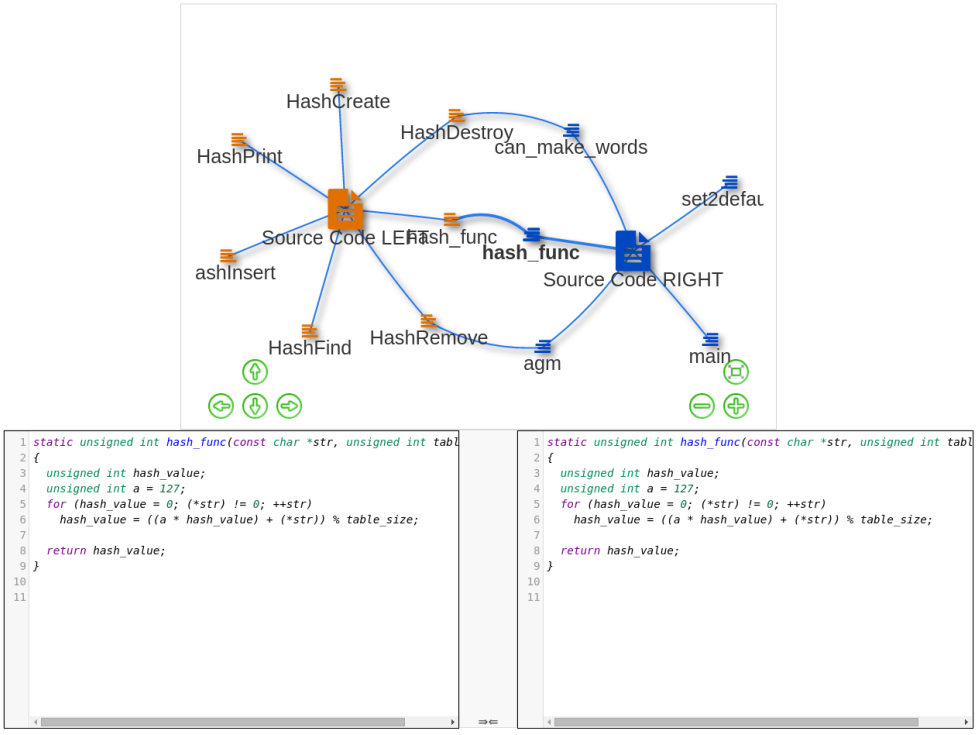
El usuario inicia el cálculo de la similitud pulsando en el botón *Estimate!* mostrado en la figura A.2a. Después del cálculo de la similitud, el resultado se muestra como en la figura A.2b. Para poder explorar la herramienta, se dispone de un ejemplo didáctico pulsando el botón *Example C-C*. En

[1] `http://desocore2.appspot.com/`. Para ver la versión anterior en forma de *applet* visitar: `http://memex2.dsic.upv.es/DeSoCoRe/`

[2] `http://visjs.org/`



(a) Pantalla de entrada: el usuario debe seleccionar cada lenguaje de programación manualmente.



(b) Pantalla de salida: las funciones reutilizadas están conectadas mediante un eje y las funciones se muestran en el área de texto inferior.

Figura A.2: Interfaces de la herramienta DeSoCoRe

la figura A.2 se muestra un caso potencial de reutilización. La función *hash_func* está seleccionada en el código fuente de la derecha, y la función candidata de ser la reutilización de la misma se visualiza en el código de la izquierda también queda seleccionada. En la parte inferior se muestra el código fuente como texto plano de ambas funciones candidatas de reutilización viéndose como son prácticamente idénticas.

Apéndice B

Publicaciones relacionadas

Esta investigación ha generado un total de 13 publicaciones científicas. Por cada publicación mostramos su impacto (en términos de citas)[1] y en qué apartado de la tesis se mencionan.

Publicaciones derivadas de la tesis

1. **Flores, E.**, L. Moreno y P. Rosso (2016). «Detecting Source Code Re-Use with Ensemble Models». En: *Proceedings of the 4th Spanish Conference on Information Retrieval. CERI '16*. ACM, 16:1-16:7. DOI: 10.1145/2934732.2934738.

2. **Flores, E.**, A. Barrón-Cedeño, L. Moreno y P. Rosso (2015b). «Cross-language source code re-use detection using latent semantic analysis». En: *Journal of Universal Computer Science*, 21.13, págs. 1708–1725.

3. **Flores, E.**, A. Barrón-Cedeño, L. Moreno y P. Rosso (2015c). «Uncovering source code reuse in large-scale academic environments». En: *Computer Applications in Engineering Education* 23.3, págs. 383–390. DOI: 10.1002/cae.21608.

4. **Flores, E.**, P. Rosso, L. Moreno y E. Villatoro-Tello (2015d). «On the detection of source code re- use». En: *Proceedings of the Forum for Information Retrieval Evaluation. FIRE 2014*. Bangalore, India: ACM, págs. 21–30. DOI: 10.1145/2824864.2824878.

5. **Flores, E.**, P. Rosso, E. Villatoro-Tello, L. Moreno, R. Alcover y V. Chirivella (2015a). «PAN@FIRE: Overview of CL-SOCO track on the detection of Cross-Language SOurce COde re-use». En: *FIRE 2015 Working Notes. Seventh International Workshop of the Forum for Information Retrieval Evaluation*, Gandhinagar, India, 4–6 Diciembre.

6. **Flores, E.**, M. Ibarra-Romero, L. Moreno, G. Sidorov y P. Rosso (2014c). «Modelos de recuperación de información basados en n-gramas aplicados a la reutilización de código fuente». En: *Proceedings of the 3rd Spanish Conference on Information Retrieval*, págs. 185–188.

[1] Estos números se han obtenido a través de `http://scholar.google.com/` el 2 de Abril de 2016.

7. **Flores**, E., A. Barrón-Cedeño, L. Moreno y P. Rosso (2014b). «Cross-language source code re-use detection». En: *Proceedings of the 3rd Spanish Conference on Information Retrieval*, págs. 145–156.

8. **Flores, E.**, P. Rosso, L. Moreno y E. Villatoro-Tello (2014a). «PAN@FIRE: Overview of SOCO Track on the detection of SOurce COde re-use». En: FIRE 2014 Working Notes. Sixth International Workshop of the Forum for Information Retrieval Evaluation, Bangalore, India, 5–7 Diciembre.

9. **Flores, E.**, A. Barrón-Cedeño, P. Rosso y L. Moreno (2012). «DeSoCoRe: Detecting Source Code Re-Use across programming languages». En: *Proceedings of the Demonstration Session at the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Montréal, Canada: Association for Computational Linguistics, págs. 1-4.

10. **Flores, E.**, A. Barrón-Cedeño, P. Rosso y L. Moreno (2011c). Detección de reutilización de código fuente entre lenguajes de programación en base a la frecuencia de términos. En: *Proceedings IV Jornadas PLN-TIMM*, Torres, Jaén, Abril 7-8, págs. 21–26, ISBN 978-84-15364-00-9.

11. **Flores, E.**, A. Barrón-Cedeño, P. Rosso y L. Moreno (2011b). «Towards the detection of cross-language source code reuse». En: *Natural Language Processing and Information Systems*. Ed. por R. Muñoz, A. Montoyo y E. Métais. Vol. 6716. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, págs. 250–253, DOI: `http://dx.doi.org/10.1007/978-3-642-22327-3_31`.

12. **Flores, E.**, A. Barrón-Cedeño, P. Rosso y L. Moreno (2011a). «Detecting source code reuse across programming languages». En: *27th Conference of the Spanish Society for Natural Language Processing (SEPLN)*, Huelva, 5–7 Septiembre.

Publicaciones parcialmente relacionadas

13. Franco-Salvador, M., I. Bensalem, **E. Flores**, P. Gupta y P. Rosso (2015). «PAN 2015 shared task on plagiarism detection: Evaluation of corpora for text alignment». En: *Working Notes Papers of the CLEF 2015 Evaluation Labs*. CEUR Workshop Proceedings. `http://ceur-ws.org/Vol-1391/`.

La tabla B.1 muestra el impacto de las publicaciones y los apartados relacionados. El trabajo Franco-Salvador et al. (2015) no está centrado en el tema de la tesis, pero está relacionado con este. En concreto trata del etiquetado de corpus translingües con casos de reutilización en lenguas poco estudiadas.

publicación	apartado(s)	factor de impacto	citas (auto-citas)	libros	revistas	conferencias	tesis
1	5.1.3						
2	3.2.2, 5.2.3	0,466	1(1)				
3	3.2.1, 5.1.2	0,296	7(4)	1	1	1	
4	6.1		4(1)		3		
5	6.3		4		4		
6	5.1.3		3(3)				
7	3.2.2, 5.2.3						
8	6.1		6(1)			5	
9	A	CORE A	10(3)	1	2	4	
10	5.2.1						
11	5.2.1	CORE C	18(7)		4	5	2
12	5.2.2						
13			1		1		

Tabla B.1: Descripción de las publicaciones derivadas de la tesis. La información incluye el apartado relacionado en la tesis, factor de impacto y número de citas (con auto-citas) incluyendo: libros, revistas, conferencias y tesis (no se incluyen las auto-citas).

Bibliografía

Arwin, C. y S. Tahaghoghi (2006). «Plagiarism detection across programming languages». En: *Proceedings of the 29th Australian Computer Science Conference, Australian Computer Society* 48, págs. 277-286.

Baer, N. y R. Zeidman (2012). «Measuring Whitespace Pattern Sequences as an Indication of Plagiarism». En: *Journal of Software Engineering and Applications* 5.4, págs. 249-254. DOI: 10.4236/jsea.2012.54029.

Bandara, U. y G. Wijayrathna (2012). «Detection of Source Code Plagiarism Using Machine Learning Approach». En: *International Journal of Computer Theory and Engineering* 4.5, págs. 674-678. DOI: 10.7763/IJCTE.2012.V4.555.

Barrón-Cedeño, A., A. Eiselt y P. Rosso (2009a). «Monolingual Text Similarity Measures: A comparison of Models over Wikipedia Articles Revisions». En: *Proceedings of 7th International Conference on Natural Language Processing*, págs. 29-38.

Barrón-Cedeño, A., P. Rosso, E. Agirre y G. Labaka (2010). «Plagiarism Detection across Distant Language Pairs». En: *Proceedings of the 23rd International Conference on Computational Linguistics, COLING-2010, Beijing, China*, págs. 37-45.

Barrón-Cedeño, A., P. Rosso y J. Benedí (2009b). «Reducing the Plagiarism Detection Search Space on the basis of the Kullback-Leibler Distance». En: *Proceedings 10th International Conference on Computational Linguistics and Intelligent Text Processing (CICLing), Springer-Verlag, LNCS(5449)*, págs. 523-534. DOI: 10.1007/978-3-642-00382-0_42.

Barrón-Cedeño, A. (2010). «On the mono-and cross-language detection of text reuse and plagiarism». En: *Proceedings of the 33rd International Conference on Research and Development in Information retrieval, ACM SIGIR*, págs. 914-914. DOI: 10.1145/1835449.1835687.

Barrón-Cedeño, A., M. Lestari-Paramita, P. Clough y P. Rosso (2014). «A Comparison of Approaches for Measuring Cross-Lingual Similarity of Wikipedia Articles». En: *Proceedings of the 36th European Conference on IR Research: Advances in Information Retrieval*. Vol. 8416.

Lecture Notes in Computer Science. Springer International Publishing, págs. 424-429. DOI: 10.1007/978-3-319-06028-6_36.

Barrón-Cedeño, A. y P. Rosso (2009c). «On automatic plagiarism detection based on n-grams comparison». En: *Proceedings of the 31th European Conference on IR Research: Advances in Information Retrieval*. Springer Berlin Heidelberg, págs. 696-700. DOI: 10.1007/978-3-642-00958-7_69.

Baxter, I. D., A. Yahin, L. Moura, M. S. Anna y L. Bier (1998). «Clone detection using abstract syntax trees». En: *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, págs. 368-377. DOI: 10.1109/ICSM.1998.738528.

Bernstein, Y. y J. Zobel (2004). «A Scalable System for Identifying Co-Derivative Documents». En: *Proceedings of the 11th International Conference on String Processing and Information Retrieval*, págs. 55-67. DOI: 10.1007/978-3-540-30213-1_6.

Brin, S., J. Davis y H. Garcia-Molina (1995). «Copy detection mechanisms for digital documents». En: *ACM SIGMOD Record*. Vol. 24. 2. ACM, págs. 398-409. DOI: 10.1145/223784.223855.

Brown, P., V. Pietra, S. Pietra y R. Mercer (1993). «The mathematics of statistical machine translation: Parameter estimation». En: *Computational linguistics* 19.2, págs. 263-311.

Burrows, S., S. M. M. Tahaghoghi y J. Zobel (2007). «Efficient plagiarism detection for large code repositories». En: *Software: Practice and Experience* 37.2, págs. 151-175. DOI: 10.1002/spe.v37:2.

Chae, D.-K., J. Ha, S.-W. Kim, B. Kang y E. G. Im (2013). «Software plagiarism detection: a graph-based approach». En: *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. ACM, págs. 1577-1580. DOI: 10.1145/2505515.2507848.

Chapman, K. y R. Lupton (2004). «Academic dishonesty in a global educational market: A comparison of Hong Kong and American university business students». En: *International Journal of Educational Management* 18.7, págs. 425-435. DOI: 10.1108/09513540410563130.

Chuda, D., P. Navrat, B. Kovacova y P. Humay (2012). «The issue of (software) plagiarism: A student view». En: *IEEE Transactions on Education* 55.1, págs. 22-28. DOI: 10.1109/TE.2011.2112768.

Chunhui, W., L. Zhiguo y L. Dongsheng (2013). «Preventing and Detecting Plagiarism in Programming Course». En: *International Journal of Security and Its Applications* 7.5, págs. 269-278.

Clough, P. y M. Stevenson (2010). «Developing A Corpus of Plagiarised Short Answers». En: *Language Resources and Evaluation: Special Issue on Plagiarism and Authorship Analysis* 45.1, págs. 5-24. DOI: 10.1007/s10579-009-9112-1.

Clough, P. (2000). «Plagiarism in natural and programming languages: an overview of current tools and technologies». En: *Research Memoranda: CS-00-05, Department of Computer Science, University of Sheffield, UK*, págs. 1-31.

Clough, P. (2003). «Old and new challenges in automatic plagiarism detection». En: *National Plagiarism Advisory Service*, págs. 391-407.

Cosma, G. y M. Joy (2008). «Towards a Definition of Source-Code Plagiarism». En: *IEEE Transactions on Education* 51.2, págs. 195-200.

Cosma, G. y M. Joy (2012). «Evaluating the performance of LSA for source-code plagiarism detection». En: *Informatica* 36.4, págs. 409-424.

Cui, B., J. Li, T. Guo, J. Wang y D. Ma (2010). «Code comparison system based on abstract syntax tree». En: *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*. IEEE, págs. 668-673. DOI: 10.1109/ICBNMT.2010.5705174.

Cullum, J. K. y R. A. Willoughby (1985). «Real rectangular matrices». En: *Lanczos algorithms for large symmetric eigenvalue computations*. Vol. 1. Boston: Brikhauser. Cap. 5, págs. 273-359. DOI: 10.1007/978-1-4684-9178-4_6.

Deerwester, S., S. T. Dumais, G. W. Furnas, T. K. Landauer y R. Harshman (1990). «Indexing by latent semantic analysis». En: *Journal of the American Society for Information Science* 41.6, págs. 391-407. DOI: 10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9.

Dreher, H. (2007). «Automatic conceptual analysis for plagiarism detection». En: *Issues in Informing Science and Information Technology* 4, págs. 601-628.

El Bachir Menai, M. y N. Al-Hassoun (2010). «Similarity detection in Java programming assignments». En: *Proceedings of the 5th International Conference on Computer Science and Education*, págs. 356-361. DOI: 10.1109/ICCSE.2010.5593613.

Ester, M., H.-P. Kriegel, J. Sander y X. Xu (1996). «A density-based algorithm for discovering clusters in large spatial databases with noise». En: *Kdd*. Vol. 96. 34, págs. 226-231.

Faidhi, J. y S. Robinson (1987). «An empirical approach for detecting program similarity and plagiarism within a university programming environment». En: *Computers & Education* 11.1, págs. 11-19. DOI: 10.1016/0360-1315(87)90042-X.

Feng, J., B. Cui y K. Xia (2013). «A code comparison algorithm based on AST for plagiarism detection». En: *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on*. IEEE, págs. 393-397. DOI: 10.1109/EIDWT.2013.73.

Fisher, S. R. A., S. Genetiker, R. A. Fisher, S. Genetician, R. A. Fisher y S. Généticien (1960). *The design of experiments*. Vol. 12. 6. Oliver y Boyd Edinburgh.

Fleiss, J. L. (1971). «Measuring nominal scale agreement among many raters». En: *Psychological bulletin* 76.5, pág. 378.

Flores, E., P. Rosso, L. Moreno y E. Villatoro-Tello (2014a). «PAN@FIRE: Overview of SOCO Track on the Detection of SOurce COde Re-use». En: *FIRE 2014 Working Notes. Sixth International Workshop of the Forum for Information Retrieval Evaluation, Bangalore, India, 5–7 December*.

Flores, E., P. Rosso, E. Villatoro-Tello, L. Moreno, R. Alcover y V. Chirivella (2015a). «PAN@FIRE: Overview of CL-SOCO track on the Detection of Cross-Language SOurce COde Re-use». En: *FIRE 2015 Working Notes. Seventh International Workshop of the Forum for Information Retrieval Evaluation, Gandhinagar, India, 4–6 December*.

Flores, E. (2012). «Reutilización de código fuente entre lenguajes de programación». Tesis de Máster. Universitat Politécnica de Valéncia.

Flores, E., A. Barrón-Cedeño, L. Moreno y P. Rosso (2014b). «Cross-Language Source Code Re-Use Detection». En: *Proceedings of the 3rd Spanish Conference on Information Retrieval (CERI)*, págs. 145-156.

Flores, E., A. Barrón-Cedeño, L. Moreno y P. Rosso (2015b). «Cross-Language Source Code Re-Use Detection Using Latent Semantic Analysis». En: *Journal of Universal Computer Science* 21.13, págs. 1708-1725. DOI: 10.3217/jucs-021-13-1708.

Flores, E., A. Barrón-Cedeño, L. Moreno y P. Rosso (2015c). «Uncovering source code reuse in large-scale academic environments». En: *Computer Applications in Engineering Education* 23.3, págs. 383-390. DOI: 10.1002/cae.21608.

Flores, E., A. Barrón-Cedeño, P. Rosso y L. Moreno (2011). «Towards the Detection of Cross-Language Source Code Reuse». En: *Natural Language Processing and Information Systems*. Ed. por R. Muñoz, A. Montoyo y E. Métais. Vol. 6716. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, págs. 250-253. DOI: 10.1007/978-3-642-22327-3_31.

Flores, E., A. Barrón-Cedeño, P. Rosso y L. Moreno (2012). «DeSoCoRe: Detecting Source Code Re-Use across Programming Languages». En: *Proceedings of the Demonstration Session at the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Montréal, Canada: Association for Computational Linguistics, págs. 1-4.

Flores, E., M. Ibarra-Romero, L. Moreno, G. Sidorov y P. Rosso (2014c). «Modelos de recuperación de información basados en n-gramas aplicados a la reutilización de código fuente». En: *Proceedings of the 3rd Spanish Conference on Information Retrieval (CERI)*, págs. 185-188.

Flores, E., L. Moreno y P. Rosso (2016). «Detecting Source Code Re-Use with Ensemble Models». En: *Proceedings of the 4th Spanish Conference on Information Retrieval*. CERI '16. ACM, 16:1-16:7. DOI: 10.1145/2934732.2934738.

Flores, E., P. Rosso, L. Moreno y E. Villatoro-Tello (2015d). «On the Detection of SOurce COde Reuse». En: *Proceedings of the Forum for Information Retrieval Evaluation*. FIRE '14. Bangalore, India: ACM, págs. 21-30. DOI: 10.1145/2824864.2824878.

Franco-Salvador, M., I. Bensalem, E. Flores, P. Gupta y P. Rosso (2015). «PAN 2015 Shared Task on Plagiarism Detection: Evaluation of Corpora for Text Alignment». En: *Working Notes Papers of the CLEF 2015 Evaluation Labs*. Vol. 1391. CLEF y CEUR-WS.org.

Frantzeskou, G., S. MacDonell, E. Stamatatos y S. Gritzalis (2008). «Examining the significance of high-level programming features in source code author classification». En: *Journal of Systems and Software* 81.3, págs. 447-460. DOI: 10.1016/j.jss.2007.03.004.

Gabrilovich, E. y S. Markovitch (2007). «Computing Semantic Relatedness Using Wikipedia-based Explicit Semantic Analysis». En: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. Vol. 7, págs. 1606-1611.

Ganguly, D. y G. Jones (2014). «DCU@FIRE-2014: An Information Retrieval Approachfor Source Code Plagiarism Detection». En: *FIRE 2014 Working Notes. Proceedings of the Sixth International Workshop of the Forum for Information Retrieval Evaluation*. Ed. por P. Majumder, M. Mitra, M. Agrawal y P. Mehta. Bangalore, India.

Ganguly, D. y G. J. F. Jones (2015). «DCU@FIRE-2014: An Information Retrieval Approach for Source Code Plagiarism Detection». En: *Proceedings of the Forum for Information Retrieval Evaluation*. FIRE '14. Bangalore, India: ACM, págs. 39-42.

García-Hernández, R. e Y. Lendeneva (2014). «Identification of similar source codes based on longest common substrings». En: *FIRE 2014 Working Notes. Proceedings of the Sixth International Workshop of the Forum for Information Retrieval Evaluation*. Ed. por P. Majumder, M. Mitra, M. Agrawal y P. Mehta. Bangalore, India.

García-Hernández, R. e Y. Lendeneva (2015). «Cross-Language Identification of Similar Source Codes based on Longest Common Substrings». En: *FIRE 2015 Working Notes. Proceedings of the Seventh International Workshop of the Forum for Information Retrieval Evaluation*. Ed. por P. Majumder, M. Mitra, M. Agrawal y P. Mehta. Gandhinagar, India.

García-Hernández, R. A., J. F. Martínez-Trinidad y J. A. Carrasco-Ochoa (2006). «A new algorithm for fast discovery of maximal sequential patterns in a document collection». En: *Computational Linguistics and Intelligent Text Processing*. Springer, págs. 514-523.

Genest, D. y M. Chein (1997). «An experiment in document retrieval using conceptual graphs». En: *Proceedings of the Conceptual Structures: Fulfilling Peirce's Dream: Fifth International Conference on Conceptual Structures*. Springer Berlin Heidelberg, págs. 489-504. DOI: 10.1007/BFb0027893.

Halstead, M. H. (1972). «Natural Laws Controlling Algorithm Structure?» En: *SIGPLAN Not.* 7.2, págs. 19-26. DOI: 10.1145/953363.953366.

Halstead, M. H. (1977). *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc.

Harrison, W. A. y K. I. Magel (1981). «A Complexity Measure Based on Nesting Level». En: *ACM Sigplan Notices* 16.3, págs. 63-74. DOI: 10.1145/947825.947829.

Heblikar, S., P. Sharma, M. Munnangi y C. Bankapur (2014). «Normalization based stop-word approach to source code reuse detection». En: *FIRE 2014 Working Notes. Proceedings of the Sixth International Workshop of the Forum for Information Retrieval Evaluation*. Ed. por P. Majumder, M. Mitra, M. Agrawal y P. Mehta. Bangalore, India.

Hellyer, L. y L. Beadle (2009). «Detecting Plagiarism in Microsoft Excel Assignments». En: *10th Annual Conference of the Subject Centre for Information and Computer Sciences*. Ed. por H. White. HE Academy, págs. 182-196.

Hislop, G. W. (1998). «Analyzing existing software for software reuse». En: *Journal of Systems and Software* 41.1, págs. 33-40. DOI: 10.1016/S0164-1212(97)10004-8.

Hunt, J. J. y W. F. Tichy (2002). «Extensible language-aware merging». En: *Proceedings of the International Conference on Software Maintenance*, págs. 511-520. DOI: 10.1109/ICSM.2002.1167812.

Jaccard, P. (1901). *Étude comparative de la distribution florale dans une portion des Alpes et des Jura*. 37. Impr. Corbaz, págs. 547-579.

Jankowitz, H. T. (1988). «Detecting Plagiarism in Student Pascale Programs». En: *The Computer Journal* 31.1, págs. 1-8. DOI: 10.1093/comjnl/31.1.1.

Joy, M. y M. Luck (1999). «Plagiarism in programming assignments». En: *IEEE Transactions on Education* 42.2, págs. 129-133. DOI: 10.1109/13.762946.

Kang, N., A. Gelbukh y S. Han (2006). «PPChecker: Plagiarism Pattern Checker in Document Copy Detection». En: *Proceedings of 9th International Conference on Text, Speech and Dialogue*, págs. 661-667. DOI: 10.1007/11846406_83.

Karp, R. M. y M. O. Rabin (1987). «Efficient randomized pattern-matching algorithms». En: *IBM Journal of Research and Development* 31.2, págs. 249-260. DOI: 10.1147/rd.312.0249.

Kešelj, V., F. Peng, N. Cercone y C. Thomas (2003). «N-gram-based author profiles for authorship attribution». En: *Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING*. Vol. 3, págs. 255-264.

Koschke, R. (2007). «Survey of Research on Software Clones». En: *Duplication, Redundancy, and Similarity in Software*. Ed. por R. Koschke, E. Merlo y A. Walenstein. Dagstuhl Seminar Proceedings 06301. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Krinke, J. (2001). «Identifying similar code with program dependence graphs». En: *Proceedings of the 8th Working Conference on Reverse Engineering*. IEEE, págs. 301-309. DOI: 10.1109/WCRE.2001.957835.

Kullback, S. y R. A. Leibler (1951). «On Information and Sufficiency». En: *The annals of mathematical statistics* 22.1, págs. 79-86. DOI: 10.2307/2236703.

Kuo, J. Y. y F. C. Huang (2010). «Code analyzer for an online course management system». En: *Journal of Systems and Software* 83.12, págs. 2478-2486. DOI: 10.1016/j.jss.2010.07.037.

Landauer, T. K. y M. L. Littman (1990). «Fully automatic cross-language document retrieval using latent semantic indexing». En: *Proceedings of the Sixth Annual Conference of the UW Centre for the New Oxford English Dictionary and Text Research*. UW Centre for the New OED y Text Research, págs. 31-38.

Landis, J. R. y G. G. Koch (1977). «The Measurement of Observer Agreement for Categorical Data». En: *Biometrics* 33.1, págs. 159-174. DOI: 10.2307/2529310.

Lange, R. C. y S. Mancoridis (2007). «Using Code Metric Histograms and Genetic Algorithms to Perform Author Identification for Software Forensics». En: *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. ACM, págs. 2082-2089. DOI: 10.1145/1276958.1277364.

Levenshtein, V. I. (1966). «Binary codes capable of correcting deletions, insertions, and reversals». En: *Soviet physics doklady*. Vol. 10. 8, págs. 707-710.

Majumder, P., M. Mitra, M. Agrawal y P. Mehta, eds. (2014). *FIRE 2014 Working Notes. Proceedings of the Sixth International Workshop of the Forum for Information Retrieval Evaluation*. Bangalore, India.

Manber, U. (1994). «Finding Similar Files in a Large File System». En: *Usenix Winter Technical Conference*. Vol. 94, págs. 1-10.

Marcus, A., A. Sergeyev, V. Rajlich y J. I. Maletic (2004). «An information retrieval approach to concept location in source code». En: *Proceedings of 11th Working Conference on Reverse Engineering*. IEEE, págs. 214-223. DOI: 10.1109/WCRE.2004.10.

Marinescu, D., A. Baicoianu y S. Dimitriu (2013). «A plagiarism detection system in computer source code». En: *International Journal of Computer Science Research and Application* 3.1, págs. 22-30.

Marinescu, D., A. Baicoianu y S. Dimitriu (2012). «Software for plagiarism detection in computer source code». En: *Proc. 7th International Conference on Virtual Learning*, págs. 373-379.

McCabe, T. J. (1976). «A Complexity Measure». En: *IEEE Transactions on Software Engineering* 2.4, págs. 308-320. DOI: 10.1109/TSE.1976.233837.

McNamee, P. y J. Mayfield (2004). «Character N-Gram Tokenization for European Language Text Retrieval». En: *Information Retrieval* 7.1, págs. 73-97. DOI: 10.1023/B:INRT.0000009441.78971.be.

Meyer, D. (2012). «Analyzing the Robustness of Clone Detection Tools Regarding Code Obfuscation». Tesis de grado. University of Magdeburg.

Mishne, G. (2003). «Source Code Retrieval using Conceptual Graphs». Tesis de Máster. Institute for Logic, Language y Computation (ILLC), University of Amsterdam.

Narayanan, S. y S. Simi (2012). «Source code plagiarism detection and performance analysis using fingerprint based distance measure method». En: *Proceedings of the 7th International Conference on Computer Science & Education (ICCSE)*. IEEE, págs. 1065-1068. DOI: 10.1109/ICCSE.2012.6295247.

Och, F. J. y H. Ney (2003). «A Systematic Comparison of Various Statistical Alignment Models». En: *Computational Linguistics* 29.1, págs. 19-51. DOI: 10.1162/089120103321337421.

Palkovskii, Y. (2014). «Undefined title». En: *FIRE 2014 Working Notes. Proceedings of the Sixth International Workshop of the Forum for Information Retrieval Evaluation*. Ed. por P. Majumder, M. Mitra, M. Agrawal y P. Mehta. Bangalore, India.

Pinto, D., J. Civera, A. Barrón-Cedeño, A. Juan y P. Rosso (2009). «A statistical approach to crosslingual natural language tasks». En: *Journal of Algorithms* 64.1, págs. 51-60. DOI: 10.1016/j.jalgor.2009.02.005.

Potthast, M., A. Barrón-Cedeño, B. Stein y P. Rosso (2011). «Cross-language plagiarism detection». En: *Language Resources and Evaluation, Special Issue on Plagiarism and Authorship Analysis* 45.1, págs. 45-62. DOI: 10.1007/s10579-009-9114-z.

Potthast, M., B. Stein y M. Anderka (2008). «A Wikipedia-Based Multilingual Retrieval Model». En: *Proc. 30th European Conference on IR Research (ECIR), Springer Berlin Heidelberg, LNCS(4956)*, págs. 522-530.

Potthast, M., A. Barrón-Cedeño, A. Eiselt, B. Stein y P. Rosso (2010a). «Overview of the 2nd International Competition on Plagiarism Detection.» En: *CLEF (Notebook Papers/LABs/-Workshops)*.

Potthast, M., B. Stein, A. Barrón-Cedeño y P. Rosso (2010b). «An evaluation framework for plagiarism detection». En: *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*. Association for Computational Linguistics, págs. 997-1005.

Potthast, M., B. Stein, A. Eiselt, A. Barrón-Cedeño y P. Rosso (2009). «Overview of the 1st International Competition on Plagiarism Detection». En: *SEPLN 09 Workshop on Uncovering Plagiarism, Authorship, and Social Software Misuse (PAN 09)*. Ed. por B. Stein, P. Rosso, E. Stamatatos, M. Koppel y E. Agirre. CEUR-WS.org, págs. 1-9.

Pouliquen, B., R. Steinberger y C. Ignat (2003). «Automatic Identification of Document Translations in Large Multilingual Document Collections». En: *Proceedings of the International Conference on Recent Advances in Natural Language Processing*, págs. 401-408.

Pramono, Y. y Suhardi (2014). «Detecting plagiarism in cross-platform mobile applications: Case study: Game application similarity in Symbian platform and Android platform». En: *International Conference on Information Technology Systems and Innovation (ICITSI)*, págs. 159-164.

Prechelt, L., G. Malpohl y M. Philippsen (2002). «Finding plagiarisms among a set of programs with JPlag». En: *Journal of Universal Computer Science* 8.11, págs. 1016-1038.

Ramírez-de-la-Cruz, A., G. Ramírez-de-la-Rosa, C. Sánchez-Sánchez, H. Jiménez-Salazar, C. Rodríguez-Lucatero y L.-R. W. (2014a). «High level features for detecting source code plagiarism across programming languages». En: *FIRE 2014 Working Notes. Proceedings of the Sixth International Workshop of the Forum for Information Retrieval Evaluation*. Ed. por P. Majumder, M. Mitra, M. Agrawal y P. Mehta. Bangalore, India.

Ramírez-de-la-Cruz, A., G. Ramírez-de-la-Rosa, C. Sánchez-Sánchez, W. A. Luna-Ramírez, H. Jiménez-Salazar y C. Rodríguez-Lucatero (2014b). «UAM@SOCO 2014: Detection of Source Code Reuse by means of combining different types of representations». En: *FIRE 2014 Working Notes. Proceedings of the Sixth International Workshop of the Forum for Information Retrieval Evaluation*. Ed. por P. Majumder, M. Mitra, M. Agrawal y P. Mehta. Bangalore, India.

Ramírez-de-la-Cruz, A., G. Ramírez-de-la-Rosa, C. Sánchez-Sánchez y H. Jiménez-Salazar (2015). «On the Importance of Lexicon, Structure and Style for Identifying Source Code Plagiarism». En: *Proceedings of the Forum for Information Retrieval Evaluation*. FIRE '14. Bangalore, India: ACM, págs. 31-38.

Rivest, R. (1992). *The MD5 message-digest algorithm*. RFC 1320. RFC Editor, págs. 1-21.

Robertson, S. y S. Walker (1999). «Okapi/Keenbow at TREC-8». En: *The Eighth Text Retrieval Conference (TREC-8)*, págs. 151-162.

Robertson, S. E., S. Walker, M. Hancock-Beaulieu, A. Gull y M. Lau (1992). «Okapi at TREC». En: *Text retrieval conference*, págs. 21-30.

Rosales, F., A. García, S. Rodríguez, J. L. Pedraza, R. Méndez y M. M. Nieto (2008). «Detection of Plagiarism in Programming Assignments». En: *IEEE Transactions on Education* 51.2, págs. 174-183. DOI: 10.1109/TE.2007.906778.

Rudman, J. «The State of Authorship Attribution Studies: Some Problems and Solutions». En: *Computers and the Humanities* 31.4, págs. 351-365. DOI: 10.1023/A:1001018624850.

Salton, G. y M. J. McGill (1986). *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc.

Schleimer, S., D. S. Wilkerson y A. Aiken (2003). «Winnowing: Local Algorithms for Document Fingerprinting». En: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, págs. 76-85. DOI: 10.1145/872757.872770.

Schulze, S. y D. Meyer (2013). «On the Robustness of Clone Detection to Code Obfuscation». En: *Proceedings of the 7th International Workshop on Software Clones*. IEEE Press, págs. 62-68.

Selby, R. W. (1989). «Quantitative studies of software reuse». En: *Software Reusability* 2, págs. 213-233. DOI: 10.1145/75722.75733.

Shah, D., H. Jethani y H. Joshi (2014). «(CLSCR) Cross Language Source Code Reuse Detection using Intermediate Language». En: *FIRE 2014 Working Notes. Proceedings of the Sixth International Workshop of the Forum for Information Retrieval Evaluation*. Ed. por P. Majumder, M. Mitra, M. Agrawal y P. Mehta. Bangalore, India.

Shivakumar, N. y H. Garcia-Molina (1995). «SCAM: A copy detection mechanism for digital documents». En: *Proceedings of the 2nd International Conference on Theory and Practice of Digital Libraries*.

Si, A., H. V. Leong y R. W. Lau (1997). «CHECK: A Document Plagiarism Detection System». En: *Proceedings of the 1997 ACM symposium on Applied computing*. ACM, págs. 70-77. DOI: 10.1145/331697.335176.

Simard, M., G. Foster y P. Isabelle (1993). «Using Cognates to Align Sentences in Bilingual Corpora». En: *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Distributed Computing - Volume 2*. Vol. 2, págs. 1071-1082.

Sparck, K. y C. van Rijsbergen (1975). *Report on the need for and provision of an "ideal" information retrieval test collection. British Library Research and Development Report, University of Cambridge*.

Spinellis, D., P. Zaharias y A. Vrechopoulos (2007). «Coping with plagiarism and grading load: Randomized programming assignments and reflective grading». En: *Computer Applications in Engineering Education* 15.2, págs. 113-123. DOI: 10.1002/cae.20096.

Stamatatos, E. (2009a). «A survey of modern authorship attribution methods». En: *Journal of the American Society for information Science and Technology* 60.3, págs. 538-556. DOI: 10.1002/asi.21001.

Stamatatos, E. (2009b). «Intrinsic plagiarism detection using character n-gram profiles». En: *Proceedings of the SEPLN'09 Workshop on Uncovering Plagiarism, Authorship and Social Software Misuse*. Vol. 2, págs. 38-46.

Stamatatos, E. (2011). «Plagiarism detection using stopword n-grams». En: *Journal of the American Society for Information Science and Technology* 62.12, págs. 2512-2527. DOI: 10.1002/asi.21630.

Stamatatos, E., M. Potthast, F. Rangel, P. Rosso y B. Stein (2015). «Overview of the PAN/CLEF 2015 Evaluation Lab». En: *Experimental IR Meets Multilinguality, Multimodality, and Interaction. 6th International Conference of the CLEF Initiative (CLEF 15)*. Ed. por J. Mothe, J. Savoy, J. Kamps, K. Pinel-Sauvagnat, G. Jones, E. SanJuan, L. Cappellato y N. Ferro. Berlin Heidelberg New York: Springer, págs. 518-538.

Stein, B., M. Koppel y E. Stamatatos (2007a). «Plagiarism analysis, authorship identification, and near-duplicate detection». En: *SIGIR Forum (PAN 2007)* 41.2, págs. 68-71.

Stein, B., S. M. zu Eissen y M. Potthast (2007b). «Strategies for retrieving plagiarized documents». En: *Proceedings of the 30th International Conference on Research and Development in Information retrieval, ACM SIGIR*. ACM, págs. 825-826.

Steinberger, R., B. Pouliquen, A. Widiger, C. Ignat, T. Erjavec, D. Tufis y D. Varga (2006). «The JRC-Acquis: A multilingual aligned parallel corpus with 20+ languages». En: *Proceedings of Language Resources and Evaluation Conference*.

Tu, Q. (2002). «On navigation and analysis of software architecture evolution». Tesis de Máster. University of Waterloo.

Turnitin en PoliformaT (2015). Noticia de la implantación de la herramienta Turnitin en la plataforma PoliformaT. (Visitado 12-03-2015).

Velez, M., D. Qiu, Y. Zhou, E. T. Barr y Z. Su (2015). «On the lexical distinguishability of source code». En: *arXiv preprint arXiv:1502.01410*.

Voga, M. y J. Grainger (2007). «Cognate status and cross-script translation priming». En: *Memory & Cognition* 35.5, págs. 938-952. DOI: 10.3758/BF03193467.

Whale, G. (1990). «Software metrics and plagiarism detection». En: *Journal of Systems and Software* 13.2, págs. 131-138. DOI: 10.1016/0164-1212(90)90118-6.

Wise, M. J. (1992). «Detection of Similarities in Student Programs: YAP'Ing May Be Preferable to Plague'Ing». En: *Proceedings of the 23rd SIGCSE Technical Symposium on Computer Science Education*. ACM, págs. 268-271.

Witten, I. H., A. Moffat y T. C. Bell (1999). *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc.

Xiong, H., H. Yan, Z. Li y H. Li (2009). «BUAA_AntiPlagiarism: A System To Detect Plagiarism for C Source Code». En: *International Conference on Computational Intelligence and Software Engineering. CiSE 2009*, págs. 1-5. DOI: 10.1109/CISE.2009.5366790.

Yang, E.-h. y J. C. Kieffer (1998). «On the performance of data compression algorithms based upon string matching». En: *IEEE Transactions on Information Theory* 44.1, págs. 47-65. DOI: 10.1109/18.650987.

Yen, S. J., Y. S. Lee, C. H. Lin y J. C. Ying (2006). «Investigating the Effect of Sampling Methods for Imbalanced Data Distributions». En: *IEEE International Conference on Systems, Man and Cybernetics, 2006. SMC '06*. Vol. 5, págs. 4163-4168. DOI: 10.1109/ICSMC.2006.384787.

Zhang, L., Y. Zhuang y Z. Yuan (2007). «A program plagiarism detection model based on information distance and clustering». En: *Internacional Conference on Intelligent Pervasive Computing*, págs. 431-436. DOI: 10.1109/IPC.2007.10.

Ziv, J. y A. Lempel (1977). «A universal algorithm for sequential data compression». En: *IEEE Transactions on Information Theory* 23.3, págs. 337-343.

Índice alfabético