Tecnológico de Monterrey

Identificación de Plagio en Código usando Redes Neuronales

Autor:
Sarah Guadalupe Martínez Navarro
A01703113

Fecha:
04 / Junio / 2025

TC3002B
Desarrollo de aplicaciones avanzadas de ciencias computacionales

I. Descripción

Proyecto académico para la materia Desarrollo de aplicaciones avanzadas de ciencias computacionales. Este artículo y repositorio implementa un modelo de aprendizaje supervisado con TensorFlow Keras para la identificación de plagio en bloques de código en el lenguaje Java. Se ha desarrollado con la técnica de Aprendizaje Supervisado. Esta es una subárea del Machine Learning en la que el modelo se entrena para identificar patrones alimentándose de un conjunto de datos con etiquetas (en este caso, utilizando 2 etiquetas). Se busca que el sistema realice una tarea de identificación donde pueda señalar la copia de código, el porcentaje de similitud entre bloques de código y una decisión de si el código es plagiado o no.

Link a repositorio de código: https://github.com/sarahmtz02/NN_PlagiarismDetector

II. Introducción

En 2024, el mercado global de la IA alcanzó un valor de 2,41 billones de dólares, y se proyecta que para 2034 supere los 30 billones, con una tasa de crecimiento anual compuesta del 32,4% . Tras el nacimiento del internet y, con ello, la automatización de soluciones, el avance del aprendizaje automático, entre otras tecnologías, se ha explotado al nivel que conocemos hoy el tema de la Inteligencia Artificial. De aquí nace el concepto de la Cuarta Revolución Industrial o Industria 4.0, la cual consiste de un fenómeno caracterizado por la convergencia y aceptación de diversas tecnologías de áreas físicas, digitales y biológicas, entre otras, en especial en los últimos años que, similar a la carrera al espacio, ahora existe la carrera entre empresas y países de quién desarrolla la Inteligencia Artificial más poderosa del mundo.

Sin embargo, este auge y rápido crecimiento de las tecnologías como la IA han traído consigo no sólo lo bueno, sino también han traído nuevas preocupaciones y dudas en múltiples ámbitos, entre ellos el tema de la presencia de estas tecnologías en la educación de hoy en día. En el contexto educativo y profesional, el plagio, especialmente en el código fuente, se ha convertido en un problema cada vez más frecuente. Herramientas como ChatGPT, GitHub Copilot y otros generadores de código automatizado, aunque útiles para aprender y mejorar la productividad, también facilitan el copiado y reutilización de fragmentos de código sin atribución adecuada. Según las estadísticas, varias universidades han reportado aumentos del 50% en casos de plagio en carreras de ingeniería informática entre 2020 y 2023, según estudios recogidos por el Journal of Computing Sciences in Colleges (2023). Esto plantea desafíos importantes tanto para docentes que buscan evaluar conocimientos, como para empresas que deben proteger sus activos tecnológicos. Como en el tema de seguridad, por cada parche que se implemente o se trabaje, ya hay 3 tecnologías más nuevas que te permiten darle la vuelta. Actualmente, la demanda de las herramientas capaces de analizar grandes volúmenes de código y detectar patrones de similitud va en aumento, en especial debido a que los métodos tradicionales se están quedando

obsoletos en tiempo, eficiencia y entendimiento. Claro, se pueden utilizar búsqueda de cadenas o análisis léxico para el desglose del código bajo estudio, pero no son suficientes para detectar plagios más sofisticados, como el cambio de nombres de variables o la reorganización de bloques de código que son los cambios usuales que se realizan después de haber copiado algo.

En este contexto, las redes neuronales artificiales aparecen como una posible solución. Gracias a su capacidad para aprender representaciones abstractas y captar similitudes más allá de la superficie del texto, estos modelos permiten detectar plagio incluso en casos en que el código ha sido reestructurado para evadir comparadores tradicionales. Una herramienta de este tipo puede analizar no solo la sintaxis, sino también la semántica del código, identificando fragmentos que conservan su lógica aunque hayan sido modificados superficialmente. Esto es especialmente útil en lenguajes como C++, donde la flexibilidad del lenguaje facilita las técnicas de ocultamiento de plagio.

III. State of the Art

Al igual que con el tema de la ciberseguridad que por cada parche y solución que salga a la luz salen múltiples tecnologías y herramientas nuevas para cumplir con el trabajo de un atacante, el campo del plagio en código fuente es un desafío que evoluciona constantemente debido a la velocidad de desarrollo de automatización de soluciones e inteligencia artificial en lo que conocemos como la Cuarta Revolución Industrial o la Industria 4.0. Bajo este contexto, son muchos los estudios e investigaciones realizados para ofrecer un mejor entendimiento al tema y orientar el desarrollo de herramientas más precisas y escalables que puedan usarse para solucionar la problemática que presentamos hoy. Como parte de este artículo, se analizaron algunos de estos trabajos ya existentes para realizar un acercamiento a nuestra solución. Entre estos, los siguientes artículos:

[1] O. Karnalim, S. Budi, H. Toba, and M. Joy, “Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation,” *Informatics in Education*, vol. 18, no. 2, pp. 321–344, 2019.

[2] D. Guo *et al.*, “GRAPHCODEBERT: PRE-TRAINING CODE REPRESENTATIONS WITH DATA FLOW,” presented at the ICLR 2021, Sep. 2021.

[3] A. Eppa, “Source Code Plagiarism Detection: A Machine Intelligence Approach,” in *2022 IEEE Fourth International Conference on Advances in Electronics, Computers and Communications (ICAECC)*, 2022, pp. 1–5.

[4] S. Surendran, “Plagiarism Detection in Source Code using Machine Learning,” Master’s Thesis, University of Windsor, Windsor, ON, Canada, 2024.

El trabajo de Karnalim *et al.* [1] destaca la necesidad de un **dataset estandarizado** y

comprensivo para la evaluación de detectores de plagio. Su contribución principal es la creación de un repositorio de datos público que simula variados escenarios de plagio, incluyendo modificaciones avanzadas y la intención subyacente de plagiar (como por ejemplo el cambio de variables cuando el código es similar). El método inicial de acercamiento al problema se basa en la recuperación de información (IR) para establecer líneas base para la detección. La principal relevancia de este trabajo para nuestro proyecto reside en la metodología de construcción y curación de datasets. Para entrenar una red neuronal que pueda discernir entre el plagio y código original en código Java, es imperativo disponer de un dataset voluminoso que refleje la complejidad de las modificaciones de código plagiado, dejando al modelo aprender patrones semánticos robustos, no solo léxicos o sintácticos.

Dentro de Guo *et al.* [2] abordan un problema fundamental en el procesamiento de lenguajes de programación: la representación del código. Argumentan que los modelos pre-entrenados existentes suelen tratar el código como una mera secuencia de caracteres y tokens, ignorando su estructura al igual que su semántica. Su propuesta es un modelo pre-entrenado que se llama GraphCodeBERT que integra el **flujo de datos (data flow)** como una estructura semántica particular. El flujo de datos relaciona las variables dentro de la funcionalidad, proporcionando una comprensión más profunda de la lógica del código. Este acercamiento es aplicable a la detección de plagio en Java de este proyecto. El plagio usualmente se refiere a transformaciones que alteran la forma superficial del código como cambios de nombres de variables y funciones pero mantienen el mismo comportamiento funcional. Al emplear representaciones de código que encapsulan el flujo de datos, las redes neuronales aprenden a identificar similitudes a nivel semántico, detectando plagio incluso en presencia de refactorizaciones significativas, cambios de nombres o inserciones de código externo. También argumenta el uso de una arquitectura híbrida usando, aparte de un modelo pre-entrenado como CodeBERT, métodos simbólicos como los ASTs (Árboles de sintaxis abstracta por sus siglas en inglés). Esta dualidad permite un rango mayor para la identificación de múltiples tipos de plagio tanto sintáctico como léxico y semántico.

Mientras tanto, Eppa [3] busca investigar la aplicación de **métodos de aprendizaje automático y Deep Learning** para la detección de plagio. El estudio evalúa la efectividad de diversas técnicas, incluyendo KNN, SVM, árboles de decisión, redes neuronales recurrentes (RNNs) y redes transformadoras basadas en atención, sobre un dataset de pares de código. Los resultados demuestran que las metodologías de aprendizaje automático y profundo superan consistentemente a los detectores de plagio tradicionales basados en texto. Ofrece un punto de partida para la selección y experimentación con arquitecturas de redes neuronales, como RNNs y Transformers, que han demostrado éxito en tareas de procesamiento de lenguaje natural y código. La metodología de preparación de un *dataset* de pares de código para entrenamiento y evaluación es un componente clave a emular.

La tesis de Surendran et al. [4] refuerza la viabilidad de la detección de plagio de código fuente utilizando aprendizaje automático. Presenta una exploración de las metodologías como el preprocesamiento de código, la extracción de características y la evaluación comparativa de modelos con diferentes capacidades. Proporciona una base para identificar las técnicas más efectivas de preprocesamiento de código Java con su estudio en los temas de normalización, tokenización, análisis de árbol de sintaxis abstracta (ASTs) y las métricas de evaluación para medir el rendimiento de los modelos de redes neuronales en la detección de plagio. Esta es la fuente principal para este artículo. Consistiendo de 84 páginas, la tesis de la estudiante de la Universidad de Windsor es la fuente más completa que se encontró para el desarrollo de este modelo.

En estos y varios papers más se habla de 6 clasificaciones o tipos de plagio que se estudian a través de los modelos desarrollados en orden de más sencillo de identificar a más complejo.

1. El tipo más sencillo de entender y explicar es el **plagio literal**, también conocido como la copia idéntica. Esto quiere decir que se reproduce un código exactamente igual al código fuente de un autor por parte de otra persona sin algún tipo de alteración. Este tipo de plagio se caracteriza por un porcentaje de similitud de entre 95-100%, casi perfecta en la secuencia de caracteres, tokens o líneas de código. Incluso llega a mantener la indentación del código, los mismos nombres para variables o funciones e incluso comentarios del código original. Al igual que es sencillo de entender, es sencillo de encontrar. Se puede hacer a través de herramientas basadas en comparación textual sin necesidad de alguna tecnología que deba procesar el texto como código literal (en nuestro caso, el uso de javalang). No requiere como tal de una red neuronal para hacer la comparación ya que puede hacerse desde una aplicación mucho más sencilla usando métodos cuantitativos como TF-IDF o Cadenas de Markov, pero también se puede usar una red neuronal.

2. En segundo lugar tenemos el **plagio con mínimas modificaciones** que, como lo dice su nombre, se ocupa meramente de hacer cambios superficiales al código fuente. Este tipo de plagio busca eludir la detección de su copia sin alterar la funcionalidad del bloque. Incluye modificaciones tales como el renombrado de identificadores de variables, métodos y/o clases, la alteración o eliminación de comentarios, ajustes de formato de indentación o añadir “código muerto” que no cumple con ninguna función simplemente para realizar un cambio. En este caso, ya no es tan sencillo de identificar en comparación textual debido a que con los renombramientos de variables y otros cambios de los anteriormente mencionados, cuando un sistema esté analizando caracter por caracter, no se va a topar con mucho plagio. Es por esto que requerimos un análisis más profundo de la estructura léxica a través del uso de tokens y sintáctica haciendo uso de **árboles de sintaxis abstracta (ASTs)**. Estas herramientas igualmente pueden desarrollarse sin necesidad de una red neuronal, pero aquí empieza a demostrarse más el potencial de uso de estas mismas.

3. A continuación, hablaremos del **plagio con reestructuración estructural**. Aquí se realizan cambios más significativos al código manteniendo su funcionalidad. Esto se puede lograr reordenando sentencias independientes, cambiar el uso de estructuras de control con otros equivalentes, fusionar o dividir bloques de código, entre otros cambios. En este tipo de plagio no sólo estamos hablando del parafraseo de palabras reservadas o variables, sino que también de una reorganización del código donde en vez de una sóla función pueden existir tres con diferentes secciones del código original. La detección de este tipo de plagio requiere de un análisis estructural similar al anterior, haciendo uso de **árboles de sintaxis abstracta (ASTs)** en su totalidad de profundidad. El uso de redes neuronales a partir de este punto es ampliamente sugerido, entrenadas con representaciones del código o usando **embeddings semánticos** como el enfoque de **GraphCodeBERT** para realizar un análisis que trascienda de las diferencias superficiales.

4. Después se encuentra el **plagio con equivalencia semántica** o **equivalencia algorítmica**. En este caso, el código plagiado puede ser sintácticamente muy distinto al original, sin embargo implementa el mismo algoritmo o produce la misma funcionalidad que el código original. Las características de este tipo incluyen el cambio, ahora en vez de palabras reservadas, del algoritmo por otro que haga lo mismo, cambiar las librerías o APIs que se utilizan o, incluso, la optimización o desoptimización intencional del código para ocultar el plagio del mismo. Identificar la equivalencia semántica demanda un análisis funcional profundo que sea insensible a las variaciones sintácticas. Aquí el uso de **flujo de datos** como el que fue utilizado en el artículo de Guo *et al.* [2] y el **flujo de control** es crítico. Una red neuronal debe ser capaz de generar una o múltiples representaciones de código que encapsulan su "intención" subyacente, lo cual requiere técnicas avanzadas de **análisis estático** o **dinámico**.

5. También existe el **plagio mixto**, el cual involucra no sólo un código original sino diversos fragmentos de código de diferentes fuentes, algunas veces complementado con código nuevo para darle coherencia. Para el análisis y detección de este tipo de plagio hay un gran desafío: la necesidad de comparar secciones de código más pequeñas en lugar del programa completo. Para esto se necesita usar **subgrafos**, **sub-ASTs** o **fragmentos de embeddings**, además de la memoria y capacidad del dispositivo para correr el análisis.

6. Finalmente se encuentra el **plagio de ideas o conceptos**. Este se sitúa en el nivel más abstracto de todos los tipos mencionados aquí. No existe una copia directa del código, sino la apropiación de la lógica algorítmica, el diseño de la solución o la estrategia particular de un autor para resolver un problema. Aunque el código puede ser completamente original, la base conceptual ha sido plagiada. Este tipo de abstracción presenta un reto para los sistemas de detección automatizados como lo es una red neuronal. Para el desarrollo de un sistema así, el enfoque principal debe dirigirse a la detección robusta del plagio literal, con mínimas modificaciones, con reestructuración estructural y con

equivalencia semántica. Esto implica un preprocesamiento sofisticado de datos (**generación de ASTs**, **grafos de flujo de datos y control**), el diseño de arquitecturas de redes neuronales capaces de procesar estructuras complejas como **GNNs** o mediante el uso de **Transformers**, y la construcción de un **dataset diverso** que cubra adecuadamente estas tipologías de plagio.

IV. Sobre el Dataset

Recuperado de: [4] Slobodkin, “ConPlag: a Dataset of Programming Contest Plagiarism in Java”, Sadovnikov, 2022. [En línea]. Disponible: https://zenodo.org/records/7332790 . [Accedido: 3-jun-2025].

La evaluación de algoritmos y sistemas para la detección de plagio en código fuente depende de la disponibilidad de datasets estandarizados y representativos. En este contexto, el dataset **ConPlag** surge como una herramienta valiosa, habiendo sido diseñado para facilitar la investigación en la detección de plagio en lenguajes de programación. Su estructura está pensada para permitir la evaluación de la robustez de los detectores frente a diversas técnicas de ofuscación de código.

```
ConPlag Dataset
├── resources
│   ├── bplag
│   ├── jplag
│   ├── sherlock
│   └── sim
└── scripts
    ├── __init__.py
    ├── algorithm.py
    ├── bplag.py
    ├── dolos.py
    ├── jplag.py
    ├── metrics.py
    ├── moss.py
    ├── runner.py
    ├── sherlock.py
    ├── sim.py
    └── utils.py
└── versions
    ├── bplag_version_1
    ├── bplag_version_2
    ├── version_1
    ├── version_2
    ├── labels.csv
    ├── test_pairs.csv
    └── train_pairs.csv
└── README.md
└── requirements.txt
```

Dentro de Resources:

Bplag es una herramienta de detección de plagio de código fuente académico basado en el comportamiento. Representa un conjunto de entregas de tareas en un formato gráfico que representa su comportamiento dinámico de ejecución. Luego, compara estos gráficos para medir la similitud de comportamiento. Una alta similitud indica plagio. (https://github.com/hjc851/BPlag)

JPlag encuentra similitudes entre pares de programas. Detecta con fiabilidad el plagio y la colusión en el desarrollo de software, incluso cuando está ofuscado. Todas las similitudes se calculan localmente; el código fuente y los resultados de plagio nunca se publican en línea. (https://github.com/jplag/JPlag)

Sherlock es un programa que encuentra similitudes entre documentos textuales. Utiliza firmas digitales para encontrar fragmentos de texto similares. Una firma digital es un número que se forma al convertir varias palabras de la entrada en una serie de bits y unirlos para formar un número. (https://github.com/diogocabral/sherlock)

Sim se utiliza para medir la similitud entre dos programas de computadora en C. Es útil para detectar plagio entre un conjunto grande de tareas. Se enfoca meramente en fragmentos de texto. (https://github.com/mpanczyk/sim)

Dentro de Scripts:

Aquí se encuentran los scripts para correr los programas que están dentro de Resources al igual que mandar llamar otras herramientas necesarias para desglosar y comprender los archivos en evaluación de plagio.

Dentro de Versions:

Versiones: En las primeras 4 carpetas se encuentran los archivos en formato .java donde se encuentran los bloques de código a comparar en búsqueda de la detección de plagio.

Labels: Dentro del archivo de labels.csv se encuentran los pares que se comparan en busca de la detección de plagio junto con la etiqueta binaria que define un valor de 0 como 'no plagio' y el valor 1 como 'plagio detectado'. Estas etiquetas son las que mantienen nuestro modelo en el área de aprendizaje supervisado y permiten que sea más sencillo entrenar el modelo.

Test y train pairs: Los archivos test_pairs.csv y test_pairs.csv contienen solamente los pares de identificadores de archivos a comparar. Estos documentos representan la división de archivos y comparaciones que entran para entrenar el modelo para posteriormente probarlo con los archivos y pares de test.

V. Propuesta de Solución

Como se mencionó en la sección de Estado del Arte, la base principal de este proyecto ha sido la tesis de Surendran et al. [4]. Se hizo el análisis de la propuesta en la tesis y se determinó que la solución para este proyecto sería el uso de una red neuronal convolucional (CNN) y una red neuronal recurrente (RNN), junto con una red de memoria larga a corto plazo (LSTM) y un modelo pre-entrenado (PTM), en este caso estaremos usando GraphCodeBERT. Para el preprocesamiento de datos, necesitaremos usar CodeBERT para realizar la tokenización del código y los embeddings de los bloques de código. Esto nos permite hacer el análisis léxico y sintáctico, además de agregar el uso de un AST para la evaluación a profundidad de la sintaxis de los códigos a revisar.

VI. Preprocesamiento de Datos

GraphCodeBERT (de ahora en adelante referenciado como CodeBERT) es un modelo pre-entrenado (PTM) para lenguajes de programación. A diferencia de los PTMs que mencionan los artículos referenciados en este artículo, CodeBERT no trata el código como una mera

secuencia de tokens, sino que reconoce y explota la estructura inherente del código, específicamente el flujo de datos (data flow). Este enfoque permite al modelo analizar la semántica del código relacionando el valor y función de las variables e interacciones que definen la lógica del programa, proporcionando una representación más contextualmente informada de la representación del código.

Para este proyecto, CodeBERT se utiliza en el procesamiento de datos en la tokenización y embedding de bloques de código para permitir la profunda evaluación del léxico y sintaxis en los códigos. La capacidad de detectar plagio depende de la habilidad del sistema para identificar similitudes funcionales y lógicas, incluso cuando el código ha sido modificado. Las representaciones de código generadas por CodeBERT permiten que los modelos de redes neuronales vean más allá de estos cambios sintácticos. Esto facilita la detección de plagio semántico, donde la lógica del código es copiada a pesar de diferencias estructurales. Por lo tanto, CodeBERT o los principios que lo sustentan son fundamentales para desarrollar un detector de plagio que pueda identificar la "intención" del código.

Tokenización

El primer paso, claro, fue cargar los datos de pares y códigos al programa seguido de la tokenización del lenguaje en los archivos. La tokenización es un proceso que se ocupa de descomponer los datos (digamos un código de 13 líneas) en unidades fáciles de “consumir”, sea por palabra o por caracter. Tras una breve investigación, se definió que crear los tokens por caracter es más eficiente cuando se trata de análisis de código ya que hace de la evaluación un proceso más flexible, adaptable a cambios y más preciso para comparar y calcular la similitud entre dos bloques de código. Este proceso inicial es de vital importancia porque nos ayuda a transformar la entrada del código fuente de un formato textual bruto a una secuencia estructurada que es manejable para el análisis computacional. Para la identificación de plagio, los algoritmos pueden comparar secuencias de tokens utilizando **métricas de similitud**. Este acercamiento es particularmente efectivo para detectar plagio literal y plagio con mínimas modificaciones, ya que este método se enfoca en secuencia de tokens en el orden que tienen o introducen solamente ligeros cambios. Aunque la tokenización es un proceso que por sí solo no es suficiente para detectar plagio con reestructuraciones del código o equivalencias semánticas, es un paso inicial para el desarrollo de herramientas que puedan hacerlo a través de un análisis más complejo como los **árboles de sintaxis abstracta (ASTs)**. La manera en la que se realizó, es la siguiente:

```
# Tokenización javalang
def tokenize_javalang(code):
    try:
        tokens =
list(javalang.tokenizer.tokenize(code
))
        return [token.value for token in
tokens]
    except:
        return []

# Tokenización y padding para LSTM
```

```
y CNN
def get_sequences(codes, tokenizer, maxlen):
    seqs = tokenizer.texts_to_sequences(codes)
    return pad_sequences(seqs, maxlen=maxlen, padding='post', truncating='post')

...

# Prepara secuencias de tokens
tokens_1 = [' '.join(tokenize_javalang(code)) for code in codes_1]
tokens_2 = [' '.join(tokenize_javalang(code)) for code in codes_2]
X1_seq = get_sequences(tokens_1, keras_tokenizer, maxlen)
X2_seq = get_sequences(tokens_2, keras_tokenizer, maxlen)
```

Embeddings

Los embeddings son representaciones densas y de baja dimensión de entidades (como palabras, tokens o fragmentos de código) que se encuentran en un espacio vectorial continuo. En este espacio, las entidades con significados o propiedades similares a estos se mapean a puntos cercanos, realizando así las comparaciones al igual que relaciones semánticas y sintácticas en el código analizado. A diferencia de representaciones dispersas o basadas en recuentos como los tokens, los embeddings permiten a las redes neuronales procesar la información más eficientemente y capturar matices más complejos del lenguaje como la semántica. La similitud entre dos fragmentos embebidos se calcula a partir de la distancia en el campo vectorial que comparten los fragmentos. Esta distancia se infiere usando una métrica de similitud llamada **distancia coseno**. La distancia coseno mide la diferencia entre dos vectores usando el ángulo entre ellos y se calcula como 1 menos la similitud cosenoidal:

$$distancia\ coseno(A, B) = 1 - \frac{A*B}{||A||\ ||B||}$$

Un modelo pre-entrenado (PTM) como CodeBERT demuestra ser una herramienta eficaz siendo que es capaz de producir embeddings que encapsulan la semántica y la lógica funcional del código. Estos embeddings permiten que un detector de plagio basado en redes neuronales identifique cuándo dos fragmentos de código tienen un comportamiento funcional idéntico o muy similar, incluso si su sintaxis es radicalmente diferente. Así, se proporciona una representación también de alto nivel que las redes neuronales pueden utilizar para aprender los patrones de plagio más allá de las similitudes superficiales del código Java.

```
def codebert_embed(code, max_length=256):
    inputs = codebert_tokenizer(
        code, return_tensors='tf', padding='max_length', truncation=True, max_length=max_length
    )
    outputs = codebert_model(**inputs)
    emb = outputs.last_hidden_state[:, 0, :]  # [CLS] token embedding
    return emb.numpy().squeeze()  # (768,)
```

```
...

# Prepara embeddings CodeBERT (puede tardar)
print("Generando embeddings de CodeBERT (esto puede tardar unos minutos)...")
X1_cb = np.stack([codebert_embed(code) for code in codes_1])
X2_cb = np.stack([codebert_embed(code) for code in codes_2])
print("Listo.")
```

Etiquetas

Después, asignamos las etiquetas correspondientes que se encuentran en el archivo labels.csv a los datos dentro de las carpetas de versiones. Las etiquetas son la diferencia entre el aprendizaje supervisado y no supervisado. Estas ofrecen una línea base para reconocer los patrones directos entre plagio y no plagio. El objetivo de la red es ajustar sus parámetros internos (pesos y sesgos) para minimizar la diferencia entre sus predicciones y las etiquetas de verdad fundamental. Mientras más claras y definidas sean las etiquetas en los datos mejor aprenderá el modelo. En este caso contamos con las etiquetas **'sub1'** y **'sub2'** que representan los 2 fragmentos de código que se van a comparar y la etiqueta **'veredict'** que establece si el resultado de esa comparación es plagio o no plagio en valor binario.

```
# --- Etiquetas ---
data_dir = "./conplag_dataset/versions/"
labels_csv = os.path.join(data_dir, "labels.csv")
labels_df = pd.read_csv(labels_csv)
# Asegúrate de que las columnas 'sub1', 'sub2' y 'verdict' existen en el CSV
labels_dict = {(str(row['sub1']), str(row['sub2'])): row['verdict'] for _, row in labels_df.iterrows()}

y = []
for name1, name2 in zip(nombres_1, nombres_2):
    label = labels_dict.get((name1, name2), labels_dict.get((name2, name1), None))
    if label is None:
        print(f"Warning: No label found for pair ({name1}, {name2})")
        label = 0
    y.append(label)
y = np.array(y)
```

División del Dataset

Finalmente en el paso de preprocesamiento de datos, se necesita hacer la división del dataset. Esto consiste en dividir el conjunto de datos total en subconjuntos para tareas diferentes, principalmente aquellas de entrenamiento, validación y prueba. Se realiza para asegurar que el modelo sea robusto y generalice adecuadamente a datos no vistos buscando evitar el fenómeno de 'overfitting' o sobreajuste, donde el modelo no aprende los patrones sino que se los memoriza y al enfrentarse con datos que nunca ha visto antes desconoce la respuesta. Para este proyecto particular, definimos las proporciones de datos como 70% dedicado al entrenamiento del modelo, 20% para probar el modelo después de su entrenamiento y 10% para la validación. Esta proporción 70-20-10 asegura tener la

mayor cantidad de datos para el entrenamiento sin olvidar aquellos que se necesitan para la prueba y validación.

El **conjunto de entrenamiento** o 'training' constituye la porción más grande del dataset y es la única que la red neuronal conoce durante la fase de aprendizaje. El modelo lo utiliza para ajustar sus parámetros de predicción de acuerdo a lo que necesita para entender y asignar los pesos y sesgos correspondientes a cada una de las clasificaciones.

El **conjunto de validación** se utiliza para definir los hiperparámetros del modelo, que son los datos como la tasa de aprendizaje, el número de capas que requiere el modelo o el tamaño del batch a procesar. Todo esto permite realizar una evaluación del rendimiento del modelo en lo que está entrenando y tomar decisiones a partir de ello.

El **conjunto de prueba** o 'test' no se le muestra ni presenta al modelo durante la fase de entrenamiento, de manera que desconoce lo que se encuentra en los datos que aquí se encuentran. Es la prueba final del modelo, sirviendo como medida fiable de la capacidad de generalización y predicción que tiene el modelo una vez que el entrenamiento y desarrollo ha terminado. Esto garantiza que el modelo no haya sido influenciado por los datos dentro del conjunto de prueba, lo cual proporciona una métrica de rendimiento completamente objetiva del modelo.

```
from sklearn.model_selection import train_test_split
```

```
 # Split 80% train/val, 20% test
X1_seq_temp, X1_seq_test,
X2_seq_temp, X2_seq_test,
X1_cb_temp, X1_cb_test,
X2_cb_temp, X2_cb_test, y_temp,
y_test = train_test_split(
    X1_seq, X2_seq, X1_cb, X2_cb, y,
test_size=0.2, random_state=42,
stratify=y
 )

# # Split de validación (10% del total,
es 12.5% de train/val)
X1_seq_train, X1_seq_val,
X2_seq_train, X2_seq_val,
X1_cb_train, X1_cb_val, X2_cb_train,
X2_cb_val, y_train, y_val =
train_test_split(
    X1_seq_temp, X2_seq_temp,
X1_cb_temp, X2_cb_temp, y_temp,
test_size=0.125, random_state=42,
stratify=y_temp
 )
```

Esta división nos deja con la siguiente asignación de datos:

```
Entrenamiento (original): Plagio = 176, No Plagio = 461, Total = 637
Validación (original): Plagio = 25, No Plagio = 66, Total = 91
Test (original): Plagio = 50, No Plagio = 133, Total = 183
```

VII. Modelo

CNN

Las **Redes Neuronales Convolucionales (CNNs)** son un tipo de red neuronal artificial especialmente diseñada para el procesamiento de datos visuales como imágenes y vídeos. Son ideales para tareas de reconocimiento de patrones y objetos, como la clasificación, detección y segmentación de imágenes. Tiene una arquitectura fácilmente identificable. Se

caracteriza por el uso de capas convolucionales (de aquí su nombre) que aplican filtros llamados kernels a regiones pequeñas de la entrada, de esta manera aprenden patrones jerárquicos y características especiales de lo que se está analizando eficientemente. Después de la convolución, entran las capas de 'pooling' o agrupación que reducen el tamaño de los mapas de características. Esto ayuda a evitar el sobreajuste del modelo, que generalice más adecuadamente y hacer que la red sea más robusta frente a variaciones en la entrada del programa.
Algunos de los beneficios de usar una CNN son que son muy eficientes en identificar patrones tanto visuales como textuales, además de su rápido procesamiento de grandes volúmenes de datos. No requiere de extracción manual para leer y analizar los datos y tiene una excelente versatilidad de usos para tareas tanto con imágenes, videos y textos. Sin embargo, tiene una restricción particular que requiere el uso de otros tipos de redes neuronales para este proyecto que es que sufre de dificultades para identificar semántica compleja y, por lo tanto, pueden no ser las más adecuadas para capturar la semántica profunda o el flujo de datos. Pero para complementar esta debilidad de las CNN, añadimos el apoyo de CodeBERT con embeddings para proveer representaciones de entrada más fáciles de consumir.

```
# --- CNN branch ---
def cnn_branch():
    inp = Input(shape=(maxlen,))
    x = Embedding(vocab_size,
embed_dim, mask_zero=True)(inp)
    #x = Conv1D(32, 5,
activation='relu',
kernel_regularizer=tf.keras.regularize
rs.l2(0.01))(x)
    x = Conv1D(16, 3,
activation='relu')(x)
    x = GlobalMaxPooling1D()(x)
    return Model(inp, x,
name="CNN_Branch")
```

RNN y LSTM

Las **Redes Neuronales Recurrentes (RNNs)** están específicamente diseñadas para el procesamiento de datos secuenciales, por ejemplo, texto, series de tiempo o código, que es lo que nos interesa bajo este contexto. Algo particular y muy útil de las RNNs es que poseen una "memoria interna". Esto les permite mantener un estado oculto que puede capturar información de pasos previos en una secuencia. En esto se asimila y busca imitar la forma en que los humanos realizan conversiones de datos secuenciales, como la traducción de texto de un idioma a otro. De sus potenciales usos que se aprovecharon para el desarrollo de este proyecto se encuentran:

El **análisis de secuencias de tokens**, capturando patrones representativos dentro del léxico y la sintaxis del código bajo evaluación. Se utiliza para la detección del plagio literal y el plagio con mínimas modificaciones, debido a que la secuencia de tokens es similar y fácil de comparar.

La **generación de embeddings de secuencia** para fragmentos de código de manera que se usen estos vectores para encapsular información secuencial y mejorar el entendimiento y detección de la similitud en semántica.

Finalmente, la **detección de patrones de código** que se caracteriza por identificar estructuras y construcciones de código específicas o fragmentos de algoritmos que se pueden identificar como plagio incluso después de cambios menores.

Dentro de la clase de RNNs, se encuentra un tipo más especializado llamado **Redes Neuronales de Memoria a Largo Corto Plazo (LSTM)** que es el tipo de RNN que se utilizó en este detector de plagio. Lo especial de las LSTM es su excelente manejo de memoria y flujo de información, ya que cuenta con unas compuertas que regulan el flujo de información hacia y desde una celda de estado, permitiendo a la red aprender cuándo recordar u olvidar información de pasos anteriores en la secuencia, lo que confiere una capacidad superior para capturar dependencias a largo alcance.

```
# --- LSTM branch ---
def lstm_branch():
    inp = Input(shape=(maxlen,))
    x = Embedding(vocab_size,
embed_dim, mask_zero=True)(inp)
    x = LSTM(16)(x)
    return Model(inp, x,
name="LSTM_Branch")
```

PTM

Los Modelos Pre Entrenados (o PTMs) son modelos de Machine Learning que han sido previamente entrenados con grandes conjuntos de datos. Esto quiere decir que el modelo ya ha probado ser adecuado para casos similares para ser personalizados para una tarea específica. En el caso de este proyecto, el uso de CodeBERT como capa de entrenamiento paralela en la arquitectura de red neuronal. Su objetivo particular es **generar representaciones contextualizadas y semánticamente completas** (embeddings) del código fuente de entrada.
Como capa paralela a las dos anteriores, CodeBERT recibe el código tokenizado y crea los embeddings para encapsular información del flujo de datos y la funcionalidad subyacente del código. Y al final, los embeddings son concatenados con las características extraídas de las ramas paralelas de la red y finalmente alimentados a la capa densa posterior para su clasificación final. Este proceso permite que el modelo se aproveche de la comprensión y profundo análisis de CodeBERT mientras las demás capas se utilizan para integrar y refinar la información con características específicas, permitiendo la capacidad del sistema para detectar plagios con cambios semánticos como lo son el plagio con reestructuración compleja y equivalencias semánticas.

Siamese Model - Final Model

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 256)	0
input_layer_5 (InputLayer)	(None, 256)	0
input_layer_7 (InputLayer)	(None, 768)	0
input_layer_4 (InputLayer)	(None, 256)	0
input_layer_6 (InputLayer)	(None, 256)	0
input_layer_8 (InputLayer)	(None, 768)	0
LSTM_Branch (Functional)	(None, 16)	1,289,280
CNN_Branch (Functional)	(None, 16)	1,286,160
CodeBERT_Branch (Functional)	(None, 32)	24,608
concatenate (Concatenate)	(None, 64)	0
concatenate_1 (Concatenate)	(None, 64)	0
lambda (Lambda)	(None, 64)	0
dense_1 (Dense)	(None, 32)	2,080
dense_2 (Dense)	(None, 16)	528
dense_3 (Dense)	(None, 1)	17

El modelo que ha sido explicado en este artículo entra en la categoría de una SNN o Red Neuronal Siamés. Esta es una arquitectura diseñada para profundizar en métricas de similitud de entre dos o más entradas. Consiste del uso de dos o más ramas paralelas procesando cada una de las entradas y estas comparten los pesos de las ramas. Esta configuración permite que la red aprenda a extraer características comparables de ambas entradas, de modo que la distancia o relación entre las representaciones resultantes indique el grado de similitud entre ellas.

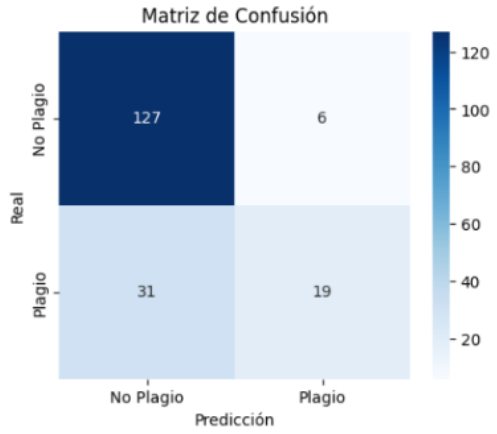
En este proyecto, tenemos tres ramas que se ejecutan paralelamente. Una rama LSTM para dependencias secuenciales, una rama CNN para patrones locales jerárquicos, y una rama CodeBERT que procesa embeddings pre-calculados para capturar la semántica profunda. Cada uno de los pares de código en los archivos de train_pair y test_pair son alimentados simultáneamente a estas ramas que comparten los mismos pesos. De esta manera, se generan tres embeddings para cada una de las dos entradas (seis embeddings totales entre ambas entradas en cada par). Después de la ejecución paralela de las tres ramas, se juntan cual trenza a través de una concatenación de los embeddings de cada código. Finalmente, la relación entre ellos y la similitud total se infiere calculando la diferencia absoluta entre los vectores de sus características. Este vector que representa la diferencia entre ellos es alimentado a una red densa para una clasificación binaria representando la decisión de plagio o no plagio. Esta decisión final es lo que permite al modelo aprender a cuantificar la similitud entre códigos.

VIII. Resultados

En el desarrollo de este proyecto se obtuvieron los siguientes resultados:

	Plagio	No Plagio
Accuracy	0.80	0.76
Recall	0.95	0.38
F1	0.87	0.51

.

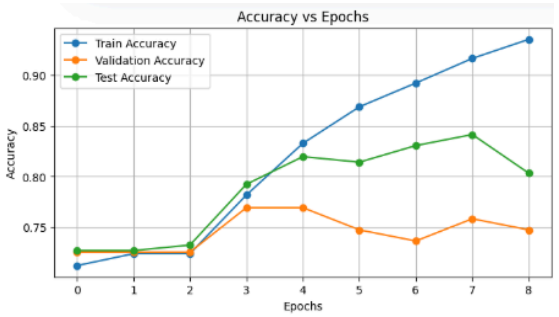


Bajo estas métricas se da a entender que el modelo desarrollado tiene un sesgo hacia el no plagio, habiéndose equivocado 31/50 veces prediciendo que el plagio no es plagio.

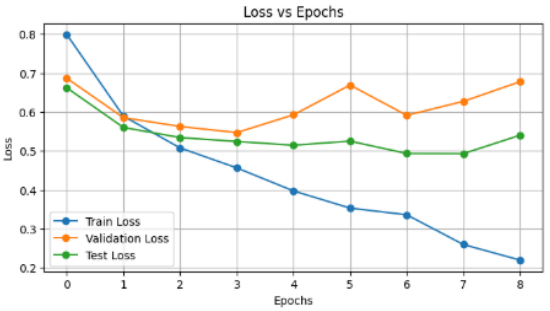
Según el recall, cuando predice que algo no es plagio, sólo está un 38% seguro de su respuesta. A diferencia del 95% de seguridad que tiene cuando menciona que algo es plagio.

Pero, ¿cómo es esto posible si el cálculo de precisión dice que es un 76-80%? La respuesta es muy sencilla. Esos no son los valores reales. O al menos no son objetivos respecto al modelo.
La explicación detrás de esto es el 'overfitting' o sobreajuste de datos. Este es un fenómeno muy común en las redes neuronales, donde el modelo memoriza las características específicas de los patrones en el conjunto de datos de entrenamiento hasta el punto donde pierde su capacidad de generalizar y predecir adecuadamente datos nuevos y no vistos. Esto quiere decir que si no se pide la predicción de una de las entradas de entrenamiento, la posibilidad de que prediga mal es muy alta. Esto presenta un rendimiento excelente en los datos de entrenamiento, pero se vuelve muy pobre en los conjuntos de datos de validación y prueba. Esto se puede notar a través de los siguientes gráficos:

PRECISIÓN:



PÉRDIDA



En estos gráficos se evidencia la diferencia entre la línea de entrenamiento y las de prueba y validación que decaen mucho en su rendimiento.

Este fenómeno ocurre principalmente por dos razones codependientes. Primero, la complejidad excesiva del modelo en relación con la cantidad. Un modelo con demasiados parámetros, por ejemplo una red neuronal con demasiadas capas o neuronas como la que fue presentada anteriormente tiene la capacidad de ajustarse a cualquier detalle, incluso el ruido, presente en el training set. Nuestro dataset constaba de un conjunto de 5000 archivos totales, un número bastante bajo, sin embargo, se desarrolló una red siamés con 3 ramas diferentes y más capas densas al final para concatenar todo. En otras palabras, se descubrió que era como tratar

de cavar un hoyo para una macetita con una excavadora de construcción.

Y en segundo lugar, una cantidad insuficiente o calidad deficiente de los datos de entrenamiento. Si el dataset de entrenamiento no es lo suficientemente grande respecto de la distribución real de los datos, el modelo no puede aprender patrones generalizables y, en cambio, se enfoca en las características específicas y posiblemente idiosincrásicas del conjunto limitado que tiene a su disposición. Respecto a nuestro conjunto de datos, contábamos con un desbalance absoluto de los datos, con 600 ejemplos de No plagio y 200 de Plagio.

Es por esto que de las áreas a mejorar posteriormente en el desarrollo de este trabajo tenemos la simplificación de la arquitectura del modelo y el balanceo del dataset a usar para que la diferencia entre la cantidad de datos no sea tan abismal.

IX. Mejoras

Tras la revisión del uso del modelo en este proyecto, se concluyó por los motivos de la sección anterior, que probablemente no fue la mejor de las opciones. En primer lugar, el tamaño del dataset y su proporción respecto a las dos categorías de 'plagio' y 'no plagio'.

No Plagio	600 muestras
Plagio	200 muestras

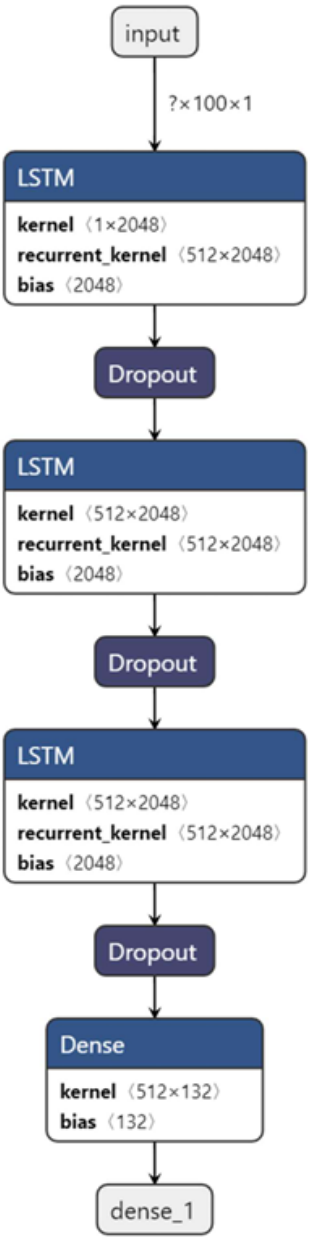
Siendo que la proporción del dataset es 3:1 entre 'no plagio' y 'plagio', el modelo sufrió de un sesgo dirigido a favorecer una clase sobre otra. La analogía propuesta también es que al pasar los pocos datos por tres ramas al mismo tiempo fue como hacer una trenza con muy poco cabello de manera que los datos se llegaron a perder todavía más.

Tomando en cuenta el dataset, el modelo propuesto por S. Surendran [4] no fue la mejor decisión, por lo que no lo volvería a elegir para este particular dataset. Un aprendizaje muy importante de este trabajo fue que se debe tomar la decisión de cuál arquitectura usar, comparar los datos para asegurarse que son similares y de las mismas proporciones para un resultado similar. Justo tomando en cuenta ese aprendizaje, volví a revisar los artículos elegidos para este trabajo y me topé nuevamente con A. Eppa [6]. Revisé primero su dataset y su división era la siguiente:

No. of Plagiarized Pairs	519
No. of Non-plagiarized Pairs	722

En su cantidad, es mucho más cercano a los números que tenemos, la única dificultad siendo que tendría que duplicar la cantidad de muestras de 'Plagio'. Pero tengo considerado juntar con otros conjuntos de datos para completar los números. Claro, manteniendo también la proporcionalidad de ambos datasets según su sección (train o test).
Esa sería la primera acción.
Posteriormente, la arquitectura que se tomó en el estudio de A. Eppa [6], a diferencia del propuesto por Surendran [4], no es siamés sino que depende totalmente

en RNNs como lo muestra la imagen más abajo:



Cuenta con 3 capas LSTM y sale por una densa. Hay un dropout después de cada una debido a la alta cantidad de filtros que tiene cada una, para que no se sobrecargue. Además de que ha usado un PTM llamado MOSS, aunque yo estaría usando CodeBERT aún, para hacer el embedding de antemano para la evaluación de la semántica del código. Esto, sin embargo, también tiene algunas desventajas como lo son:

1. Independencia del Aprendizaje de Embeddings y de la Tarea. Si bien los embeddings generados por modelos pre entrenados como CodeBERT son semánticamente ricos, no se fine-tunean para una tarea en espacífico durante el entrenamiento de la RNN. Esto significa que las representaciones vectoriales podrían no capturar óptimamente las características más discriminatorias para diferenciar el plagio. La RNN solo aprenderá a clasificar basándose en las propiedades que el embedding ya contiene, sin la capacidad de retro propagar gradientes para mejorar la calidad del embedding para la tarea de plagio.

2. Persistencia del Sesgo Lineal de las RNNs. Aunque los embeddings pre-calculados pueden incorporar información estructural o semántica de grafos, la RNN sigue procesando esta información en una secuencia lineal. Esto significa que cualquier relación no secuencial inherente al código, como las interconexiones en un Árbol de Sintaxis Abstracta (AST) o las relaciones de un Grafo de Flujo de Datos (DFG) podría no ser capturada eficazmente por la RNN.

3. Gestión de Dependencias a Muy Largo Plazo. A pesar de que las LSTMs son superiores a las RNNs básicas para capturar dependencias a largo plazo, aún pueden encontrar dificultades con secuencias extremadamente largas o relaciones muy dispersas en el código. Si los embeddings representan archivos de código completos o fragmentos muy extensos, la capacidad de la RNN para mantener un estado coherente y detectar

dependencias cruciales a lo largo de toda la secuencia podría ser limitada.

4. Menor Flexibilidad y Adaptabilidad: Al desacoplar la generación de embeddings del entrenamiento del clasificador, el modelo pierde flexibilidad. Cualquier cambio en las estrategias de embedding o cualquier descubrimiento sobre qué características son más relevantes para el plagio requeriría una regeneración completa de los embeddings del dataset, en lugar de permitir que la red ajuste de forma end-to-end las representaciones. Esto puede hacer que el modelo sea menos adaptable a diferentes dominios o tipos de código si los embeddings pre-calculados no son universalmente óptimos para todas las variaciones de plagio.

Para resolver la mayoría de estos problemas, se recomienda en el estudio, que "las incrustaciones de la última capa oculta de la red optimizada se usen como características profundas para representar el código fuente. La tercera capa oculta LSTM tiene 512 unidades, por lo que el vector que representa las características profundas contiene 512 elementos. Tras obtener las características profundas de todos los pares del conjunto de datos creado previamente, se realizó un análisis de componentes principales [9] para la reducción de dimensionalidad con 250 componentes principales. Los vectores de longitud 512 se proyectaron sobre este subespacio principal." [6] A. Eppa 2022. A través del entrenamiento anterior no supervisado por parte de CodeBERT, y el reentrenamiento posterior con las LSTM, se puede apoyar a normalizar la información.

También, podría usarse nuevamente el Concatenate hecho en el modelo actual para que las capas densas subsiguientes aprendan relaciones de similitud más complejas.
Otra opción es que en lugar de una función fija, se podría diseñar una pequeña sub-red con una o dos capas densas que tome los dos vectores y aprenda la métrica de similitud óptima directamente, lo que proporcionaría mayor flexibilidad.

Más importante, se debe evitar el overfitting. Esto se puede evitar manteniendo el Early stopping que esté basada en el rendimiento en el conjunto de validación para detener el entrenamiento antes de que el modelo comience a sobreajustarse. Igualmente, podría generar más ejemplos de plagio aplicando diversas transformaciones al código original para incrementar la diversidad y el tamaño del conjunto de entrenamiento. Solucionando dos problemas de un tiro.

X. Referencias

[1] Statista, "Artificial Intelligence (AI) market size worldwide 2019–2030", Statista, 2024. [En línea]. Disponible: https://www.statista.com/statistics/730835/global-ai-market-size . [Accedido: 3-jun-2025].

[2] "Coding Integrity Unveiled: Exploring the Pros and Cons of Detecting Plagiarism in Programming Assignments Using Copyleaks," Journal of Computing Sciences in Colleges, vol. 39, no. 6, pp. 104, Apr. 2024. Publicado: 16 May 2024. [En línea]. Disponible: https://dl.acm.org/doi/10.5555/3665464.3665471 . [Accedido: 3-jun-2025].

[3] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy y M. Ekhtiarzadeh, “A systematic literature review on source code similarity measurement and clone detection: techniques, applications, and challenges,” arXiv:2306.16171 [cs.SE], Jun. 2023. [En línea]. Disponible: https://arxiv.org/abs/2306.16171 . [Accedido: 3-jun-2025].

[4] O. Karnalim, S. Budi, H. Toba, and M. Joy, “Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation,” *Informatics in Education*, vol. 18, no. 2, pp. 321–344, 2019.

[5] D. Guo *et al.*, “GRAPHCODEBERT: PRE-TRAINING CODE REPRESENTATIONS WITH DATA FLOW,” presented at the ICLR 2021, Sep. 2021.

[6] A. Eppa, “Source Code Plagiarism Detection: A Machine Intelligence Approach,” in *2022 IEEE Fourth International Conference on Advances in Electronics, Computers and Communications (ICAECC)*, 2022, pp. 1–5.

[7] S. Surendran, “Plagiarism Detection in Source Code using Machine Learning,” Master’s Thesis, University of Windsor, Windsor, ON, Canada, 2024.

[8] Slobodkin, “ConPlag: a Dataset of Programming Contest Plagiarism in Java”, Sadovnikov, 2022. [En línea]. Disponible: https://zenodo.org/records/7332790 . [Accedido: 3-jun-2025].