Detección de similitudes entre códigos aplicando el algoritmo de Winnowing a fingerprints generadas por árboles de sintaxis

José Ángel González Carrera[1], Carlos Daniel Díaz Arrazate[2], Oscar Sebastián Martínez Sánchez[3], Jesús Jiménez Aguilar[4], and Carlos Eduardo Ruiz Lira[5]

[1]Ing. en Tecnologías Computacionales

June 5, 2024

Abstract

El sistema propuesto en este artículo se basa en el análisis léxico-sintáctico del código fuente. Este análisis incluye la extracción de características decorativas, como el uso de comentarios, nombres de identificadores, variables y funciones, así como la estructura del código en el documento a partir del procesamiento del árbol de sintaxis generado. Utilizando un método basado en fingerprinting entre códigos base para Python, que emplea el algoritmo de Winnowing y una variación de la implementación de K-Grams basada en nodos consecutivos en el árbol de sintaxis, el sistema permite determinar un índice de similitud entre múltiples códigos. Además, proporciona una visualización de las secciones coincidentes para facilitar el proceso de identificación de plagio entre ellos.

Keywords: Análisis léxico-sintáctico, Detección de plagio, Código fuente, Fingerprinting, Algoritmo de Winnowing, K-Grams, Árbol de sintaxis, Similitud de código, Visualización de coincidencias, Python.

1 Introducción

La integridad académica es un tema fundamental en cualquier institución educativa, ya que promueve el desarrollo ético, académico y profesional de los estudiantes. El plagio, por lo tanto, representa una amenaza a este principio; debilita y afecta los valores fundamentales del aprendizaje y la investigación. Cuando los estudiantes recurren al plagio, no solo comprometen su desarrollo sino que socavan los valores de la honestidad, la integridad intelectual y el esfuerzo personal en la adquisición de conocimientos.[**5**]

El plagio puede presentarse de diferentes maneras y se puede clasificar de acuerdo a la severidad de la agresión. De acuerdo con [**6**], entre las prácticas comunes se encuentra la clonación completa del

documento, la duplicación o reciclado de trabajos propios, el parafraseo del trabajo original o combinar ideas provenientes de múltiples fuentes en un solo documento, entre otras. Actualmente, existen múltiples herramientas y sistemas de software enfocadas en la detección de plagio en documentos, entre las cuales se incluyen Turnitin, Grammarly, Plagscan, entre otras.

Sin embargo, la detección de plagio sigue representando un desafío en asignaturas de programación, pues a diferencia del análisis de documentos de texto simples, la detección de plagio en código fuente requiere un análisis sintáctico y semántico dependiente del lenguaje de programación utilizado en el trabajo. El plagio en la programación, por lo tanto, puede adoptar distintas formas [**4**]: desde copiar directamente el código, realizar modificaciones superficiales como renombrar identificadores, hasta parafrasear código, lo que implica reescribir usando una sintaxis alternativa.

Existen distintas alternativas para la detección de plagio, las cuales tienen alcances diferentes. Los grafos de dependencia [**1**] representan las relaciones y dependencias entre partes del código, detectando similitudes más allá de la sintaxis. Otra solución presenta la identificación de clones [**2**] en base a cuatro puntos de comparación: nombre, expresiones y flujo de control. Un enfoque distinto se basa en técnicas en algoritmos de k-grams y hashing [**3**] para encontrar similitudes en base a las huellas digitales del código.

No obstante, determinar el plagio entre trabajos sigue estando a discreción del evaluador, pues los programas de software son únicamente capaces de determinar un índice de similitud, y no la razón de por qué son similares entre sí. Además, existe una filosofía que argumenta que si un alumno es capaz de realizar modificaciones al código de tal forma que no es detectado, el alumno ha demostrado las habilidades necesarias como para haber desarrollado su propio trabajo [**4**]. En concordancia a esta filosofía, el análisis de similitud de código fuente a nivel sintáctico resulta ser el análisis más fructífero a desarrollar en la comparación de trabajos.

El resto del artículo se estructura de la siguiente forma. La sección 2 presenta una revisión de literatura sobre el estado del arte actual en cuanto a técnicas de detección de similitud en códigos fuente, incluyendo incluyendo la subsecuencia común más larga (LCS), los árboles de dependencia y el algoritmo de Winnowing. Para cada técnica se presenta un análisis de las ventajas y desventajas de la técnica así como una comparativa entre las mismas. La sección 3 presenta el diseño y justificación del método propuesto en base a la generación del árbol de sintaxis para la generación de los K-Grams del fingerprint del documento con el algoritmo de Winnowing. La sección 4 presenta una serie de experimentación del sistema con un conjunto de códigos fuente para generar el índice de similitud y su comparación con otros sistemas existentes. Finalmente la sección 5 concluye este artículo y ofrecen áreas de interés a desarrollar a futuro.

2 Revisión de literatura

El algoritmo propuesto en el artículo [**2**] se basa en el concepto de la subsecuencia común más larga (LCS, por sus siglas en inglés), problema crucial en el campo de la informática y la programación dinámica. El problema consiste en encontrar la subsecuencia más larga en ambas secuencias dadas; se obtiene de otra secuencia al eliminar algunos elementos o ninguno, sin alterar el orden de los elementos rasantes. Este artículo contribuye con una solución eficiente y precisa, orientada a detectar

plagio en tareas estudiantiles mediante la identificación de funciones clonadas en el código.

En el artículo [**2**] introduce una métrica para detectar la clonación de funciones en el código, considerando aspectos cosméticos como renombrar variables, modificar la documentación, cambiar el orden de las funciones en el archivo y reorganizar el código. Además, se utilizan varios puntos de comparación para los clones, tales como el nombre y la estructura de la función, frecuencia de estructuras de código(comentarios y nombres de las variables), y se define una delta para cada métrica. Si el delta supera el umbral predefinido, la métrica se considera similar, y si todas las métricas son similares, el código lo clasifica como un clon. También se analizan las expresiones (naturaleza y complejidad de las expresiones de la función) y el flujo de control (número de decisiones, bucles, recursividad, nodos y formas de salir de una función).

En el artículo [**8**] menciona que las gráficas de dependencia son herramientas visuales que se utilizan para mostrar la relación entre variables en un conjunto de datos. Estas gráficas son especialmente útiles para identificar patrones, tendencias y posibles correlaciones entre las variables, lo que puede ser crucial para la toma de decisiones.

Aunque se evaluó la posibilidad de emplear este formato junto con gráficas de dependencia para alcanzar nuestra solución. Sin embargo, se encontró que no satisfacía completamente los requisitos del proyecto. No obstante, el umbral de aceptación y procesamiento de los datos eran muy similares a lo que planeábamos utilizar. Esta evaluación nos ayudó a delimitar el alcance de nuestra solución.

En el documento [**10**] se menciona en el cual el análisis simbólico se usa para registrar el comportamiento de ejecución de un programa. El análisis simbólico permite expresar expresiones de programas en forma simbólica, lo que ayuda a derivar el comportamiento funcional de un programa a partir de representaciones algebraicas de sus cálculos. Mientras que durante la ejecución normal del programa se calcula el valor numérico, se pierde la información sobre cómo se alcanza ese valor. Por lo tanto, el análisis simbólico nos ayuda a entender la relación entre diferentes cálculos, lo que resulta útil para optimizar el programa utilizando técnicas como la propagación constante, la reducción de fuerza y la eliminación de cálculos redundantes. Además, facilita la comprensión e ilustración del análisis regional del programa, contribuyendo a su optimización, paralelización y comprensión.

Por otro lado, el algoritmo de Winnowing es utilizado para la detección de plagio en el documento [**4**]. Este algoritmo genera k-grams, los cuales son subsecuencias de caracteres o palabras, donde cada subsecuencia es de un tamaño arbitrario k. Posteriormente, esos k-grams pasan por un proceso de hashing y son utilizados para generar la huella del documento. La parte de la generación de la huella es fundamental para la eficiencia y eficacia del algoritmo. La huella es generada mediante la selección del valor de los hash de algunos k-grams, tomando en cuenta una ventana con un tamaño arbitrario de w. Contando con todas las huellas de los documentos, se procede a comparar entre sí para determinar si existe similitud entre las secuencias.

Cada uno de los artículos antes mencionados sigue un enfoque distinto para la detección de plagio entre códigos. El artículo relacionado a la subsecuencia más larga es sencillo de implementar, pero limitado a no detectar cambios superficiales en el código. El enfoque sobre los árboles de dependencia es más complejo y puede identificar plagios incluso con cambios en la sintaxis, lo cual no coincide con el objetivo de nuestra solución. Finalmente, el artículo respecto al algoritmo de winnowing, presenta una forma de encontrar similitud entre textos.

Tomando como base el algoritmo de winnowing, es posible realizar modificaciones para que sea capaz de detectar plagio entre código. En lugar de carácteres de un texto, la unidad de las secuencias será un nodo de un árbol de sintaxis, lo que permite el uso del algoritmo pero con el enfoque de la detección de código.

3 Propuesta de solucion

Basado en el enfoque de comparación de huellas digitales (fingerprints) para identificar similitudes entre códigos, la Figura 1 ilustra el proceso completo que sigue un archivo de Python para generar su huella digital. Este proceso incluye la generación de un árbol de sintaxis, el filtrado de nodos del árbol y el agrupamiento y hashing de k-grams.

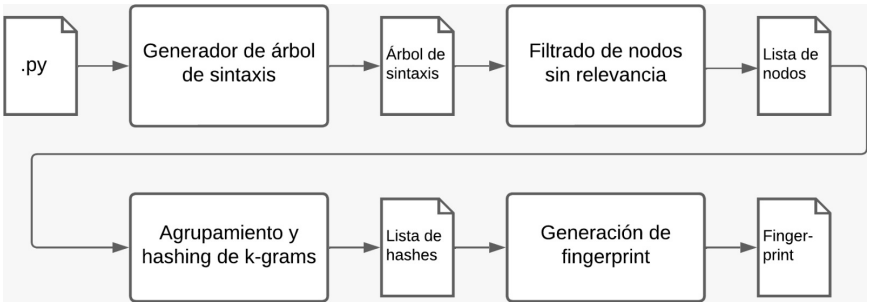

Figure 1: Pasos para la generación de un fingerprint de un archivo .py.

Las 4 fases del proceso consiste en lo siguiente:

1. **Generación de árbol de sintaxis**.- El primer paso consiste en la obtención de un árbol de sintaxis abstracta, esto a través de la librería estándar de Python, ast, el cual está conformado por nodos, los cuales respectivamente almacenan el tipo del nodo, su valor y su posición en el código. La Figura 2 muestra la representación de un árbol de sintaxis obtenido de un código.

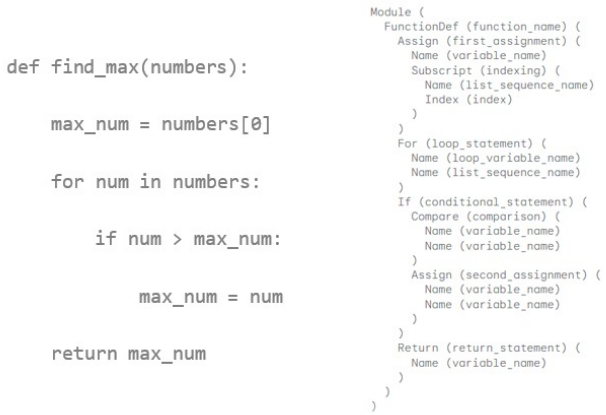

Figure 2: Representación gráfica de un árbol de sintaxis.

2. **Filtrado de nodos**.- Hay algunos nodos que no tienen un valor significativo en la implementación de la solución. Por ejemplo, hay un nodo que representa al módulo del código, y dado que solo se considera un archivo para realizar la comparación, éste puede ser ignorado. La Figura 3 muestra la lista de los nodos de un árbol de sintaxis ya filtrado.

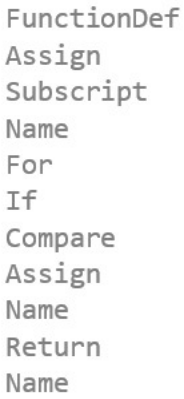

Figure 3: Lista de nodos de un árbol de sintaxis filtrado.

3. **Agrupamiento y hashing k-grams**.- K-grams es una secuencia de caracteres o palabras consecutivas en un documento, en cuanto a la solución. La propuesta busca utilizar k-grams de nodos consecutivos, en lugar de caracteres simples.

   El número de nodos consecutivos, representado por la variable $\boldsymbol{k}$, debe ser lo suficientemente grande para evitar falsos positivos, posteriormente, en la sección de experimentación, se determinó que utilizar 3-5 nodos por cada secuencia proveé detecciones adecuadas a lo que se busca detectar. El fingerprint del código es representado por un subconjunto de los hashes

generados por los k-grams, de forma que, cada porción del código, representada por un k-gram, será transformada en un hash.

Una vez generada la lista de k-grams, cada k-gram pasa por un proceso de hashing. Se utiliza el algoritmo md5 para realizar la operación de hash, ya que, nos otorga un valor de 128 bit, a comparación de otros algoritmos que nos generan una secuencia más larga. Dado que se cuenta con un conjunto finito de k-grams, cuyo tamaño no es muy grande, se puede afirmar que las colisiones por la generación del mismo hash por dos k-gram distintos es cercana a cero.

La Figura 4 muestra los k-grams generados y el hash correspondiente a cada k-gram, en este caso, tomando en cuenta un valor de 5 nodos consecutivos.

```
[Import, FunctionDef, Assign, Subscript, Name]  -   053781c386a12083d21200be8a352044
[FunctionDef, Assign, Subscript, Name, For]     -   e19e8f77d21ca1ec79f17c8b08ed4188
[Assign, Subscript, Name, For, If]              -   195813fb911669b775f10ce416e9ef19
[Subscript, Name, For, If, Compare]             -   59b3bd28141001684f4ec655ca2af810
[Name, For, If, Compare, Assign]                -   b66e65820aac2a0bee7210357d63aa27
[For, If, Compare, Assign, Name]                -   8fcec2be7cd88942734b6db3eb66ffa9
[If, Compare, Assign, Name, Return]             -   181b54ed5f811443c19e0e19960d1257
[Compare, Assign, Name, Return, Name]           -   b85779a848a6c1e53335581b93f8e030
```

Figure 4: K-grams y su respectivo hash.

4. **Generación fingerprint**.- Para la generación del fingerprint, se toma como base la lista de hashes de los k-grams (Figura 4), a la cual se le aplica el algoritmo de winnowing, el cual consiste en agrupar los hashes en grupos de ventanas de tamaño $\boldsymbol{w}$. Definiendo un valor $\boldsymbol{t}$, que representa un umbral, se puede determinar el tamaño de la ventana $\boldsymbol{w}$ mediante w = t - k + 1. Contando con un valor de t más grande, las ventanas del algoritmo de winnowing serán más grandes, aumentando la sensibilidad de encontrar hashes similares.

   Por cada ventana generada, se procede a guardar el hash con menor valor en un arregló que resultará en el fingerprint. Antes de guardar el hash en el fingerprint se verifica que el mismo hash con la misma posición no se encuentre previamente en el fingerprint. Puede haber casos donde exista el mismo hash pero en diferentes posiciones, por lo que es considerado como un elemento distinto del fingerprint. La Figura 5 muestra el fingerprint de un código.

```
[
    053781c386a12083d21200be8a352044 {'lineno': 0, 'end_lineno': 3, 'col_offset': 0, 'end_col_offset': 21},
    195813fb911669b775f10ce416e9ef19 {'lineno': 3, 'end_lineno': 6, 'col_offset': 4, 'end_col_offset': 25},
    59b3bd28141001684f4ec655ca2af810 {'lineno': 3, 'end_lineno': 5, 'col_offset': 14, 'end_col_offset': 24},
    181b54ed5f811443c19e0e19960d1257 {'lineno': 5, 'end_lineno': 7, 'col_offset': 8, 'end_col_offset': 18},
]
```

Figure 5: Fingerprint generado de un código.

Para poder encontrar similitudes entre dos archivos, es necesario tener sus respectivos fingerprints. La Figura 6 muestra cual es el proceso para obtener el porcentaje de similitud y coincidencias,

teniendo como entrada dos fingerprints.

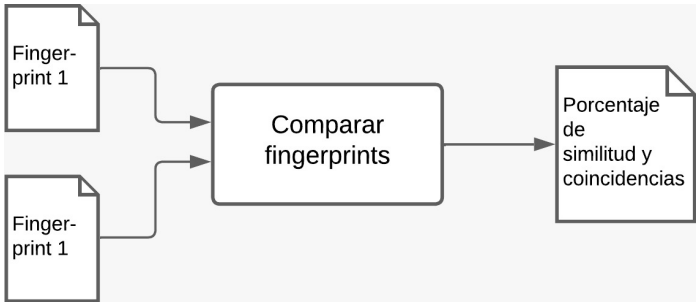

Figure 6: Pseudocódigo de la propuesta de solución.

La comparación consiste en utilizar el índice Jaccard [**8**], el cual es una métrica empleada para medir la similitud y la diversidad de conjuntos de datos. Para calcular el índice, se divide el tamaño de la intersección de los conjuntos por el tamaño de la unión de los conjuntos. Utilizando el índice Jaccard, es posible obtener un porcentaje de similitud entre dos fingerprints.

Adicionalmente, cada elemento del fingerprint contiene la posición de dicho hash, por lo que es posible obtener que hashes se encuentran en cada archivo, así como su posición en cada uno.

La Figura 7 muestra un pseudocódigo de la comparación de dos archivos, obteniendo como resultado el porcentaje de similitud entre ambos códigos.

```
def get_similarity(file_1, file_2):
    def generate_fingerprint(file):
        ast = get_ast(file)
        ast_nodes = get_children(ast)
        kgrams = generate_kgrams(ast_nodes)
        hashed_kgrams = hash_kgrams(kgrams)
        return winnowing(hashed_kgrams)

    # Generate fingerprints for both files
    fingerprint_1 = generate_fingerprint(file_1)
    fingerprint_2 = generate_fingerprint(file_2)

    # Return the similarity between the fingerprints
    return match_fingerprints (fingerprint_1, fingerprint_2)
```

Figure 7: Pseudocódigo de la propuesta de solución.

La solución propuesta es capaz de detectar cambios estéticos, como lo puede ser el cambio del nombre de variables o funciones, así como la reubicación de algunas partes de código, que solamente buscan dar la apariencia de que se trata de un código nuevo, cuando no lo es. Esto es posible por el preprocesamiento que hace en la generación y filtrado del árbol de sintaxis, y por el hecho de guardar la posición de cada hash del fingerprint.

Casos en donde la solución no podría resultar eficiente se destacan por un cambio radical del código, que va más allá de la estética. Cambios en la sintaxis del código son imposibles de detectar,

puesto que lo propuesto se basa en la agrupación de nodos de un árbol de sintaxis y, si los nodos cambian excesivamente, las agrupaciones serían distintas. Un ejemplo claro sería el cambio de una función recursiva por una iterativa, ya que, si bien realizan la misma función, utilizan distintas estructuras de control, ciclos, llamadas a funciones, etc. Una forma de resolver este inconveniente sería utilizando un grafo de dependencia, pero iría fuera del objetivo propuesto por este trabajo, que es el detectar código meramente estético.

4 Experimentación

Para experimentar e identificar la precisión del método presentado, y por lo tanto la efectividad del algoritmo, se llevó a cabo un experimento que consiste en una comparativa de la solución presentada con respecto a una metodología existente. Se considera y asume que implementaciones existentes en el mercado ya han realizado una experimentación exhaustiva sobre su efectividad y por lo tanto, presentan una base robusta para ser objetos a una comparación. Para elegir las posibles herramientas para la comparativa se tomó en cuenta que la herramienta utilice una heurística similar a la solución propuesta para la detección de similitudes en código fuente. El método propuesto utiliza el concepto de generación de huellas digitales a partir del análisis y extracción de la representación sintáctica del código a través del árbol de sintaxis, k-grams para la generación de secuencias continuas de nodos y el algoritmo de winnowing para selección de la huella única del documento. En base a esto metodologías como BPlag [**10**], el cual implementa un análisis simbólico de las dependencias en el código, fueron descartados y quedan fuera de alcance del método propuesto en este artículo.

Para esta evaluación, fueron identificados dos herramientas para su uso en la experimentación: MOSS[**11**] y DOLOS [**12**]. Sin embargo, MOSS mostró dificultades para su configuración debido a que no cuenta con un método de instalación local, y la interfaz web presentó fallas continuas durante el proceso de experimentación, por lo que se decidió continuar únicamente con DOLOS.

DOLOS [https://dolos.ugent.be/about/algorithm.html] funciona mediante un proceso de tokenización, fingerprinting, indexado y reporte. En primer lugar, se extraen los tokens utilizando la librería de TreeSitter, de esta manera el sistema evita técnicas de plagiarismo simples como el renombramiento de variables y funciones. Posteriormente, DOLOS intenta encontrar secuencias comunes de tokens con k-grams y un algoritmo de rolling hashing para optimizar el proceso de hashing, y el algoritmo de winnowing para utilizar un subconjunto de hashes y mejorar el consumo de memoria.

Adicionalmente, DOLOS permite una instalación local del sistema a partir de un repositorio de Github, en la forma de una aplicación web completa para el indexado y visualización del análisis de similitud. El sistema adicionalmente reporta métricas que incluyen el índice de similitud, superposición de fragmentos y el fragmento más largo en común entre un conjunto de documentos analizados.

4.1 Experimento comparativa DOLOS

El propósito de este experimento consiste en evaluar el programa al utilizar un conjunto de programas de Python y observar si el método propuesta es capaz de obtener una similitud similar al obtenido

por DOLOS.

Se utilizó un conjunto de códigos de python proveniente de entregas en 3, 000 preguntas de la plataforma Codeforces [**7**], principalmente de los concursos realizados entre los años 2020-2023. De manera aleatoria, se tomó una muestra de 41 archivos para la evaluación a partir del conjunto de 3 preguntas diferentes para garantizar que ciertas entregas fueran similares y garantizar una muestra diversa entre entregas y evaluar si el sistema es capaz de identificar cuando dos códigos son completamente diferentes, y cuando son muy similares.

4.2 Método

En primer lugar, el dataset original cuenta con archivos en diferentes lenguajes de programación (Python, C++, Java). Por lo tanto , se realizó un filtrado, de manera manual, de todos los archivos, de forma que únicamente se mantuvieran aquellos archivos del lenguaje de programación python. Posteriormente, se escogió de manera aleatoria un subset de archivos, debido a la amplia cantidad de archivos en el dataset. Finalmente, se realizó un preformateo de los archivos, en este caso, formateando los caracteres de CRLF a LF, con el fin de mantener integridad y evitar problemas al momento de utilizar los archivos en esta etapa de experimentación.

A continuación, los códigos seleccionados fueron introducidos a la DOLOS para su análisis. Para cada documento, se consideró únicamente el documento con el mayor índice de similitud. De esta forma, se evalúa si el sistema es capaz de identificar el mismo documento con la más alta similitud, y cual es el porcentaje de similitud obtenido.

Como puntos de comparación, se consideraron 2 métricas distintas para validar la propuesta de solución contra DOLOS: la precisión medida en base al número de documentos clasificados correctamente como el documento con mayor similitud, y el error absoluto medio, el cual mide la diferencia promedio en el porcentaje de similitud obtenido en archivos clasificados correctamente.

En adición a esto, para la experimentación de la solución, se emplearon dos hiperparámetros distintos para modificar el comportamiento del algoritmo. El primero, $\boldsymbol{k}$, determina la longitud de la secuencia de cada k-gram, entre más pequeño este parámetro, la detección de plagio es más sensible. El segundo, $\boldsymbol{t}$, el cual representa el límite que determina la longitud del fingerprint.

4.3 Resultados

La Figura 8 compara las métricas de exactitud y error absoluto medio para diferentes valores el los hiper parámetros de $\boldsymbol{k}$ y $\boldsymbol{t}$. De las 3 pruebas realizadas, se obtuvo una mayor exactitud con menor error absoluto medio al utilizar los valores k = 5, t = 5.

	k = 3 t = 5	k = 5 t = 5	k = 5 t = 7
Exactitud	65.85	73.17	73.17
Error absoluto medio	6.74	6.36	7.26

Figure 8: Resultados de la experimentación con diferentes valores de parámetros.

Lo que corresponde a 30 códigos clasificados correctamente de la muestra de 41 documentos. Un error absoluto de 6 puntos porcentuales indica que la implementación propuesta ofrece un porcentaje de similitud similar al obtenido por DOLOS, para documentos clasificados correctamente.

Los valores óptimos para los hiper parámetros con la literatura en [3], ya que un número pequeño para el número de nodos consecutivos en un k-gram puede ocasionar la introducción de ruido en la detección de secuencias similares (argumentos en la definición de una función, i.e) No obstante, un valor alto para el umbral t puede afectar en la capacidad de detectar cambios cosméticos (declaración de una función en bloque a diferencia de declarar el procedimiento afuera de una). Por lo tanto, un valor intermedio de los parámetros resulta ser la opción más óptima.

La Figura 9 muestra la clasificación de los documentos, los cuales están agrupados de acuerdo al porcentaje de similitud obtenido. A todo aquel documento al que se le asigne una documento similar diferente al obtenido por la plataforma de DOLOS, será considerado como una clasificación incorrecta, mientras que, a todo aquel documento al que se le asigne una similitud al mismo documento al obtenido por la plataforma DOLOS, será considerado como una clasificación correcta.

Como se observa en la figura, códigos con un porcentaje de similitud alto (mayor a 60%) tiende a ser clasificado correctamente por el sistema. Por otro lado, la clasificación decrementa en archivos con un índice de similitud bajo. Esto sugiere que el método propuesto es efectivo cuando existe una similitud alta entre los documentos

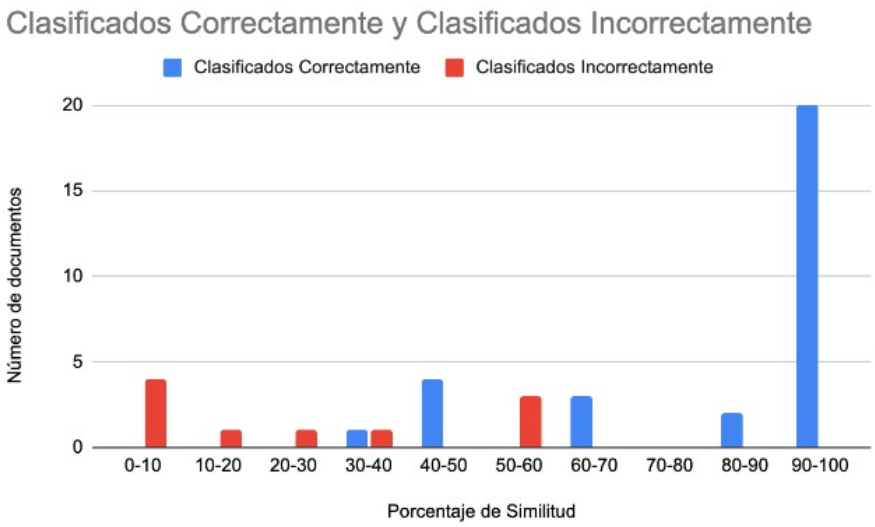

Figure 9: Relación entre número de documentos clasificados correctamente e incorrectamente

La diferencia de efectividad en base al porcentaje de similitud se basa principalmente en la técnica utilizada para la detección de similitud. Si bien la generación de fingerprint en base a la representación del árbol de sintaxis permite un evaluación robusta en cuanto a similitudes cosméticas, cuando el código presenta alteraciones a nivel semántico, o no presenta ninguna similitud, el programa puede detectar falsos positivos debido a que el código mantiene una estructura sintáctica similar. Implementaciones como análisis de grafos de dependencias [**9**] o análisis simbólico [**10**],

podrían identificar este tipo de similitudes con mayor precisión, pero dado a que estos tipos de análisis quedan fuera del alcance establecido para esta investigación, la implementación propuesta presenta una buena técnica para la identificación de similitudes en la mayoría de los casos.

4.4 Experimentación interfaz web

El análisis de similitud entre códigos fuente para la detección de plagio aún requiere de la interacción humana para poder determinar casos de plagio, dependiendo del contexto en el que se analiza [**4**]. Tomando en cuenta esto, se realizó una experimentación para mostrar los resultados obtenidos por el método de similitud propuesto a través de una interfaz web interactiva con el nombre de Pylens.

Para este programa, la información posicional de los k-grams (número de línea y columna en el documento original) es almacenada para cada fingerprint de los documentos a analizar, para su posterior visualización en la interfaz gráfica. Pylens puede ser instalado de manera local a través de un repositorio en github para el backend del sistema [https://github.com/JAngelGC/code_detector], así como el frontend [https://github.com/Carlos24Rz/code-similiary-ui], con instrucciones para su instalación. Adicionalmente, se requiere una configuración con los servicios de Google Firebase para el indexado y almacenamiento de la información por parte de los usuarios. La figura 10 muestra la arquitectura utilizada por el sistema para la comunicación entre el cliente y el servidor.

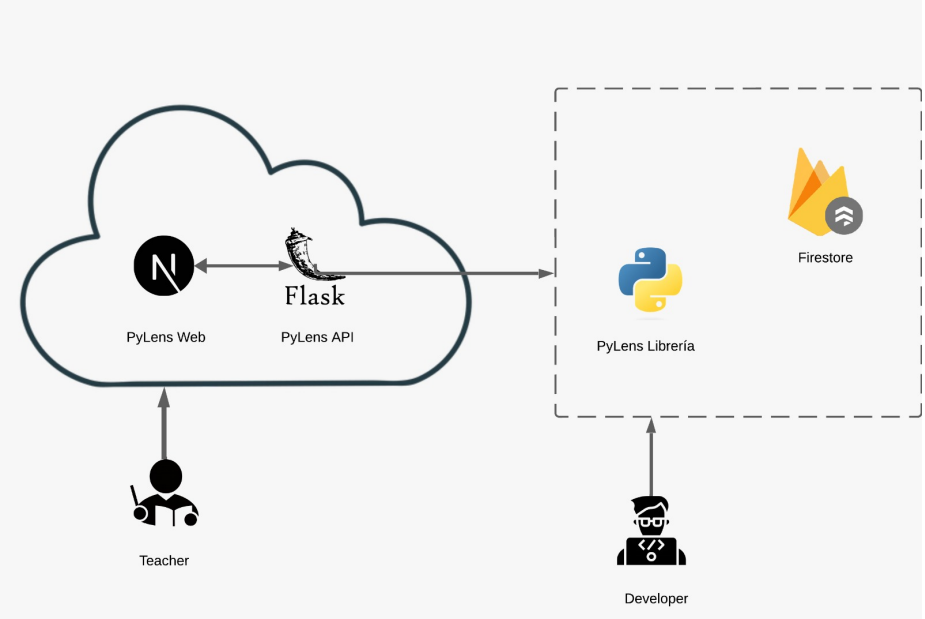

Figure 10: Arquitectura cliente-servidor de PyLens.

Para el análisis de similitud, la plataforma presenta una visualización a 2 vistas entre un par de los documentos análisis, para mostrar las secciones similares en base a la información posicional de los k-grams de los fingerprints indexados anteriormente. La figura 11 muestra la representación de la comparativa entre 2 documentos.

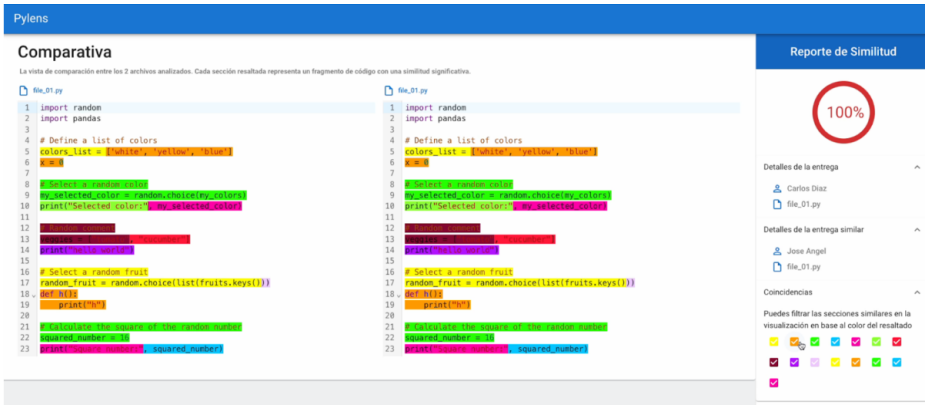

Figure 11: Interfaz comparativa en Pylens.

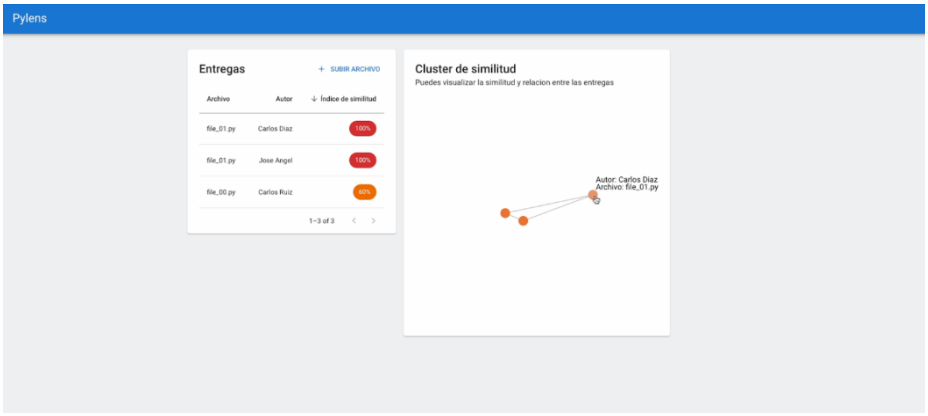

Figure 12: Dashboard Comparativa entre entregas Pylens.

Así mismo como se observa en la figura 12, la interfaz ofrece un dashboard para la visualización de entregas indexadas en el sistema y observar el índice de similitud entre ellas, a través de un grafo dirigido por fuerzas, en el que los nodos representan las entregas en el sistema y las aristas representan el porcentaje de similitud entre ellas, por lo que una similitud mayor entre dos archivos se visualiza con una distancia menor entre los vértices.

5 Conclusiones

Se ha presentado una propuesta de solución que genera un índice de similitud entre dos códigos distintos, la cual está basada en la comparación de fingerprints generadas por medio de los nodos de un árbol de sintaxis. No obstante, es importante hacer énfasis en la necesidad de interacción humana para que la propuesta de implementación sea factible, puesto que aún se requiere del juicio humano, en este caso, el profesor, para que este sea el encargado final de determinar si efectivamente ha existido un proceso de plagio, tomando en cuenta el índice de similitud obtenido por el algoritmo.

Aunado a ello, es importante recalcar que la solución va más allá de la propuesta e implementación

de un algoritmo, puesto que también se cuenta con una interfaz gráfica de usuario, la cual facilita la interpretación de las similitudes encontradas, resultando en una mejor experiencia de usuario, volviendo el proceso más ágil y evitando que el usuario tenga que directamente interactuar con la implementación del algoritmo.

En cuanto a la propuesta, para llegar al algoritmo propuesto, se analizaron diferentes enfoques para encontrar similitudes entre código, teniendo como desenlace una solución fundamentada en el algoritmo de Winnowing en conjunto con la técnica de k-grams, los cuales han mostrado su eficacia y eficiencia para la detección de similitudes. Adicionalmente, se agregó una parte de procesamiento del árbol de sintaxis que permite ampliar el rango para encontrar similitudes, lo cual le otorga una plusvalía a la propuesta, ocasionando un mayor rango de alcance en la detección de similitudes entre códigos.

A través de los experimentos realizados, se ha demostrado que la solución propuesta tiene valores cercanos de exactitud y precisión con plataformas verificadas, como es el caso de Dolos, lo cual garantiza la fiabilidad de la solución y permite contar con un punto de referencia en cuanto a la eficacia de la misma.

Finalmente, como forma de retroalimentación para trabajo o mejora a futuro, existen áreas de oportunidad en la propuesta, por ejemplo, se puede ampliar la fase en la que se obtienen los nodos del árbol de sintaxis, puesto que, al ser Python un lenguaje considerablemente grande, se tuvo que limitar el procesamiento de algunos nodos, reduciendo la capacidad del algoritmo de encontrar plagio en algunas situaciones, por lo que existen casos inexplorados por la propuesta, lo cual podría ser una expansión a la propuesta que permitiría una mejor detección en la similitud de códigos para la detección de plagio.

References

[1] Andrianov, I., Rzheutskaya, S., Sukonschikov, A., Kochkin, D., Shvetsov, A., Sorokin, A. (2020). Duplicate and Plagiarism Search in Program Code Using Suffix Trees over Compiled Code. 26th Conference of Open Innovations Association (FRUCT), pp. 1-7,. https://doi.org/10.23919/fruct48808.2020.9087465

[2] Wang, H., Zhong, J., Zhang, D. (2015). A Duplicate Code Checking Algorithm for the Programming Experiment. 2015 Second International Conference on Mathematics and Computers in Sciences and in Industry (MCSI).pp. 39-42, doi:10.1109/mcsi.2015.12

[3] Sutoyo, R., Ramadhani, I., Ardiatma, A. D., Bavana, S. C., Warnars, H. L. H. S., Trisetyarso, A., ... Suparta, W. (2017). Detecting documents plagiarism using winnowing algorithm and k-gram method. IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom). pp. 67-72, doi:10.1109/cyberneticscom.2017.8311686

[4] M. Joy and M. Luck, (1999). "Plagiarism in programming assignments," in IEEE Transactions on Education, vol. 42, no. 2, pp. 129-133, doi: 10.1109/13.762946.

[5] Marcos, M., Negrete, M. Q., Arias, J., Quispe, I. M., Carranza, C. M. (2023). Plagio académico: Comprender, prevenir y abordar. Instituto Universitario de Innovación Ciencia y Tecnología Inudi Perú eBooks. https://doi.org/10.35622/inudi.b.099

[6] Shah, Jay N Shah, Jenifei Baral, Gehanath Baral, Reetu Shah, Jesifei. (2022). Types of plagiarism and how to avoid misconduct: Pros and cons of plagiarism detection tools in research writing and publication. Nepal Journal of Obstetrics and Gynaecology. 16. 3-18. 10.3126/njog.v16i2.42085.

[7] Codeforces source code submissions dataset. (2023). Kaggle. https://www.kaggle.com/datasets/yeoyunsianggeremie/codeforces-code-dataset

[8] Costa, L. D. F. (2021). Further generalizations of the Jaccard index. arXiv preprint arXiv:2110.09619.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. (1987). The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9, 3 319–349. https://doi.org/10.1145/24039.24041

[10] H. Cheers, Y. Lin and S. P. Smith. (2021). "Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity," in IEEE Access, vol. 9, pp. 50391-50412,, doi: 10.1109/ACCESS.2021.3069367.

[11] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. (2003). Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data . Association for Computing Machinery, New York, NY, USA, 76–85. https://doi.org/10.1145/872757.872770

[12] Rien Maertens, Maarten Van Neyghem, Maxiem Geldhof, Charlotte Van Petegem, Niko Strijbol, Peter Dawyndt, Bart Mesuere. (2024). Discovering and exploring cases of educational source code plagiarism with Dolos, SoftwareX, Volume 26,101755, ISSN 2352-7110, https://doi.org/10.1016/j.softx.2024.101755.