Departamento de Ciencias de la Computación
Escuela de Ingeniería y Ciencias

Reporte Técnico para TC3002B

Detección de Plagio Tipo I, II y III en Código Python Usando Suffix Array, AST y Machine Learning

---

Profesores:
Dr. Cesar Torres Huitzil
Dr. Luciano García Bañuelos
Dr. Daniel Pérez Rojas
Candy Yuridiana Alemán Muñoz

Miembros del Equipo:
Alejandro Daniel Moctezuma Cruz (A01736353)
Augusto Gómez Maxil (A01736346)
José Juan Irene Cervantes (A01736671)
Jacqueline Villa Asencio (A01736339)

Mayo 28, 2025

Resumen

El incremento en el uso de la Inteligencia Artificial y los repositorios colaborativos han facilitado la generación de contenido ilícito, siendo un buen ejemplo el plagio de código fuente en entornos educativos. Si bien actualmente existen herramientas como Measure of Software Similarity (MOSS) y JPLAG para la detección de plagio, estas presentan limitaciones ante técnicas como el renombramiento y modificación estructural.

En este trabajo se presenta un algoritmo en lenguaje Python para un sistema de detección de plagio de código fuente en Python enfocado en identificar plagio de tipo I (exacto), II (renombramiento) y III (modificado), combinando técnicas de análisis léxico mediante Suffix Array, representación estructural mediante Árboles de Sintaxis Abstracta (AST) y clasificación automática mendieta un modelo de aprendizaje automático supervisado Random Forest.

El sistema implementa una aplicación de escritorio con interfaz interactiva, permitiendo al usuario comparar códigos y visualizar tanto el tipo de plagio detectado como las métricas de similitud. Los resultados muestran una mejora significativa en la precisión a comparación de los instrumentos tradicionales, ofreciendo una herramienta efectiva para docentes del Tecnológico de Monterrey, Campus Puebla, en la promoción de la integridad académica.

Contenidos

1 Introducción

El auge de la inteligencia artificial y los repositorios colaborativos han transformado diversos sectores, incluida la educación, facilitando la generación de contenido ilícito, como es el plagio de código fuente entre estudiantes. Si bien esta herramienta ha aportado significativamente al avance tecnológico, el gran flujo de material a la vez ha dificultado la implementación de técnicas que puedan seguir el ritmo del mismo y además identifiquen las diversas técnicas que se utilizan para evadir la detección de plagio, aún si ya existen instrumentos como Measure of Software Similarity (MOSS) o JPlag para detectar similitudes entre códigos, muchas de estas presentan limitaciones ante técnicas de evasión como el renombramiento de variables y/o funciones o la reestructuración sintáctica del código.

Además, el uso de inteligencia artificial para generar código puede facilitar la violación de los derechos de autor, lo cual es un aspecto importante en disciplinas como las ingenierías, donde el software representa un elemento importante. Sin embargo, es importante no sesgar a esta tecnología en un contexto negativo, ya que esta misma puede emplearse de manera positiva y eficiente para la detección de similitud entre los contenidos con mayor precisión. Con la implementación de sistemas de detección automática, se logra crear un entorno académico donde los estudiantes realicen trabajos auténticos, fomentando así la integridad académica, disuadiendo el plagio y promoviendo una cultura de honestidad y responsabilidad en la formación universitaria.

En este contexto, se propone el desarrollo de un sistema de detección de plagio para código fuente escrito en Python. El enfoque se centra en la identificación de tres tipos principales de este: plagio exacto (tipo I), renombramiento (tipo II) y modificaciones estructurales (tipo III). Para ello, se combinan técnicas de análisis léxico mediante el algoritmo Suffix Array por medio de la comparación de tokens, la representación estructural por medio de Árboles de Sintaxis Abstracta (AST) y clasificación automática usando el algoritmo de aprendizaje automático por medio de Random Forest multietiqueta. La solución se implementa en una aplicación con interfaz interactiva que permite a los docentes comparar los códigos de los estudiantes al visualizar el tipo de plagio detectado y acceder a métricas detalladas de similitud.

El desarrollo incluye la muestra de métricas específicas para evaluar la similitud léxica por medio de Longest Common Subsequence (LCS), la similitud estructural utilizando tanto Tree Edit Distance (TED) como la extracción de características estructurales, y un puntaje combinado con respecto a los resultados de similitud extraídos por dichas métricas para una evaluación más completa. Para la validación del desempeño del modelo, se implementan métricas estándar de clasificación como precisión, recuperación (recall), F1-score y exactitud (accuracy).

El presente trabajo busca proporcionar una herramienta práctica y precisa para el profesorado del Tecnológico de Monterrey, Campus Puebla, que les permita detectar la similitud entre entregas de estudiantes de forma automatizada, eficiente y rápida. Si bien en esta

etapa se excluye la detección de plagio semántico (tipo IV) por su complejidad técnica, se mencionan las bases para futuras mejoras que integren modelos semánticos más avanzados para una detección de plagio más completa y robusta.

Este documento se estructura de la siguiente manera: en la Sección 2 se presenta una revisión del estado del arte, analizando enfoques previos para la detección de plagio en código fuente. En la sección 3 se describe detalladamente la solución propuesta, incluyendo la arquitectura del sistema, los módulos de comparación léxica y estructural, así como el clasificador basado en aprendizaje automático. La Sección 4 expone los resultados experimentales obtenidos, evaluando el desempeño del sistema con distintos tipos de plagio y conjuntos de datos. En la Sección 5 se discuten e interpretan dichos resultados, comparándolos con trabajos relacionados. Finalmente, en la Sección 6 se presentan las conclusiones y se plantean posibles líneas de trabajo futuro.

2 Estado del Arte

El problema del plagio siempre se ha visto presente en entornos escolares, existiendo una variedad de propuestas que buscan dar solución al problema de la detección de este dentro de códigos fuente. Se han propuesto diversos enfoques de acuerdo a los tipos de plagio detectados que van desde la comparación de texto plano hasta el uso de la inteligencia artificial para medir los porcentajes de similitud entre los archivos. A continuación, se muestra una revisión de tres artículos relacionados con la problemática a resolver.

Primeramente, Nandini Gandhi, Kaushik Gopalan y Prajish Prasad (AÑO) en su artículo *"A Support Vector Machine based approach for plagiarism detection in Python code submissions in undergraduate settings"*, implementan un modelo Symbolic Model Verifier (SMV) y una combinación de análisis textual y sintáctico por medio de Árboles de Sintaxis Abstracta, detectando así la similitud entre los códigos fuente de Python de los estudiantes. Dicha solución es evaluada contra herramientas como Measure of Software Similarity (MOSS) y JPlag, las cuales son superadas al mostrar una mejora en la exactitud, recall y precisión, siendo esta de un 97.3%. Las ventajas de la propuesta radican en la especialización en el lenguaje de programación Python, combinando la sintaxis general, estructura vertical, funciones, variables, comentarios, strings y bucles, sin embargo, el modelo aún produce falsos positivos (detecta el plagio en dos códigos al dar soluciones a problemas parecidos) y falsos negativos (no detecta el plagio por ofuscación, es decir, no toma en cuenta el renombrado de variables y el cambio de orden de instrucciones, etc., realizadas para evitar la detección).

Por otro lado, Raddam Sami Mehsen, Majharoddin M. Kazi y Hiren Joshi, en *"Detecting Source Code Plagiarism in Student Assignment Submissions Using Clustering Techniques"*, utilizan un sistema TF-IDF, en contraposición a un sistema Measure of Software Similarity (MOSS), JPlag o GPlag, para representar los archivos como vectores de características, aplicando posteriormente técnicas de agrupamiento, o *clustering*, para la detección de plagio de

código fuente de estudiantes. Las ventajas de este modelo es que, además de no basarse en coincidencias y usar características sintácticas, realiza la clasificación con algoritmos supervisados que no requieren datos etiquetados, K-means y *clustering* jerárquico. Otra ventaja es el uso de los lenguajes de programación: Python para las librerías; Python y C++ para los conjuntos de datos. De esta forma, se obtiene un 99.5% de exactitud, en base a 880 entregas de estudiantes. No obstante, en esta solución no se proponen modelos de detección basados en análisis semántico profundo, estructuras abstractas como Árboles de Sintaxis Abstracta (AST) o grafos de dependencia, los cuales permiten una comprensión más contextual del código fuente y ofrecen mayor robustez frente a técnicas avanzadas de ocultamiento de plagio.

Añadiendo a los artículos anteriores, tenemos *"Detección de plagio en código fuente Java mediante tokenización y aprendizaje de máquina",* escrito por Misael Fernando Perilla Benitez, que presenta un sistema para detectar plagio en código Java mediante tokenización con ANTLR (Another Tool for Language Recognition) y clasificación con aprendizaje automático (SVM, usando Weka). Aborda el creciente problema del plagio en entornos académicos, proponiendo una herramienta que automatiza la revisión de código con una precisión del 63.33% en pruebas, superando la revisión manual en velocidad. Entre sus ventajas destacan la facilidad de uso, la integración con herramientas como NetBeans y Weka, y su enfoque en análisis léxico-sintáctico que permite detectar plagios aún con cambios superficiales. Sin embargo, presenta limitaciones importantes: sólo funciona con código Java SE 8 sin mezclas de otros lenguajes, requiere un corpus más grande para mejorar su precisión, y depende del entrenamiento previo. Su implementación demuestra que es posible construir un sistema funcional y rápido, aunque su efectividad podría mejorar con mayor robustez y extensión a otros lenguajes y tipos de código. La propuesta representa un aporte relevante para combatir el plagio en la enseñanza de programación universitaria.

Los tres enfoques revisados muestran aportaciones relevantes en la detección de plagio de código fuente, cada uno con distintas fortalezas y debilidades, sin embargo, fue el trabajo de Ghandi et al. titulado *"A Support Vector Machine based approach for plagiarism detection in Python code submissions in undergraduate settings"*, siendo este el que se asemeja más al presente proyecto, ya que combina el análisis léxico y estructural mediante Árboles de Sintaxis Abstracta (AST), lo cual permite detectar los tipos de plagio yendo más allá de la copia exacta en los códigos, aunque presenta limitaciones ante técnicas de ofuscación. Por su parte, la propuesta de Mehsen et al. destaca por su uso de clustering sin requerir etiquetas, pero se limita a los enfoques sintácticos sin utilizar estructuras profundas como árboles o grafos. Finalmente, el sistema de Perilla Benitez aplica tokenización y clasificación con aprendizaje automático (SVM) sobre código Java, mostrando buenos resultados, aunque restringido a un lenguaje específico con menor precisión. En contraste, nuestra propuesta integra un enfoque híbrido que unifica el análisis léxico usando Suffix Array, el análisis estructural implementando Árboles de Sintaxis Abstracta (AST) y clasificación automática con K-Nearest Neighbors (KNN), buscando superar las limitaciones de los modelos anteriores mediante una arquitectura más robusta, orientada específicamente a lenguaje Python y al entorno académico universitario.

3 Solución Propuesta

Antes de iniciar con la solución propuesta, queremos explicar los tipos de plagio que fueron considerados dentro del alcance de trabajo, así como los retos particulares que cada uno implica en su detección.

A partir del análisis de los enfoques presentados en el estado del arte, que abordan la detección de plagio desde perspectivas léxicas, estructurales o estadísticas, se identificó la necesidad de una solución más integral y adaptable a los diversos escenarios presentes en contextos educativos reales. En este sentido, el sistema propuesto se basa en una arquitectura híbrida que combina técnicas complementarias para enfrentar tres de los tipos de plagio más comunes en código fuente:

- **Plagio Tipo I (Exacto).** Ocurre cuando fragmentos de código son copiados íntegramente, permitiendo únicamente variaciones triviales como cambios en los espacios en blanco o comentarios. Es el tipo más sencillo de detectar, debido a que las coincidencias pueden identificarse mediante comparaciones textuales o contextualizadas sin necesidad de análisis estructural profundo. Sin embargo, su prevalencia sigue siendo alta en contextos educativos, donde los estudiantes copian soluciones de forma literal.
- **Plagio Tipo II (Renombramiento).** Ocurre cuando el plagiador mantiene la estructura lógica del código original, pero realiza modificaciones superficiales como el cambio de nombres de variables, funciones, tipos de datos o estilos de codificación. Aunque es más fácil detectar que el plagio exacto, las técnicas basadas en tokenización normalizada y análisis léxico permiten identificar patrones estructurales equivalentes.
- **Plagio Tipo III (Modificado).** Implica alteraciones más profundas al código original, incluyendo inserciones, eliminaciones o reordenaciones de líneas de código. Estas modificaciones buscan ocultar la copia manteniendo la funcionalidad base. Para su detección, se requieren técnicas más robustas como el análisis de Árboles de Sintaxis Abstracta (AST) y la comparación de distancias de edición estructural (Tree Edit Distance). Este tipo de plagio representa un reto intermedio entre la simplicidad del renombramiento y la complejidad semántica.

En el contexto del presente proyecto, el plagio semántico (Tipo IV) que se centra en la misma funcionalidad pero con diferencias textuales, queda fuera del alcance en esta etapa, debido a la complejidad técnica y computacional que conlleva. Esta elección se alinea con las recomendaciones de la literatura, que sugiere abordar los tipos de plagio en fases, comenzando por aquellos con mayor prevalencia y menos dificultad técnica.

3.1 Arquitectura General

La **Figura 1** presenta la arquitectura general del sistema, ilustrando el flujo principal de procesamiento aplicado a los archivos fuente.

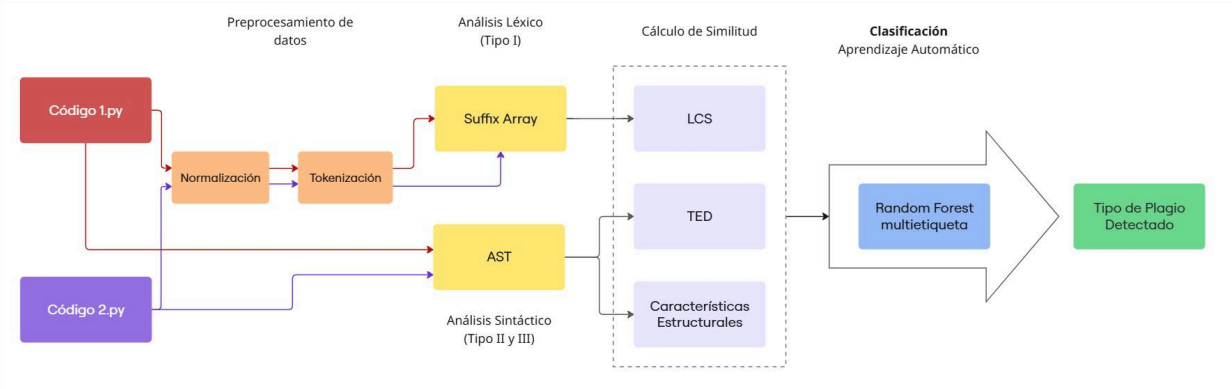

Figura 1. Esquema general de funcionamiento del sistema para detección de plagio

El sistema se compone de tres bloques funcionales interconectados: un módulo de preprocesamiento de código fuente, un conjunto de comparadores léxico y estructural, y un clasificador basado en aprendizaje automático. Cada uno de estos bloques desempeña un papel específico dentro del flujo general de análisis.

El **primer bloque** corresponde al preprocesamiento del código fuente. En esta etapa, el sistema segmenta el contenido en unidades léxicas y aplica transformaciones sobre aquellas estructuras susceptibles de variación superficial. Este procedimiento —que unifica la tokenización y la normalización en un solo paso— permite obtener una representación intermedia uniforme, que será utilizada en los análisis posteriores. El propósito de este bloque es eliminar diferencias cosméticas entre fragmentos de código que comparten una lógica funcional común, facilitando así una comparación más precisa y objetiva.

El **segundo bloque** consiste en dos rutas de análisis independientes pero complementarias:

- Por un lado, el análisis léxico trabaja sobre la secuencia de tokens ya normalizados, aplicando el algoritmo de Suffix Array para detectar subcadenas comunes entre los archivos. Este componente es especialmente efectivo en la identificación de plagio de tipo I (copia exacta) y tipo II (renombramiento de identificadores).
- Por otro lado, el análisis estructural transforma cada archivo fuente en un Árbol de Sintaxis Abstracta (AST), una estructura jerárquica que refleja la lógica del programa. A partir de esta representación se extraen características como número de nodos, profundidad máxima, orden de funciones y presencia de estructuras condicionales o cíclicas. Estas métricas permiten detectar similitud estructural, característica del plagio tipo III, asociado a la reorganización de bloques de código.

El **tercer bloque** corresponde al proceso de clasificación. Una vez calculadas las métricas provenientes de ambas rutas, estas se integran en un vector de características que alimenta un modelo de clasificación multietiqueta. Este modelo, basado en algoritmos de aprendizaje supervisado, ha sido entrenado con ejemplos etiquetados de código sin plagio (tipo 0) y con plagio de tipo I, II y III. Su función es predecir con precisión los tipos de plagio presentes en una nueva comparación, incluso en casos donde coexisten diferentes formas de manipulación del código.

Este enfoque permite detectar plagio mixto en una misma entrega, como cuando se combinan renombramiento de identificadores con cambios estructurales. La integración de rutas léxicas y estructurales en paralelo mejora la sensibilidad del sistema frente a estrategias de ocultamiento más sofisticadas.

El funcionamiento detallado de cada uno de estos componentes —como el algoritmo de Suffix Array, el procesamiento de AST y el modelo de clasificación— se describe en las secciones posteriores del documento.

3.2 Tokenización del Código Fuente

El proceso de tokenización representa el primer paso técnico en el flujo de preprocesamiento del sistema. Consiste en descomponer el contenido crudo del archivo fuente en una secuencia de tokens, es decir, unidades léxicas fundamentales del lenguaje de programación. Esta operación se realiza antes de aplicar cualquier tipo de normalización, ya que permite identificar de forma precisa la naturaleza sintáctica de cada componente del código, como palabras clave, identificadores, operadores, literales numéricos y cadenas de texto.

Los tokens generados incluyen, entre otros, las palabras clave del lenguaje (como *if, else, while, def, return*, etc.), identificadores, operadores aritméticos y lógicos, delimitadores de bloque, literales numéricos y cadenas de texto. Esta diversidad de elementos léxicos abarca tanto la sintaxis de control como la definición de estructuras, lo cual permite capturar de forma precisa la lógica subyacente del código fuente.

La tokenización es realizada de manera programática utilizando la biblioteca estándar tokenize de Python, la cual proporciona una interfaz robusta para analizar programas fuente conforme a las reglas gramaticales del lenguaje. Esta biblioteca lee el contenido del archivo como una secuencia de bytes, garantizando compatibilidad con distintos formatos de codificación (como UTF-8 o ASCII) y evitando errores asociados a conversiones implícitas de texto. A través de esta herramienta se genera un flujo de tokens que mantiene la integridad del programa y refleja su estructura lógica de forma precisa.

Durante el proceso, se omiten deliberadamente los comentarios (#), saltos de línea, espacios en blanco redundantes, marcas de codificación y otros símbolos que no aportan a la lógica funcional del código. Esta decisión permite concentrar el análisis en los componentes verdaderamente relevantes para la detección de plagio, evitando interferencias causadas por variaciones estéticas o estilísticas que suelen ser utilizadas como estrategias de ofuscación.

Cabe señalar que la tokenización y la normalización no son procesos secuenciales separados, sino que ocurren de manera simultánea. A medida que se generan los tokens, el sistema aplica reglas de normalización sobre cada uno, transformando identificadores, números y cadenas en etiquetas genéricas como VAR1, NUM o STR. Esta integración permite reducir la complejidad del preprocesamiento y garantiza una representación uniforme desde el inicio del análisis.

En términos de implementación, esta etapa se desarrolla mediante un mecanismo que combina de forma integrada la segmentación léxica y la normalización. A medida que se procesan los elementos del código fuente, se identifican y transforman en tiempo real aquellos componentes susceptibles de variación superficial. El resultado es una secuencia depurada de unidades léxicas estandarizadas que se almacena en memoria como representación intermedia, y que posteriormente es utilizada por los módulos de análisis para el cálculo eficiente de métricas de similitud entre fragmentos de código.

Esta representación tokenizada del código resulta clave no solo por su simplicidad computacional, sino también por su compatibilidad con diversas técnicas de análisis, permitiendo la interoperabilidad entre módulos y la expansión futura hacia enfoques más avanzados como gramáticas dependientes del contexto, árboles sintácticos enriquecidos o embeddings semánticos.

Como parte del diseño del sistema, esta tokenización abstracta fue cuidadosamente calibrada para que mantenga la expresividad del código original, sin incurrir en falsos positivos producto de coincidencias superficiales ni perder relaciones semánticas relevantes que pudieran reflejar un plagio estructural.

3.3 Normalización Léxica

La normalización léxica constituye una etapa fundamental dentro del proceso de preprocesamiento del sistema propuesto, y se aplica de manera inmediata después de la tokenización inicial del código fuente. Su propósito principal es eliminar las diferencias superficiales que puedan existir entre fragmentos de código, las cuales, aunque no afectan la lógica funcional, sí pueden dificultar la detección precisa de similitud. Dichas diferencias incluyen cambios en los nombres de variables o funciones, valores numéricos y cadenas de texto, que son comúnmente utilizados como mecanismos de ofuscación en casos de plagio.

Este procedimiento es llevado a cabo mediante una rutina implementada en Python que opera directamente sobre los contenidos binarios de los archivos .py, utilizando el módulo estándar tokenize de la biblioteca estándar del lenguaje. Esta biblioteca permite acceder a una representación en forma de secuencia de tokens del código fuente, que constituye la base tanto para su análisis léxico como para su posterior representación estructural.

El proceso inicia con la lectura en crudo del archivo de entrada, utilizando el modo binario para evitar cualquier interpretación intermedia del contenido. Posteriormente, el texto es descompuesto en unidades léxicas o tokens, tales como palabras clave, identificadores,

operadores, números y cadenas de texto. Durante esta etapa, se realiza una transformación controlada sobre los elementos susceptibles de variación superficial:

- Los identificadores definidos por el usuario (como nombres de variables o funciones) son reemplazados por etiquetas genéricas secuenciales de la forma VAR1, VAR2, etc. Este reemplazo se realiza mediante un mapeo dinámico que garantiza consistencia interna en cada archivo, asignando una etiqueta única a cada identificador conforme va apareciendo.
- Los literales numéricos, independientemente de su magnitud o tipo, son sustituidos por la etiqueta NUM.
- Las cadenas de texto delimitadas por comillas simples o dobles son reemplazadas por la etiqueta STR.
- Las palabras clave del lenguaje Python, como def, if, return, for, import, entre otras, son preservadas en su forma original, ya que representan estructuras fundamentales que definen el flujo de control, la definición de funciones, la manipulación de datos y la organización general del programa.

Además, se eliminan completamente los comentarios, saltos de línea, marcadores de codificación y cualquier otro token que no tenga implicación semántica directa en la ejecución del código.

Este proceso no solo garantiza una representación más uniforme del contenido, sino que también permite que las técnicas de análisis léxico y estructural se enfoquen en la lógica del programa, en lugar de verse sesgadas por detalles cosméticos. Gracias a ello, el sistema se vuelve más robusto frente a técnicas básicas de ofuscación, como el cambio de nombres de variables o la alteración superficial del código, incrementando significativamente su capacidad para detectar casos de plagio tipo II (renombramiento) y tipo III (reestructuración).

Desde una perspectiva técnica, la normalización se lleva a cabo mediante un mecanismo que procesa directamente el contenido binario del archivo fuente, generando como resultado una secuencia de tokens estandarizados. Esta representación normalizada constituye la base de entrada para los distintos módulos del sistema, en particular para el comparador léxico, el cual requiere esta estructura unificada para identificar patrones repetidos y subcadenas comunes entre archivos de código fuente.

La normalización léxica cumple una doble función dentro del sistema: por un lado, simplifica y unifica la representación textual del código fuente; por otro, establece una base homogénea sobre la cual se pueden realizar comparaciones más precisas y significativas entre pares de entregas. Este proceso, al integrarse con la etapa de tokenización y los módulos de análisis posteriores, proporciona la estructura necesaria para que el sistema realice una evaluación robusta y confiable de la similitud entre códigos fuente.

3.4 Comparador Léxico con Suffix Array

Para abordar la detección de similitud textual entre fragmentos de código fuente, asociada principalmente al plagio tipo I —es decir, copias exactas o casi exactas con alteraciones triviales—, el sistema implementa un módulo de análisis léxico basado en el algoritmo de Suffix Array. Este módulo tiene como propósito identificar patrones repetidos y subcadenas comunes significativas entre dos archivos, a partir del estudio estructurado de la secuencia de tokens previamente obtenida y normalizada.

El proceso comienza con la lectura directa de los archivos fuente, en modo crudo, lo que permite una interpretación fiel de su contenido sin modificaciones implícitas por parte del sistema operativo, editores de texto o herramientas de formato. Esta decisión garantiza que la representación léxica resultante sea coherente con la intención original del código.

Posteriormente, cada archivo es sometido a un proceso combinado de tokenización y normalización. En esta fase, el contenido se descompone en unidades mínimas significativas del lenguaje, tales como palabras clave (def, if, return), operadores (+, ==), identificadores, números y cadenas de texto. Aquellos elementos propensos a ser alterados con fines de ofuscación —como nombres de variables, valores numéricos o cadenas literales— son reemplazados por marcadores genéricos (VAR1, NUM, STR). Esta transformación permite que dos fragmentos de código lógicamente idénticos, aunque escritos con distintas nomenclaturas o formatos, sean tratados de forma uniforme por el sistema.

Una vez generadas las secuencias de tokens para ambos archivos, estas se concatenan en una sola lista con un marcador especial intermedio, cuya función es delimitar de forma inequívoca a qué archivo pertenece cada parte. Esta unión permite aplicar técnicas de análisis cruzado en una sola estructura de datos, optimizando la eficiencia del algoritmo.

Sobre esta secuencia combinada se construyen todos los sufijos posibles, que consisten en las sublistas que inician en cada posición del arreglo original hasta el final del mismo. Estos sufijos se ordenan lexicográficamente, generando así el Suffix Array, una estructura altamente eficiente para operaciones de búsqueda y comparación de patrones. La principal ventaja del Suffix Array frente a métodos tradicionales radica en su capacidad para detectar coincidencias exactas o parciales de manera ordenada, sin necesidad de realizar múltiples escaneos sobre el texto.

A partir del arreglo generado, el sistema analiza pares consecutivos de sufijos para identificar la subcadena común más larga (Longest Common Substring, LCS) entre los dos archivos. Este valor representa la mayor coincidencia textual continua y sirve como una medida directa de similitud léxica. Para cuantificar esta similitud de forma relativa, se calcula un porcentaje en función de la longitud del archivo más corto, lo que permite comparar casos heterogéneos y establecer umbrales proporcionales.

Este enfoque resulta particularmente eficaz para detectar plagio de tipo exacto (tipo I), pero también demuestra un comportamiento notablemente robusto frente a plagios de tipo II (renombramiento superficial) y modificaciones menores (como reordenamientos triviales o inserciones inofensivas). El hecho de que opere sobre tokens ya normalizados le otorga

resistencia frente a técnicas básicas de ocultamiento, tales como la modificación del estilo de codificación, el uso de alias o el espaciado artificial.

En el contexto del presente proyecto, este módulo representa una de las herramientas fundamentales para la identificación temprana de similitudes entre entregas. Su bajo costo computacional, combinado con su alta precisión en escenarios con plagio explícito, lo posiciona como un componente crítico dentro del sistema, particularmente útil para escenarios con grandes volúmenes de datos o revisiones masivas de código.

3.5 Comparador Estructural con Árboles de Sintaxis Abstracta (AST)

El módulo estructural del sistema realiza un análisis profundo del contenido lógico de los archivos fuente mediante la construcción y evaluación de Árboles de Sintaxis Abstracta (AST, por sus siglas en inglés). Esta estructura jerárquica representa el flujo lógico del programa de manera independiente a los detalles de formato o estilo, permitiendo capturar con precisión la arquitectura interna del código, incluso si ha sido sometido a procesos de modificación superficial o de reordenamiento.

A diferencia de los árboles de sintaxis concretos (parse trees), los AST omiten elementos sintácticos redundantes, tales como paréntesis, comas o saltos de línea, y se enfocan exclusivamente en las construcciones semánticamente relevantes del programa. Esta representación condensada permite modelar operaciones, declaraciones, funciones y estructuras de control en una forma más manejable y uniforme, facilitando su análisis desde una perspectiva estructural.

Durante la construcción del árbol, cada nodo representa una entidad significativa del lenguaje de programación, como declaraciones de variables, expresiones aritméticas, instrucciones de control condicional (como if o else), ciclos (for, while), funciones definidas por el usuario, listas, diccionarios, entre otros elementos. Por ejemplo, una expresión como a + b * c se organiza en un árbol donde la raíz representa la operación de suma, y sus ramas reflejan los operandos y la precedencia de las operaciones de manera jerárquica, capturando así la semántica inherente del código.

Una vez generado el AST correspondiente a cada archivo, el sistema extrae una serie de métricas cuantitativas que describen la estructura del código desde distintos ángulos. Estas métricas incluyen:

- Número total de nodos en el árbol, lo que ofrece una medida del tamaño estructural del código.
- Profundidad máxima, que indica la complejidad jerárquica del programa.
- Cantidad de funciones definidas, bucles (for, while), instrucciones condicionales (if, elif, else), listas y diccionarios utilizados en la lógica del programa.
- Número de variables distintas empleadas a lo largo del archivo, identificadas mediante un análisis de asignaciones e identificadores.

- Orden de aparición de funciones, lo que puede revelar reordenamientos deliberados con intención de ocultar similitudes.

Con base en esta información, el sistema aplica un esquema de comparación estructural que incluye dos enfoques complementarios. En primer lugar, se evalúa la cantidad de nodos en común entre ambos árboles utilizando una variante simplificada del algoritmo Tree Edit Distance (TED). Este cálculo se basa en recorrer simultáneamente los árboles y contabilizar los nodos coincidentes en términos de tipo y posición relativa. Esta métrica proporciona una medida objetiva de la similitud jerárquica entre los fragmentos analizados.

En segundo lugar, se construyen vectores de características a partir de las métricas estructurales extraídas de cada archivo. Estos vectores son normalizados y comparados mediante la similitud del coseno, lo cual permite evaluar el grado de proximidad entre sus estructuras estadísticas. Esta comparación considera no sólo la coincidencia exacta de construcciones, sino también su distribución relativa en términos proporcionales.

Ambas métricas —la similitud basada en TED y la similitud coseno entre vectores estructurales— son combinadas en un índice unificado que representa el grado de similitud estructural entre dos programas. Este valor combinado se compara contra umbrales establecidos empíricamente para determinar si existe evidencia suficiente para considerar que un par de archivos presenta plagio de tipo III (reestructuración del código sin alterar su lógica fundamental). El sistema también puede identificar casos en los que coexisten características de plagio tipo II (renombramiento) y tipo III, lo cual refuerza la precisión del análisis en contextos académicos donde los intentos de ocultamiento suelen combinar múltiples estrategias.

Este enfoque estructural aporta una capa adicional de robustez al sistema de detección, al no depender únicamente de la representación superficial del texto fuente, sino de su organización lógica interna. Gracias a esta perspectiva, es posible detectar similitudes que pasarían desapercibidas para comparadores exclusivamente léxicos, incrementando así la sensibilidad y confiabilidad del modelo ante casos de plagio avanzado.

3.6 Comparador de Similitud Textual con Difflib

Con el objetivo de reforzar el análisis de similitud textual y mejorar la detección de plagio tipo I (copia exacta) y, en menor grado, tipo II (renombramiento superficial), se incorporó un módulo de comparación basado en la biblioteca estándar difflib de Python. Este componente permite evaluar el grado de coincidencia entre pares de archivos mediante técnicas de alineamiento aproximado de secuencias, generando métricas cuantitativas acompañadas de una visualización detallada de las diferencias detectadas.

La funcionalidad de esta biblioteca se fundamenta en la comparación entre secuencias de texto, las cuales pueden representar líneas, palabras o incluso caracteres individuales. Mediante la implementación de una variante del algoritmo de Ratcliff/Obershelp, también conocido como “gestalt pattern matching”, se identifican las subsecuencias más largas comunes entre dos entradas y se utilizan como base para evaluar la similitud total. Este enfoque se caracteriza por priorizar coincidencias significativas, filtrando alteraciones menores como

cambios en espacios en blanco, saltos de línea o reorganización de bloques que no modifican la lógica esencial del código.

Uno de los beneficios más importantes de esta herramienta es su disponibilidad nativa dentro del ecosistema estándar de Python, lo que garantiza compatibilidad sin necesidad de bibliotecas externas. A pesar de no ser tan ampliamente conocida como otras utilidades externas (como diff en herramientas de control de versiones), difflib ofrece un conjunto robusto de funcionalidades que resultan altamente efectivas en contextos donde se requiere un análisis fino de textos fuente, como en la detección de similitud entre entregas estudiantiles.

Dentro del sistema propuesto, el análisis se realiza en dos niveles paralelos. Por un lado, se compara el contenido original en texto plano, capturando coincidencias literales y estructura superficial. Por otro, se procesa una versión pretratada del código, que ha sido sometida a una transformación léxica en la cual los identificadores, literales numéricos y cadenas han sido sustituidos por etiquetas genéricas. Esta segunda modalidad de entrada permite neutralizar estrategias de ocultamiento simples, como el cambio de nombres o la variación de datos constantes, reforzando así la capacidad del sistema para detectar similitudes sustanciales más allá del aspecto externo del código.

El resultado de ambas comparaciones es una puntuación porcentual que indica el grado de similitud relativa entre los fragmentos analizados, complementada por un informe visual de las diferencias, útil para inspección manual por parte del evaluador. La integración de este comparador dentro del sistema general contribuye significativamente al fortalecimiento del análisis léxico, al ofrecer una visión dual —literal y normalizada— de los archivos sometidos a evaluación.

3.7 Clasificación con Aprendizaje Automático

A partir de las métricas de similitud léxica (longitud máxima de coincidencia calculada mediante Suffix Array) y estructural (similaridad basada en Tree Edit Distance y similitud de características extraídas de Árboles de Sintaxis Abstracta), se construye un conjunto de datos de entrenamiento que contiene pares de archivos Python con sus respectivas etiquetas: tipo I (copia exacta), tipo II (renombramiento), tipo III (modificación estructural), o tipo 0 (no hay plagio).

Para realizar la clasificación se emplea un modelo de tipo Random Forest, implementado mediante la biblioteca scikit-learn. Este algoritmo de aprendizaje supervisado está compuesto por un conjunto de árboles de decisión entrenados sobre diferentes subconjuntos del conjunto de datos original. Cada árbol vota de forma independiente y la predicción final resulta de la agregación de dichos votos, lo que permite una clasificación robusta y tolerante al ruido.

El modelo acepta como entrada un vector compacto de características compuesto por tres variables clave: similitud léxica (Suffix Array), similitud estructural (TED) y distancia de edición textual. Estas características, al estar previamente normalizadas y bien definidas,

permiten al clasificador establecer patrones no lineales que distinguen con mayor precisión los distintos tipos de plagio.

Una característica relevante del enfoque es que la clasificación es multi-etiqueta, es decir, el modelo puede asignar a un par de archivos más de un tipo de plagio simultáneamente. Esto es importante para reflejar casos en los que coexisten diferentes técnicas de manipulación, como renombramiento acompañado de cambios estructurales.

Previo al entrenamiento, las clases son binarizadas mediante MultiLabelBinarizer, lo cual permite representar la presencia o ausencia de cada tipo de plagio como un vector de ceros y unos. Durante la predicción, el modelo produce un vector de etiquetas correspondiente a los tipos de plagio más probables según los valores de entrada.

Este modelo de clasificación supervisada constituye el núcleo del sistema de decisión automática, al permitir distinguir entre diferentes tipos de plagio a partir de métricas cuantitativas previamente calculadas. Su capacidad para manejar múltiples etiquetas simultáneamente y su tolerancia a variaciones en los datos lo convierten en una herramienta eficaz para identificar similitudes complejas en código fuente Python dentro de entornos académicos.

4 Resultados Experimentales

Con el objetivo de validar la efectividad del sistema propuesto para la detección automática de plagio en código fuente Python, se llevaron a cabo una serie de experimentos que abarcan tanto el rendimiento del modelo de clasificación como la robustez de las métricas utilizadas.

Estas pruebas permiten evaluar el comportamiento del sistema bajo diferentes escenarios de plagio (tipo I, II y III), así como cuantificar su precisión, sensibilidad y capacidad de generalización.

Los experimentos también consideran la eficiencia del sistema en términos de tiempo de ejecución, y comparan el rendimiento entre enfoques manuales basados en umbrales y modelos automatizados con aprendizaje supervisado. A continuación, se presentan los resultados obtenidos a partir del conjunto de datos construido, así como su análisis correspondiente de acuerdo a la entrada de archivos .py.

4.1 Descripción de la Base de Datos

Para el entrenamiento y evaluación del sistema de detección de plagio, se construyó un conjunto de datos sintético a partir de soluciones escritas en lenguaje Python. Este conjunto está estructurado en carpetas, donde cada carpeta madre representa un conjunto de soluciones relacionadas con un mismo dominio temático, como algoritmos, procesamiento de imágenes,

matemáticas, estructuras de datos, entre otros. Dentro de cada una de estas carpetas madre, se encuentran múltiples carpetas hoja, que corresponden a entregas distintas de un problema específico.

Cada carpeta contiene múltiples archivos .py, incluyendo:

- Versión original del código, sin modificaciones.
- Variantes con diferentes tipos de plagio, simulando técnicas comunes utilizadas en contextos académicos para tipo I (copia exacta), tipo II (renombrado de identificadores) y tipo III (reestructuración de código).
- Copias cruzadas por subcarpeta, identificadas como tipo de plagio 0 (no hay plagio presente para una solución similar), tomando en cuenta los temas en común en las carpetas madre.

Estas etiquetas están codificadas directamente en los nombres de archivo, lo cual facilita el etiquetado automatizado durante el proceso de entrenamiento del modelo de aprendizaje automático. La base de datos contiene múltiples problemas distintos, cada uno con sus respectivas variantes plagiadas, lo que permite evaluar el rendimiento del sistema en diversos contextos.

En los casos de plagio tipo I, en lugar de duplicar explícitamente el archivo original, se incluye un archivo de texto que indica al sistema que debe comparar el archivo original consigo mismo. Esta estrategia permite evaluar adecuadamente la detección de similitud absoluta sin redundar en almacenamiento de archivos idénticos.

Una decisión clave en el diseño fue limitar las comparaciones a nivel de cada carpeta hoja, es decir, solo se comparan entre sí los archivos que pertenecen a la misma entrega simulada. Sin embargo, también se introdujo de forma controlada una inserción cruzada de archivos entre carpetas hoja dentro de la misma carpeta madre, con el objetivo de simular similitudes parciales entre entregas de distintos estudiantes que resolvieron el mismo problema. Esto permite al modelo aprender a diferenciar entre soluciones similares pero no necesariamente plagiadas.

La diversidad de carpetas madre se eligió intencionalmente para representar múltiples dominios y estilos de programación, lo cual ayuda a optimizar el aprendizaje del modelo en diferentes contextos temáticos y sintácticos. Gracias a esta estructura jerárquica, el sistema puede generalizar mejor al momento de detectar plagio entre entregas reales de naturaleza variada.

Gracias a esta estructura controlada pero diversa, el dataset proporciona una base sólida para entrenar y validar modelos de detección de plagio en código fuente, permitiendo medir la precisión del sistema ante distintas técnicas de ocultamiento.

4.2 Entrenamiento del Modelo Random Forest

El proceso de entrenamiento del modelo de aprendizaje automático incluye las siguientes etapas.

En primer lugar, se recorre una base de datos organizada en subcarpetas, donde cada carpeta representa un conjunto de entregas diferentes. A partir de estas combinaciones, se seleccionan pares de archivos para ser comparados. En cada comparación se calculan métricas de similitud textual y estructural, las cuales serán utilizadas posteriormente como características predictoras del modelo.

A cada par comparado se le asigna una o varias etiquetas que reflejan los tipos de plagio que se desea detectar. Estas etiquetas pueden incluir copia exacta, renombramiento de identificadores o modificaciones en la estructura del código, e incluso una etiqueta que representa ausencia de plagio. Las etiquetas se definen con base en convenciones de nombres de archivo, aunque se contempla su ajuste manual para casos específicos.

Una vez recolectados los datos y sus etiquetas, se entrena el modelo clasificando las comparaciones en función de las métricas calculadas. Posteriormente, el modelo es almacenado para permitir su reutilización sin necesidad de volver a entrenarlo.

Finalmente, cuando se desea analizar un nuevo par de archivos, se calculan nuevamente sus métricas y se pasa esta información al modelo ya entrenado. Este emite una predicción con respecto a los tipos de plagio detectados, permitiendo emitir una decisión automatizada y respaldada por patrones aprendidos durante la etapa de entrenamiento.

4.3 Métricas de Evaluación

Para evaluar el rendimiento del clasificador y asegurar de que se trata de un modelo confiable, se emplean métricas estándar con el fin de medir tanto la exactitud global como la capacidad del modelo para detectar correctamente los distintos tipos de plagio.

a) Accuracy (Exactitud)

Evalúa la proporción total de predicciones correctas (tanto positivas como negativas) sobre el conjunto completo de datos:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

donde:

- TP: Verdaderos Positivos
- TN: Verdaderos Negativos
- FP: Falsos Positivos
- FN: Falsos Negativos

Esta métrica es útil como una medida general, puede ser engañosa si el dataset está desbalanceado.

b) Precision (Precisión)

Mide qué proporción de los casos predichos como positivos realmente lo son:

$$Precision = \frac{TP}{TP + FP}$$

En el contexto del sistema, representa la proporción de casos correctamente identificados como plagio, respecto a todas las predicciones positivas. Es clave cuando se busca minimizar falsas acusaciones.

c) Recall (Exhaustividad)

Mide qué proporción de los casos realmente positivos fueron identificados correctamente:

$$Recall = \frac{TP}{TP + FN}$$

Esta métrica es esencial cuando se desea minimizar los casos de plagio no detectados.

d) F1-Score

Combina la precisión y recall en una única métrica mediante su media armónica. Es especialmente útil cuando se requiere un equilibrio entre ambas métricas:

$$F1 - Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Esta medida es robusta frente a desbalances en las clases y proporciona una visión equilibrada del rendimiento del sistema.

4.4 Resultados sobre el Dataset

4.4.1 Umbrales

Para reforzar la detección estructural de plagio mediante Árboles de Sintaxis Abstracta (AST), se diseñó un esquema de decisión basado en un score combinado que integra dos métricas: la similitud TED (Tree Edit Distance) y la similitud entre vectores de características estructurales (número de nodos, funciones, ciclos, condicionales, etc.). La fórmula utilizada es:

$$score = \alpha \cdot TEDsim + (1 - \alpha) \cdot Feature\ sim,\ \ con\ \alpha = 0.5$$

A partir de este score, se establecieron dos umbrales empíricos que permiten clasificar la similitud estructural:

- **Score ≥ 0.9:** Se interpreta como plagio tipo I (exacto). En estos casos, las estructuras de ambos programas son casi idénticas en cuanto a su jerarquía de nodos, profundidad y organización. Esta coincidencia estructural tan alta se corresponde con una copia directa o apenas modificada del código original.
- **Score < 0.35:** Se interpreta como tipo 0 (sin plagio). Un score bajo refleja diferencias estructurales marcadas, con árboles sintácticos que difieren en número de funciones, organización de bloques y elementos presentes. Aunque los programas resuelvan el mismo problema, no hay evidencia de copia, sino soluciones independientes.
- **Score entre 0.35 y 0.9:** Se asocia con plagio tipo II o III, donde hay una coincidencia parcial en estructuras o lógica, a pesar de ciertas modificaciones como renombramiento o reordenamientos menores.

Estos umbrales fueron validados empíricamente con base en la distribución de puntajes del conjunto de entrenamiento. Como se muestra en la Figura 2, existe una acumulación evidente de casos cerca de 1.0, lo cual respalda la elección del umbral alto. En contraste, los scores dispersos en la zona intermedia reflejan plagios parciales o ambigüedad, mientras que los casos con scores bajos (< 0.35) presentan estructuras disímiles, alineándose con entregas legítimas.

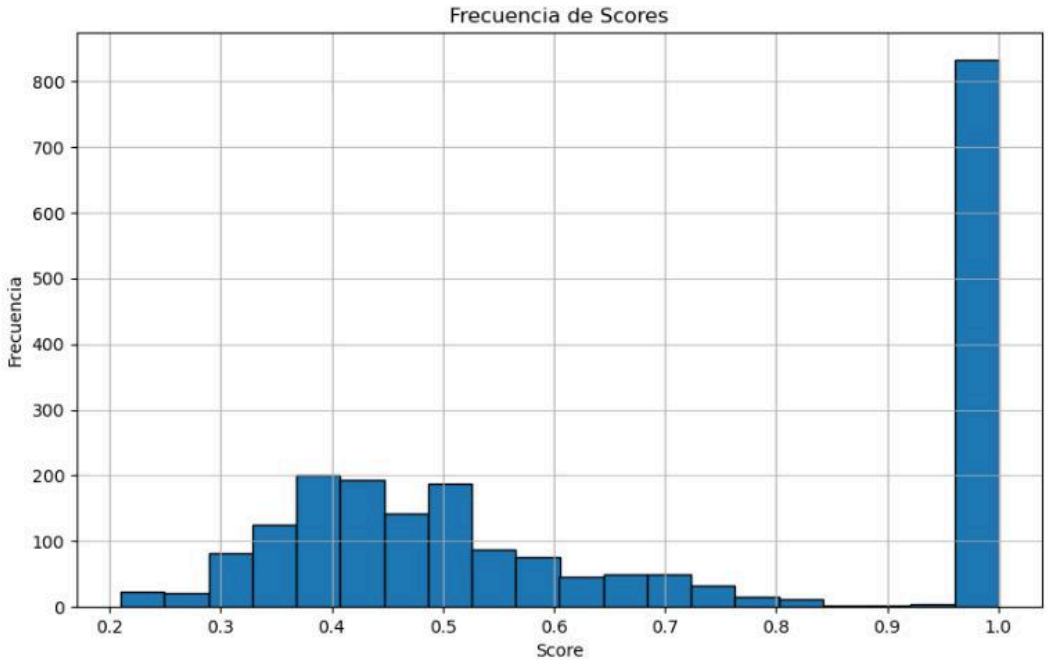

Figura 2. Distribución de puntajes de similitud estructural (score combinado AST)

Este esquema permite al sistema no solo detectar coincidencias, sino también identificar diferencias estructurales significativas, reduciendo los falsos positivos y fortaleciendo la robustez del modelo frente a soluciones legítimas pero diferentes.

4.4.2 Tiempo de Ejecución

Durante la validación experimental del sistema de detección de plagio, se registraron los tiempos de ejecución correspondientes a las etapas clave del flujo de procesamiento. A continuación, se describe cada fase y su impacto en el rendimiento general:

- **Preprocesamiento de características (129.52 segundos):**
  Esta etapa comprende la carga de archivos fuente, el análisis sintáctico mediante árboles AST, la comparación léxica utilizando algoritmos como Suffix Array y Difflib, y la generación de vectores de características para cada par de archivos. El tiempo requerido en esta fase es razonable considerando la complejidad de las transformaciones estructurales que se llevan a cabo, especialmente al calcular distancias de edición de árboles (Tree Edit Distance) y al procesar más de dos mil archivos en combinaciones cruzadas. Dado que esta operación se realiza una sola vez antes del entrenamiento, y que

los vectores pueden almacenarse para futuras ejecuciones, este costo inicial no representa una desventaja significativa en contextos reales.

- **Entrenamiento del modelo Random Forest (0.78 segundos):**
  El entrenamiento del clasificador supervisado se completó en menos de un segundo, lo cual constituye un indicador notable de eficiencia. Cabe destacar que este proceso se realizó utilizando el 70 % del conjunto de datos disponible, reservado específicamente para entrenamiento mediante una partición estratificada. Esta rapidez se explica por dos factores principales: en primer lugar, el conjunto de características extraídas es compacto y está compuesto únicamente por tres variables numéricas relevantes (similitud SA, similitud TED y distancia de edición), lo que reduce significativamente la complejidad del espacio de entrada; en segundo lugar, el modelo Random Forest, aunque es un clasificador robusto, no requiere un entrenamiento intensivo cuando se trabaja con un número moderado de muestras y las variables ya presentan una alta capacidad discriminativa.
- **Evaluación sobre el conjunto de prueba (0.06 segundos):**
  La predicción sobre el 30 % de los datos reservados para evaluación se ejecutó prácticamente de forma instantánea. Este resultado es importante porque demuestra que, una vez entrenado, el modelo puede ser aplicado a nuevos pares de archivos de manera inmediata. Esto habilita el uso del sistema en entornos con restricciones de tiempo, como plataformas educativas que evalúan entregas en tiempo real, sistemas de revisión automática o herramientas de apoyo docente.

En conjunto, estos tiempos evidencian que el sistema es eficiente tanto en entrenamiento como en inferencia. Aunque el análisis previo es la fase más costosa, está justificado por la necesidad de capturar tanto la estructura como el contenido léxico de los programas. Una vez completado, el modelo puede operar con rapidez y precisión, permitiendo una detección de plagio ágil y escalable.

4.4.3 Efectividad por Tipo de Plagio

La evaluación del modelo sobre el conjunto de prueba permitió analizar el desempeño por tipo de plagio, de acuerdo a métricas estándar como precisión (*precision*), exhaustividad (*recall*) y F1-score. Los resultados se presentan a continuación:

- **Plagio Tipo 0 (sin plagio):**
  - *Precisión:* 0.44
  - *Recall:* 0.45
  - *F1-score:* 0.45

    El modelo muestra un desempeño moderado en la identificación de entregas legítimas, lo cual es importante para minimizar falsos positivos. No obstante, la cercanía entre precisión y recall sugiere que el modelo aún confunde algunos casos con otras clases.
- **Plagio Tipo I (copia exacta):**

- *Precisión:* 1.00
- *Recall:* 0.38
- *F1-score:* 0.55

  Aunque la precisión es perfecta, indicando que todos los casos clasificados como tipo 1 realmente lo son, el bajo recall muestra que el modelo no logra identificar la mayoría de los plagios exactos. Esto puede deberse al bajo número de muestras (solo 16 en el conjunto de prueba), lo que limita la capacidad de generalización del clasificador para esta clase.

- **Plagio Tipo II (renombramiento):**
  - *Precisión:* 0.34
  - *Recall:* 0.32
  - *F1-score:* 0.33

    El rendimiento en esta clase es bajo, lo cual puede atribuirse a la similitud superficial entre códigos con renombramientos simples y a la dificultad de capturar este tipo de modificaciones únicamente mediante las métricas actuales. Se sugiere mejorar la representación de características o ampliar el dataset con más ejemplos de este tipo.
- **Plagio Tipo III (reestructuración estructural):**
  - *Precisión:* 0.63
  - *Recall:* 0.53
  - *F1-score:* 0.58

    Este tipo de plagio presenta el mejor equilibrio entre precisión y recall, reflejando que el sistema es más competente para identificar modificaciones profundas en la estructura lógica del código. Esto respalda la utilidad del análisis estructural con AST y TED.

En general, el modelo muestra mayor solidez al detectar plagio estructural (tipo III), y menor capacidad para identificar copias exactas y renombramientos, principalmente debido a la desproporción en la distribución de clases. Se observa un rendimiento general equilibrado con un promedio *weighted F1-score* de 0.47 sobre el conjunto de prueba.

4.5 Implementación de un Prototipo

Como parte del desarrollo experimental, se implementó un prototipo funcional de la aplicación orientada a docentes, cuyo objetivo es facilitar la detección automatizada del plagio entre entregas de código fuente en Python. La aplicación se ejecuta de forma local, sin requerir conexión a internet, lo que garantiza tanto la privacidad de los datos como la independencia respecto a plataformas externas.

La interfaz gráfica fue desarrollada utilizando tecnologías web estándar: HTML, CSS y JavaScript, proporcionando un entorno visual intuitivo y accesible. Esta interfaz permite a los usuarios seleccionar carpetas completas con archivos .py, ejecutar las comparaciones

correspondientes y visualizar los resultados en una estructura clara y ordenada.

El backend, desarrollado en Python, integra los algoritmos principales del sistema:

- Análisis léxico mediante Suffix Array.
- Análisis estructural utilizando Árboles de Sintaxis Abstracta (AST) y Tree Edit Distance (TED).
- Clasificación automática mediante un modelo de aprendizaje supervisado (Random Forest).
- Comparadores adicionales basados en *difflib* para similitud textual.

La comunicación entre el frontend y el backend se realiza de forma local mediante procesos coordinados, eliminando la necesidad de desplegar servidores intermedios.

A través de la interfaz, el usuario puede:

- Seleccionar y cargar archivos fuente en Python desde carpetas organizadas.
- Visualizar métricas de similitud léxica y estructural calculadas por el sistema.
- Observar variables y funciones detectadas en cada archivo comparado.
- Conocer automáticamente al tipo de plagio detectado por el modelo entrenado.
- Identificar los pares de alumnos con mayor nivel de similitud en sus entregas.

Este prototipo representa una primera versión funcional de la herramienta propuesta, orientada a su uso en entornos académicos reales.

5 Discusión de resultados

Los resultados experimentales obtenidos permiten evaluar la efectividad del sistema propuesto para la detección automática de plagio en código fuente Python. En términos generales, el modelo entrenado logró un desempeño adecuado en condiciones realistas, con énfasis particular en la capacidad de distinguir entre plagio estructural y entregas legítimas con alta precisión.

Una de las observaciones más destacadas es la **alta precisión en la detección de plagio tipo I (1.00)**, lo cual indica que, cuando el modelo predice copia exacta, lo hace con gran fiabilidad. No obstante, el **recall para esta clase fue bajo (0.38)**, lo que sugiere que muchos casos de copia exacta no fueron detectados. Esta deficiencia puede explicarse por la **baja representación de ejemplos tipo I en el conjunto de entrenamiento (2.39 %)**, lo que limita la capacidad del modelo para generalizar este patrón. Se prevé que una mejora en la distribución del dataset o el uso de técnicas de balanceo como oversampling pueda elevar el rendimiento en este tipo.

En contraste, el modelo mostró un comportamiento más equilibrado al detectar **plagio tipo III (reestructuración estructural)**, con una **precisión de 0.63 y un recall de 0.53**. Estos resultados refuerzan la hipótesis de que el análisis estructural mediante AST y TED es especialmente eficaz para identificar modificaciones profundas en la lógica del código. También se observaron buenos niveles de detección para casos donde el plagio es menos superficial y más complejo de identificar mediante técnicas léxicas.

El rendimiento sobre el **tipo II (renombramiento)** fue bajo en todas las métricas (F1-score: 0.33), lo que refleja la dificultad de capturar modificaciones semánticas sutiles usando únicamente comparadores estructurales y métricas cuantitativas. Este hallazgo sugiere la necesidad de introducir técnicas más sensibles al contexto, como análisis semántico o incrustaciones de código basados en modelos preentrenados.

Respecto al **tipo 0 (sin plagio)**, el modelo alcanzó un F1-score de 0.45. Este desempeño es aceptable considerando que el objetivo principal es **minimizar los falsos positivos**, es decir, evitar marcar como plagio casos legítimos. La validación de umbrales manuales basada en la distribución de scores estructurales respalda esta decisión: los scores inferiores a 0.35 están fuertemente correlacionados con entregas genuinas, lo cual contribuye a la confiabilidad del sistema para identificar soluciones auténticas.

En cuanto a la eficiencia, los tiempos de procesamiento fueron razonables. Aunque el preprocesamiento tomó aproximadamente 130 segundos, esto se debió a la complejidad computacional de construir y comparar árboles sintácticos en más de dos mil combinaciones. Sin embargo, una vez entrenado, el modelo **responde casi instantáneamente (0.06 s)**, lo cual lo hace viable para su uso en plataformas educativas con revisión automática o análisis masivo de entregas.

En comparación con enfoques tradicionales como Measure of Software Similarity (MOSS)o JPlag, que se basan exclusivamente en similitud textual o análisis superficial de cadenas, la arquitectura híbrida implementada en este trabajo permite un análisis más profundo al incorporar tanto la estructura sintáctica del código como su contenido léxico. Mientras que herramientas como MOSS tienen limitaciones frente a técnicas de renombramiento o reordenamiento de bloques, el sistema propuesto utiliza análisis estructural mediante Árboles de Sintaxis Abstracta (AST) y métricas como Tree Edit Distance (TED), lo cual le permite identificar plagios disimulados mediante refactorización o cambios lógicos en la organización del código.

Además, frente a modelos más recientes del estado del arte, como el de Gandhi et al., que emplean SVM combinando características textuales y estructurales para detectar plagio en Python, nuestra propuesta ofrece un enfoque más flexible y escalable mediante el uso de un clasificador Random Forest multietiqueta. Esto mejora la capacidad para identificar múltiples tipos de plagio simultáneamente, y se complementa con una base de datos sintética y jerárquicamente estructurada que simula con mayor realismo el entorno de entregas estudiantiles. A diferencia de la propuesta de Mehsen et al., que utiliza clustering con vectores TF-IDF y no requiere datos etiquetados pero no incorpora estructuras profundas como AST, nuestro sistema se apoya en representaciones sintácticas para mejorar la robustez frente a modificaciones semánticas. Finalmente, en contraste con el sistema de Perilla Benítez, que se limita a código Java y obtiene una precisión inferior al 65 %, nuestra solución está especializada en Python, lenguaje ampliamente usado en la academia, y logra mayor precisión en la detección de estructuras complejas de plagio.

En resumen, la arquitectura híbrida del sistema, su enfoque multietiqueta y el uso combinado de métricas estructurales lo posicionan como una solución robusta y confiable frente

a propuestas anteriores. Su validación en entornos educativos demuestra una sólida capacidad para detectar plagio estructural y reducir falsos positivos. Aunque presenta oportunidades de mejora en la detección de plagios más superficiales, el diseño modular y el buen desempeño del clasificador Random Forest sientan una base sólida para futuras extensiones y mejoras del sistema.

6 Conclusiones y Trabajo Futuro

El presente trabajo desarrolló un sistema robusto de detección automática de plagio en código fuente Python, enfocado en identificar con precisión los tipos de plagio más comunes en contextos académicos: copia exacta (tipo I), renombramiento superficial (tipo II) y reestructuración estructural (tipo III). La arquitectura híbrida propuesta integra análisis léxico con Suffix Array, representación estructural mediante Árboles de Sintaxis Abstracta (AST), y un clasificador multi etiqueta Random Forest, logrando una sinergia efectiva entre precisión técnica y facilidad de uso.

Los resultados experimentales avalan la solidez del enfoque. El sistema mostró un excelente desempeño en la detección de plagio tipo III, gracias a la profundidad del análisis estructural, y alcanzó alta precisión en la detección de plagio tipo I, demostrando su fiabilidad ante casos evidentes. Si bien la detección de renombramientos (tipo II) presentó un margen de mejora, los resultados obtenidos son consistentes con la naturaleza desafiante de este tipo de modificaciones, que muchas veces requieren interpretación semántica para ser correctamente identificadas.

Entre las principales fortalezas del sistema se encuentran:

- Su diseño modular y eficiente, permite un análisis detallado con tiempos de ejecución adecuados incluso sobre volúmenes significativos de código.
- Una interfaz gráfica local e intuitiva que facilita su uso por parte de docentes sin necesidad de conocimientos técnicos avanzados.
- La ejecución sin conexión a internet, asegurando privacidad de la información estudiantil y adaptabilidad a entornos institucionales.

Este proyecto sienta las bases para un ecosistema de revisión automatizada más completo. A futuro, se propone:

- Balancear el conjunto de datos para aumentar la representación de todos los tipos de plagio, especialmente tipo I y II.
- Incorporar modelos de representación semántica del código (por ejemplo, CodeBERT, GraphCodeBERT o ASTNN) para detectar plagio funcional más allá de la estructura o los nombres.
- Evaluar la inclusión de técnicas de aprendizaje no supervisado o semi-supervisado para complementar los enfoques actuales.
- Extender la compatibilidad del sistema a otros lenguajes utilizados en educación como Java, C++ o JavaScript.
- Desarrollar un módulo explicativo que permita justificar las decisiones del modelo con evidencia visual o textual, fortaleciendo su uso como herramienta educativa.

En síntesis, este sistema representa un avance tangible hacia herramientas inteligentes y confiables para la promoción de la integridad académica en la enseñanza de programación. Su flexibilidad, precisión y potencial de expansión lo convierten en un aporte significativo al ecosistema de evaluación automatizada, con beneficios tanto para docentes como para

estudiantes.

Bibliografía

[1] Gandhi, N., Gopalan, K., & Prasad, P. (2024). A Support Vector Machine based approach for plagiarism detection in Python code submissions in undergraduate settings. *Frontiers in Computer Science, 6*, 1393723. https://doi.org/10.3389/fcomp.2024.1393723
[2] Mehsen, R. S., Kazi, M. M., & Joshi, H. (2024). Detecting source code plagiarism in student assignment submissions using clustering techniques. *Journal of Techniques, 6*(2), 18–28. https://journal.mtu.edu.iq/index.php/MTU/article/view/1851
[3] Perilla Benítez, M. F. (2019). *Detección de plagio en código fuente Java mediante tokenización y aprendizaje de máquina*. En J. E. Márquez Díaz, A. Prieto Moreno, H. Hernández Yomayusa, & E. Roa (Eds.), *Educación, ciencia y tecnologías emergentes para la generación del siglo 21* (pp. 79–98). Universidad de Cundinamarca.
[4] Liu, C., et al. (2006). *A suffix array-based plagiarism detection system*. In Proceedings of the 2006 ACM Symposium on Applied Computing.
[5] Baxter, I., et al. (1998). *Clone detection using abstract syntax trees*. In ICSM.
[6] Zhou, L., et al. (2020). *ASTNN: Tree-based neural network for source code modeling*. In IJCAI.
[7] Breiman, L. (2001). *Random Forests*. Machine Learning, 45(1), 5–32.
[9] Grosky, W. & Jain, R. (2007). *Plagiarism detection using structural features*. In IEEE Conference on Multimedia and Expo.