Artículo Original / Original Research

Ciencias Matemáticas, Vol. 38, No. 2, 2024, Pag. 47-56
Recibido (*Received*): 03-04-2025, Revisado (*Reviewed*): 12-05-2025
Aceptado (*Accepted*): 20-05-2025, En línea (*Online*): 06-08-2025

Clasificación de similitud entre códigos de C#
Classification of similarity between C# codes

María de Lourdes Choy Fernández[1], Rodrigo García Gómez[2], Alberto Fernández Oliva*[3]

Resumen El fraude es un problema recurrente en la educación. Los profesores luchan constantemente contra él, tanto en exámenes como en proyectos y evaluaciones teóricas. El sistema de evaluación de la asignatura de Programación en la carrera de Ciencia de la Computación de la Universidad de La Habana, depende, en gran medida, de la realización de proyectos evaluativos por parte de los estudiantes. Contar con herramientas computacionales que permitan de forma automática detectar plagios en tales proyectos será de gran utilidad en el ámbito académico. Esta investigación aborda el problema de la detección de similitudes en el código fuente de proyectos de C#. El proceso comienza con la extracción de árboles de sintaxis abstracta (AST), para establecer una representación estructural del código. A partir de los AST, se extraen características que describen elementos esenciales del mismo y dichas características se agrupan en vectores que se utilizan como entrada de una red neuronal siamesa. Dicha red se entrena como un modelo clasificador de similitudes capaz de detectar parejas de códigos de C# que han sido clonados.

Palabras Clave: árbol de sintaxis abstracta, entrenamiento, gramática, red neuronal siamesa, similitud.

Abstract *Cheating is a recurring problem in education. Professors constantly fight against it, both in exams and in theoretical projects and assessments. The assessment system for the Programming course in the Computer Science program at the University of Havana relies largely on students completing assessment projects. Having computational tools that automatically detect plagiarism in such projects will be very useful in the academic field. This research addresses the problem of detecting similarities in the source code of C# projects. The process begins with the extraction of abstract syntax trees (ASTs) to establish a structural representation of the code. From the ASTs, features that describe essential elements of the code are extracted, and these features are grouped into vectors that are used as input to a siamese neural network. This network is trained as a similarity classifier model capable of detecting pairs of cloned C# code.*

Keywords: *abstract syntax tree, training, grammar, siamese neural network, similarity.*

Mathematics Subject Classification: 68T05, 68T07, 68N19, 68N15, 68Q32.

[1]Departamento de Computación, Facultad de Matemática y Computación, Universidad de La Habana, Cuba. Email: mlchony7@gmail.com.
[2]Departamento de Computación, Facultad de Matemática y Computación, Universidad de La Habana, Cuba. Email: rodrigo.garcia21111@gmail.com.
[3]Departamento de Computación, Facultad de Matemática y Computación, Universidad de La Habana, Cuba. Email: afoliva55@gmail.com.
*Autor para Correspondencia (*Corresponding Author*)

Editado por (*Edited by*): Damian Valdés Santiago, Facultad de Matemática y Computación, Universidad de La Habana, La Habana, Cuba.

Citar como: Choy Fernández, M. L., García Gómez, R. & Fernández Oliva, A. (2024). Clasificación de similitud entre códigos de C#. *Ciencias Matemáticas*, 38(2), 47-56. DOI: https://doi.org/10.5281/zenodo.16757265. Recuperado a partir de https://revistas.uh.cu/rcm/article/view/11135.

Introducción

El análisis de similitud de código en proyectos computacionales se centra en la comparación de fragmentos de este para identificar similitudes y diferencias entre ellos. Este enfoque se utiliza en la detección de plagios y en la revisión de código, en sentido general. Esta investigación surge de la necesidad de dotar a las instituciones educativas de herramientas que permitan un análisis preciso del código de forma tal que facilite la detección de plagios en entornos académicos y, de igual forma, pretende contribuir al entendimiento teórico del problema y ofrecer una base para futuras investigaciones en el área.

En el presente trabajo, se combinan técnicas de aprendizaje automático (AA) [10] y técnicas de extracción de árboles de sintaxis abstracta (AST, por sus siglas en inglés) [5]. En este sentido, la propuesta se basa en la extracción de dichos árboles a partir de códigos de C# para, posteriormente, convertirlos en vectores de características y utilizarlos como entrada a un modelo de red neuronal siamesa [13, 8].

Las redes siamesas constan de dos ramas idénticas que comparten los mismos parámetros. Cada rama procesa una de las entradas y genera una representación vectorial de alto nivel, la cual captura las características abstractas, o esenciales, de los datos de entrada.

En investigaciones tales como [2], esta arquitectura se utiliza junto con redes convolucionales (CNN) [14] para analizar similitudes semánticas, donde se reportan precisiones de hasta un 90 %. El modelo asigna valores entre 0 y 1 a parejas de códigos según su similitud (un valor más cercano a 1 indica mayor similitud) y se utilizará en un software clasificador.

La red siamesa se entrena utilizando un conjunto de códigos generados por inteligencias artificiales generativas y además, por un conjunto compuesto por copias manuales de proyectos de C#. De este modo se obtendrá una herramienta práctica, que se podrá integrar en entornos de desarrollo reales.

A diferencia de otras herramientas de detección de plagio (Tabla 1), esta propuesta se destaca por la modificación y actualización de la gramática de C#. Esto permite un análisis actualizado del código fuente, pues se adapta a las últimas características y sintaxis del lenguaje.

La validación de la herramienta se llevará a cabo en escenarios prácticos, como la detección de plagios en tareas de programación de código. Esto no solo demostrará la efectividad de las técnicas implementadas, sino también garantizará su utilidad para los usuarios finales.

Relevancia del estudio

La relevancia de esta investigación radica en su potencial para transformar la forma en que se evalúa la integridad académica en el campo de la programación. Al desarrollar una herramienta que combina análisis de similitud de código con técnicas de aprendizaje automático, este estudio no solo mejorará la detección de plagio, sino que también fomentará prácticas éticas en el desarrollo de software. Además, la actualización de la gramática de C# en la propuesta permitirá un análisis más actual, adaptándose a las evoluciones del lenguaje. Esta investigación impacta significativamente en el ámbito educativo en lo referido a la Ciencia de la Computación, promoviendo la originalidad y la innovación entre los estudiantes y proporcionando a los educadores herramientas más efectivas para evaluar los proyectos evaluativos de los alumnos.

1. Propuesta

1.1 Extracción de *features* del AST

En el análisis de similitud de código y detección de patrones, es necesario extraer las características más relevantes que capturen la estructura y el comportamiento del código para permitir que los modelos de análisis puedan, posteriormente, tanto identificar similitudes, así como detectar patrones.

Estas características, se conocen como *features* y representan los aspectos más importantes de los datos para analizar y comparar fragmentos de código de manera eficiente. Una de las técnicas más comunes para extraer tales características es el uso de Árboles de Sintaxis Abstracta (AST, por sus siglas en ingl'es), a partir de los cuales se representa, en forma de árbol, la estructura sintáctica de un fragmento de código.

1.1.1 Modificaciones a la Gramática de ANTLR para Soporte de C# Actual

La herramienta ANTLR (*Another Tool for Language Recognition*) [24] cuenta con una gramática basada en C# 6.0. Esta gramática presentaba limitaciones significativas al trabajar con el código moderno (o sea, código que utilice versiones recientes del lenguaje). Por ejemplo, se generaban errores durante el análisis sintáctico y AST incompletos. Por tal motivo, se decide trabajar con una versión más actualizada del lenguaje C#, la versión 12.0 (introducida en noviembre de 2023), para reflejar las características que fueron introducidas en versiones posteriores de este lenguaje de programación.

1.1.2 Extraccion del AST

ANTLR convierte una gramática de lenguaje en un código a partir del cual se puede generar un árbol de sintaxis. A continuación, se describen los pasos que se siguen para generar un AST a partir de un código fuente, que utiliza un proceso de análisis léxico y sintáctico.

1. **Definición de la gramática:** primero, se define la gramática del lenguaje en un archivo `.g4` que expresa la estructura de un lenguaje mediante una serie de reglas léxicas y sintácticas. Es el núcleo de la herramienta ANTLR y sirve como una especificación formal del lenguaje de programación que se desea analizar o procesar. Estas reglas se emplean para descomponer el texto de entrada en componentes que ANTLR puede reconocer y procesar.

    - Las **reglas léxicas (*tokens*)** se encuentran en el archivo *CSharpLexer.g4*. Estas se encargan de identificar los componentes más simples del lenguaje: palabras clave, identificadores, operadores, números, comillas, entre otros. Los *tokens* son las unidades mínimas de información con las que trabaja el analizador léxico, y cada uno de ellos se representa como una secuencia de caracteres que ANTLR puede reconocer.
    - Las **reglas sintácticas** se encuentran en el archivo *CSharpParser.g4*. Describen cómo los *tokens* del *lexer* se combinan en estructuras válidas dentro del lenguaje. Definen cómo los *tokens* se agrupan y organizan en expresiones, sentencias, funciones o cualquier otra estructura del lenguaje.

      Por ejemplo, una regla sintáctica puede definir cómo se estructura una asignación de variable, estableciendo que una asignación se compone de un identificador, seguido de un operador de asignación y una expresión que puede ser un valor o una operación.

Tabla 1. Estudio sobre la detención de similitudes de código utilizando técnicas de aprendizaje automático, extracción de AST y otros modelos, en diferentes lenguajes de programación [*Study on detecting code similarities using machine learning techniques, AST extraction, and other models, in different programming languages*].

Artículo científico	¿AA?	¿AST?	Modelo de AA	Exactitud	Precisión	Recobrado	Lenguaje
Abba et al. [2]	Sí	Sí	Red neuronal siamesa y convolucional	67-88 %	90 %	95 %	C++
Wang et al. [29]	Sí	Sí	Agrupamiento no supervisado	0,85 – 0,90 %			Python
Hoq et al. [12]	Sí	Sí	Máquina de vectores de soporte, *Gradient Boosting* Extremo y code2vec	97 %	99 %	97 %	Python
Wang et al. [30]	Sí	Sí	Red neuronal de grafos	96 %	94 %	95 %	Java
Yang et al. [31]	Sí	Sí	Red neuronal siamesa				Binary
Tankala et al. [28]	Sí	Sí	Red neuronal siamesa y recurrente		100 %	99 %	Java
Mehrotra et al. [19]	Sí	Sí	Red neuronal de grafos				Java
Baxter et al. [5]	No	Sí	Ninguna				Java, C
Aiken [4]	No	No	Normalización de código				Varios
Benedikt et al. [6]	No	Sí	Tokenización				C# 6.0, Java

2. **Generación del *lexer* y el *parser*:** después de definir la gramática en un archivo `.g4`, ANTLR genera automáticamente las clases necesarias para implementar el *lexer* (analizador léxico) y el *parser* (analizador sintáctico) en Python.

   Como parte de este proceso, ANTLR también genera un árbol de sintaxis básico (*parse tree*), el cual representa la estructura jerárquica del código fuente que se analiza, incluyendo todos los *tokens* y reglas gramaticales definidos en la gramática. Este árbol es fundamental para analizar el flujo y la estructura del programa, sirviendo como base para la creación de un AST. En este empeño, se utilizó ANTLR versión 4.13.2, el cual procesa los archivos `.g4` y genera las clases necesarias en Python.

3. **Análisis del código o texto de entrada:** el *lexer* y *parser* se utilizan para procesar el código fuente o texto de entrada. Este análisis comienza con el *lexer*, que divide el texto en *tokens* que siguen las reglas léxicas que se definen en la gramática. Luego, el *parser* organiza estos *tokens* en una estructura jerárquica que refleja la gramática sintáctica del lenguaje.

   Este proceso de análisis genera un árbol de sintaxis, también conocido como árbol de derivación, que representa la estructura jerárquica del código según las reglas definidas en el archivo `.g4`. Cada nodo del árbol corresponde a una regla de la gramática, lo que permite una representación estructurada del código.

4. **Creación del AST:** ANTLR no solo genera el árbol de sintaxis básico, sino que también crea un AST. A diferencia del árbol de sintaxis, el AST elimina detalles como los símbolos delimitadores o ciertos *tokens* que no son necesarios para el análisis lógico del programa.

   El AST conserva la estructura lógica y semántica del código y organiza las relaciones jerárquicas entre elementos tales como: expresiones, declaraciones y bloques de control. Esto permite realizar distintos tipos de análisis a partir de los cuales se logra, por ejemplo, la extracción de características o la transformación del código.

5. **Recorridos y transformaciones en el AST:** después de construir el AST, se utilizó el *listener CSharpParserListener* de ANTLR para recorrer el árbol y extraer información relevante del código fuente, sobre la cual se dan detalles en la próxima sección. El *listener* recorre el AST y trata cada nodo de acuerdo con la lógica que se define en el código.

1.1.3 Extracción de *features*

Se implementó una clase llamada *FeatureExtractorListener* que extiende la funcionalidad de *CSharpParserListener* para analizar el código fuente. Esta clase permite identificar determinados elementos del código, entre los que se encuentran:

- **Estructura del código:** profundidad del AST y número total de nodos.
- **Variables y datos:** cantidad de variables, constantes, tuplas, listas y diccionarios.
- **Organización del código:** número de métodos, clases, interfaces, espacios de nombres y sus respectivos nombres.
- **Flujo de control:** presencia de estructuras de control como *if*, *switch*, bucles y bloques *try-catch*.

- **Accesibilidad y modificadores:** conteo de modificadores de acceso (público, privado, protegido, etc.) y modificadores específicos *(readonly, virtual, async, etc.)*.
- **Uso de librerías y LINQ:** llamadas a bibliotecas comunes como *Console* y *Math*, así como el uso de consultas LINQ.
- **Características adicionales:** uso de expresiones *lambda*, métodos *get/set*, enumeraciones y delegados.

El objetivo es utilizar estas características como entrada para un modelo de aprendizaje automático (como una red neuronal) a partir del cual se pueda analizar el código. El siguiente paso será la preparación de un conjunto de entrenamiento (*dataset*) para este modelo.

1.1.4 Preparación del conjunto de entrenamiento

Para analizar la similitud de código, los nombres de variables y métodos se convierten en representaciones numéricas llamadas *embeddings* usando *wordvVec*, una técnica de procesamiento del lenguaje natural. Este proceso consta de:

1. **Extracción de identificadores:** se extraen nombres de variables, métodos, clases, etc., del código fuente.
2. **Entrenamiento de *word2vec*:** un modelo *word2vec* [22] se entrena con un conjunto de estos identificadores, aprendiendo las relaciones semánticas entre ellos.
3. **Conversión a *embeddings*:** los identificadores se convierten en vectores (*embeddings*), promediando vectores de características. A partir de esto, se asegura la misma dimensión y se captura su semántica y contexto.

Estos vectores se utilizan como entrada para un modelo de aprendizaje automático que permite analizar la similitud del código.

1.1.5 Vectores de características

Para preparar el conjunto de entrenamiento, los *features* que se extraen del AST se transforman en vectores de características numéricas (Figura 1), lo cual permite que un modelo de aprendizaje los reciba como entrada.

Para transformar los nombres de variables, nombres de clases, nombres de interfaces y otros identificadores de tipo *string*, en valores numéricos, se utiliza, *embeddings* con el modelo *word2vec* (como se explicó en la sección anterior). Esto permite representar características no numéricas como vectores (y_i en la Figura 1).

Los *embeddings* se combinan con otras características numéricas del código, tales como: cantidad de variables, cantidad de métodos, total de nodos y profundidad del AST (x_i en la Figura 1).

La unión de los x_i y los y_i forma un vector de características completo. Este vector proporciona una descripción multidimensional del código y captura tanto la estructura como la semántica en una única representación.

1.2 *Dataset*

El *dataset* para análisis de similitud de código C# se compone de pares de archivos:

- **Pares similares:** archivos con la misma funcionalidad pero variaciones en estructura y nomenclatura (no copias directas, sino simulaciones de cambios comunes manteniendo la lógica).
- **Pares distintos:** archivos con funcionalidades diferentes.

La composición del conjunto de datos incluye 1000 pares generados de manera sintética por inteligencias artificiales generativas tales como, GPT-4 [23], Copilot [20], Perplexity [3] y Phind [25], divididos equitativamente entre pares similares y distintos. Además, se incluyen 100 pares de proyectos C# reales, de los cuales 30 son similares y fueron modificados manualmente, mientras que 70 son distintos.

Para generar las copias manuales similares, se aplicaron varias modificaciones. Estas incluyeron el renombramiento de clases, métodos y variables. También se realizaron modificaciones estructurales, como la alteración de bucles y la reestructuración de condiciones lógicas. Finalmente, se llevaron a cabo reorganizaciones del código, que implicaron cambios en el orden de ejecución y el uso de estructuras de control alternativas.

El *dataset* final contiene 1100 pares: 1000 sintéticos y 100 no sintéticos (copias manuales consideradas plagios reales). Este *dataset* se usará para entrenar un modelo de aprendizaje automático.

1.3 Redes neuronales siamesas para la similitud de código

Las redes neuronales siamesas se emplearon para analizar la estructura y el comportamiento lógico del código. Este enfoque facilita la identificación de patrones y las relaciones textuales y sintácticas. En este apartado se detallan los componentes de esta arquitectura, la implementación del modelo y el proceso que se realizó para el entrenamiento y predicción.

- **Entrada:** dos vectores x_1 y x_2 de tamaño n.
- **Subred base:** ambas entradas se procesan mediante una red compartida (mismas capas y pesos para ambas entradas) f_θ:

  $$d_1 = f_\theta(x_1), \quad d_2 = f_\theta(x_2),$$

  que producen representaciones latentes.
- **Métrica de distancia:** la similitud entre las representaciones latentes se mide mediante una distancia como se muestra en la figura 2

  $$d(d_1, d_2) = ||d_1 - d_2||_p,$$

  donde $p = 1$ para la distancia $L1$ [7], que se utiliza en este modelo.

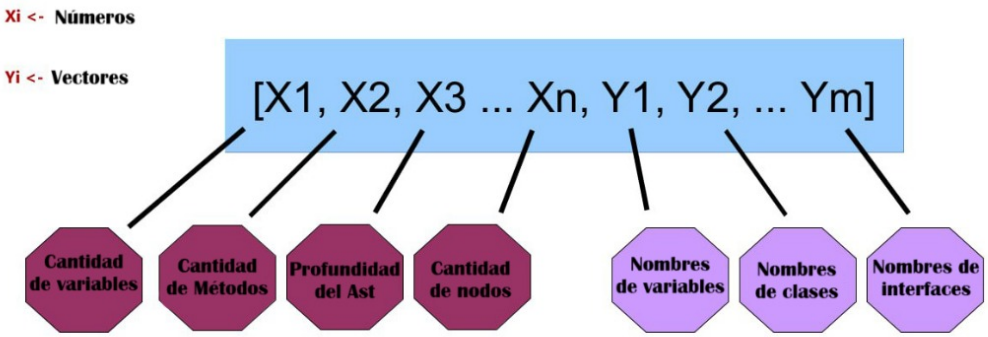

Figura 1. Ejemplo de vector de características [*Feature vector example*].

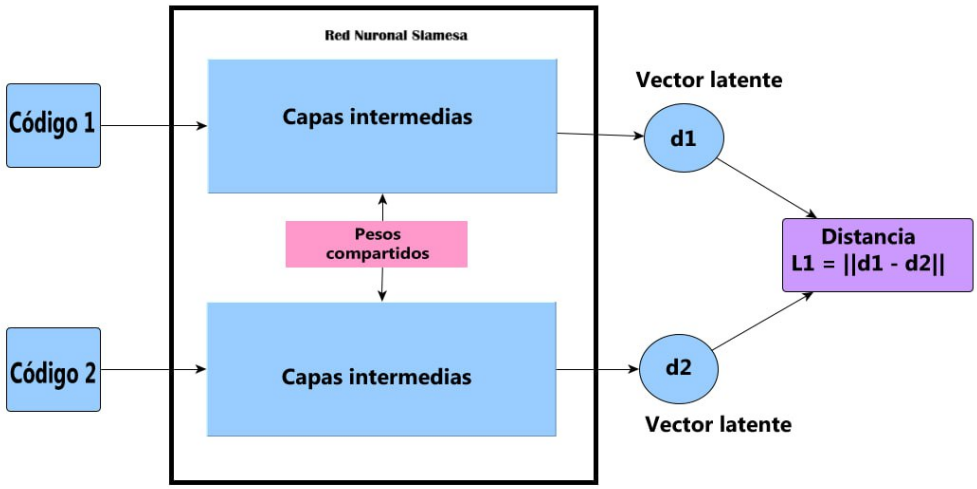

Figura 2. Estructura general de una Red Neuronal Siamesa [*General structure of a Siamese Neural Network*].

- **Clasificación binaria:** la distancia se pasa por una capa de activación sigmoidal [11] para obtener una probabilidad:

$$\hat{y} = \sigma(W \cdot d(d_1, d_2) + b).$$

1.3.1 Implementación del modelo

Esta sección describe cómo se implementa un modelo de red siamesa utilizando TensorFlow/Keras [1]. La implementación incluye tres componentes: la subred base, la arquitectura siamesa y los mecanismos de optimización.

La subred base transforma las entradas $x \in \mathbb{R}^{65}$ (la dimensión se corresponde con el número de *features* que se extraen del AST) en representaciones vectoriales como se explicó en la sección 1.1.5. Este proceso ayuda al modelo a identificar patrones y eliminar información redundante.

Las capas de la subred base son las siguientes:

1. **Capa densa (*Dense*):** cada capa densa aplica una transformación lineal seguida de una función de activación no lineal ReLU (*Rectified Linear Unit*) [10]. La fórmula matemática para una capa es:

$$h^{(l)} = \text{ReLU}(W^{(l)} \cdot h^{(l-1)} + b^{(l)}),$$

donde $W^{(l)}$ es la matriz de pesos, $b^{(l)}$ es el vector de sesgos y $h^{(l)}$ es la salida de la capa l.

2. **Regularización con *dropout*:** *dropout* apaga aleatoriamente una proporción $p = 0{,}5$ de las neuronas en cada capa durante el entrenamiento. Esto previene el sobreajuste del modelo [27].

Arquitectura detallada de la subred:

- Entrada $x \in \mathbb{R}^{65}$, donde cada dimensión es una característica del conjunto de datos.
- Tres capas densas con tamaños decrecientes (512, 256, y 128 neuronas) y activaciones ReLU.
- Salida $h(x) \in \mathbb{R}^{128}$, un vector de representación latente que condensa la información relevante de las entradas.

Esta arquitectura se utiliza en las dos ramas del modelo siamés. El orden en que se aplican las capas en cada rama se muestra en la Figura 3.

1.3.2 Construcción del modelo siamés

El modelo siamés utiliza dos copias idénticas de la subred base para procesar dos entradas, compararlas y calcular una probabilidad de similitud.

1. **Entradas:** dos vectores $x_1, x_2 \in \mathbb{R}^{65}$ que representan las instancias a comparar.

2. **Representaciones latentes:** cada entrada se transforma en una representación latente compacta mediante la subred base:

$$d_1 = f_\theta(x_1), \quad d_2 = f_\theta(x_2),$$

donde f_θ representa la subred base con parámetros compartidos θ.

3. **Cálculo de la distancia *L1*:** para medir la similitud entre las representaciones latentes, se calcula la distancia *L1*:

$$d(d_1, d_2) = ||d_1 - d_2||.$$

Esto se implementa utilizando una capa Lambda en TensorFlow/Keras.

4. **Clasificación:** la distancia calculada se pasa a una capa densa con una activación sigmoidal (*sigmoid*) que produce una probabilidad:

$$\hat{y} = \sigma(W \cdot d(d_1, d_2) + b),$$

donde:

- W son los pesos de la capa.
- b es el sesgo.
- $\sigma(z) = \frac{1}{1+e^{-z}}$ convierte z en un valor entre 0 y 1, que se interpreta como la probabilidad de similitud.

1.3.3 Función de pérdida

En el contexto del aprendizaje automático, una función de pérdida (también llamada función de costo) es una medida de cuán bien o mal funciona un modelo de aprendizaje automático. Evalúa la diferencia entre las predicciones que hace el modelo y los valores reales (etiquetas) que se observan en los datos de entrenamiento.

El objetivo del modelo durante el entrenamiento es minimizar la función de pérdida, lo que significa ajustar sus parámetros (por ejemplo los pesos de una red neuronal) de manera tal que las predicciones del modelo se acerquen lo más posible a los valores reales.

Se implementó una función de pérdida personalizada que introduce una penalización adicional a los falsos negativos. A diferencia de la función estándar *binary_crossentropy*, que considera una penalización uniforme para errores. Para la red que se propone, un falso positivo es una pareja de códigos que se detectan falsamente como copias, mientras que un falso negativo es una pareja de proyectos similares que no se puede detectar.

El software propuesto en esta investigación tiene el objetivo de capturar posibles códigos similares, para que luego un experto humano (profesor) los clasifique con exactitud. En este contexto resulta preferible devolver falsos plagios antes de fallar en detectar plagios reales.

La función de pérdida personalizada se define como:

$$\text{loss} = \textit{binary\_crossentropy}(y_{\text{true}}, y_{\text{pred}}) - \alpha \cdot y_{\text{true}} \cdot \log(1 - y_{\text{pred}} + \epsilon),$$

donde:

- y_{true}: etiqueta real (0 o 1).
- y_{pred}: predicción del modelo (probabilidad entre 0 y 1).
- α: factor de penalización para falsos negativos.
- ϵ: valor pequeño para evitar indeterminaciones en el cálculo del logaritmo.

El modelo utiliza el optimizador *Adam* (*Adaptive Moment Estimation*) para ajustar los pesos θ [10]. *Adam* combina las ventajas de:

- **Momentum:** acelera el entrenamiento en direcciones consistentes [26].
- **Tasas de aprendizaje adaptativas:** ajusta la tasa de aprendizaje individual para cada parámetro [9].

El algoritmo devuelve los parámetros actualizados θ_t, moviéndolos en la dirección que minimiza la función de pérdida.

1.3.4 Entrenamiento y predicción

Se entrena el modelo siamés utilizando un conjunto de tuplas de datos (x_1, x_2, y), donde x_1 y x_2 son dos instancias que se comparan entre sí, y y es la etiqueta binaria que indica si las instancias son similares ($y = 1$) o disímiles ($y = 0$).

Durante el entrenamiento, el modelo ajusta los pesos para minimizar la función de pérdida sobre el conjunto de entrenamiento. Se optimiza el modelo con el algoritmo de *Adam*.

El modelo procesa cada par de entradas x_1, x_2 a través de la subred base para calcular sus representaciones latentes d_1 y d_2. Posteriormente, se calcula la distancia entre las representaciones y utiliza esta información para una predicción $\hat{y}$, que se compara con la etiqueta y en la función de pérdida. A medida que el modelo se entrena, ajusta sus pesos para minimizar la diferencia entre las predicciones $\hat{y}$ y las etiquetas y.

Después de entrenar el modelo, se pueden hacer predicciones sobre nuevos pares de características $\mathbf{X_1}$ y $\mathbf{X_2}$. La similitud entre los dos fragmentos de código se calcula pasando las características de cada fragmento a través del modelo y obteniendo un valor numérico de similitud.

El valor de la similitud se obtiene de la siguiente forma:

$$\text{similarity} = \text{model.predict}([\mathbf{X_1}, \mathbf{X_2}]).$$

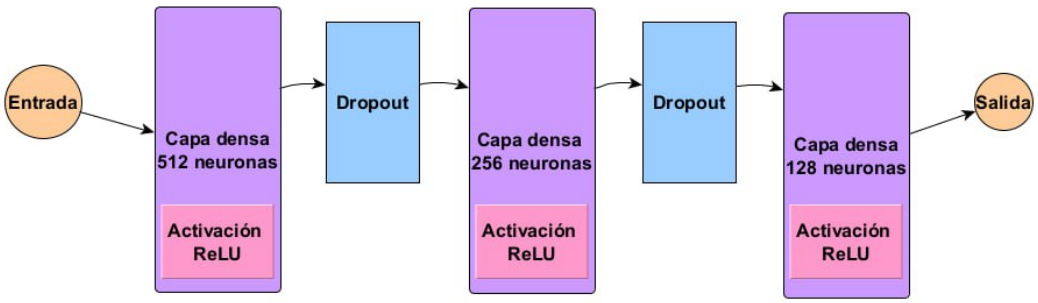

Figura 3. Capas de la subred base [*Base subnet layers*].

Un valor cercano a 1 indica que los fragmentos son similares, mientras que un valor cercano a 0 indica que son disímiles. La similitud se basa en la activación de la red, que convierte la distancia entre las representaciones latentes de los fragmentos en un valor de probabilidad.

1.3.5 Evaluación del modelo

Para evaluar el modelo se utiliza un conjunto de datos de prueba compuesto por pares de instancias etiquetados. El objetivo de esta evaluación es medir el desempeño del modelo en términos de la precisión (*precision*), exhaustividad (*recall*), y exactitud (*accuracy*). Además, se analizaron los errores para identificar áreas de mejora.

El modelo se evaluó de la siguiente forma:

1. **Cálculo de pérdida y precisión:** se calcularon la pérdida y la precisión promedio en el conjunto de datos de prueba.

2. **Predicción con umbral ajustado:** se generaron predicciones con un umbral definido para clasificar las instancias. En este caso, el umbral fue $(0{,}10)$, que se seleccionó para priorizar los verdaderos positivos.

3. **Análisis de la matriz de confusión:** se analizó la matriz de confusión para identificar los errores del modelo y categorizar las instancias en verdaderos positivos (TP), falsos positivos (FP), verdaderos negativos (TN), y falsos negativos (FN) (Tabla 2); todas las siglas de sus originales en inglés.

Tabla 2. Métricas de rendimiento del modelo [*Model performance metrics*].

Métrica	Valor
Verdaderos Positivos (TP)	98
Falsos Positivos (FP)	21
Verdaderos Negativos (TN)	75
Falsos Negativos (FN)	2

4. **Identificación de errores:** se registraron los pares de datos que resultaron falsos negativos y falsos positivos para analizar sus características. Esto permitió identificar posibles patrones en los errores.

2. Resultados

La experimentación tuvo como objetivo principal reducir los falsos negativos (como se explicó en la sección 1.3.3) y, con ello, aumentar el *recall* del modelo. Este enfoque responde a la necesidad de priorizar la sensibilidad del modelo en detrimento de la precisión del mismo, dado lo importante que resulta capturar correctamente todos los casos positivos (proyectos que son fraude), incluso, a cambio de generar un mayor número de falsos positivos. Como valor de α se utilizó $0{,}45$.

Esta estrategia conllevó a una disminución del *accuracy* en comparación con la función de pérdida *binary_crossentropy*.

Para evaluar el rendimiento del modelo, se utilizaron las métricas estándar descritas a continuación:

- ***Accuracy*:** mide la proporción de predicciones correctas sobre el total de predicciones. Se define como:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN},$$

  donde TP representa los verdaderos positivos, TN los verdaderos negativos, FP los falsos positivos y FN los falsos negativos.

- ***Precision*:** evalúa la proporción de verdaderos positivos entre todas las predicciones clasificadas como positivas por el modelo:

$$precision = \frac{TP}{TP + FP}.$$

  En esta fórmula, el aumento de los falsos positivos (FP) incrementa el denominador, lo que resulta en una disminución de la precisión. Esto refleja que el modelo clasifica incorrectamente más ejemplos como positivos, lo que reduce la proporción de verdaderos positivos respecto al total de predicciones positivas.

- ***Recall*:** mide la proporción de verdaderos positivos capturados por el modelo respecto al total de ejemplos

positivos reales:

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

En esta expresión, el aumento de los verdaderos positivos (TP) incrementa el numerador, lo que lleva a un mayor valor de *recall*. Asimismo, la disminución de los falsos negativos (FN) reduce el denominador, lo que también incrementa esta métrica. Por tanto, el *recall* es directamente proporcional a los verdaderos positivos y de forma inversa proporcional a los falsos negativos.

El diseño de una función de pérdida personalizada que enfatiza la penalización de los falsos negativos permitió optimizar el *recall*, aunque a expensas de reducir la *precision* y la *accuracy*. Este enfoque prioriza minimizar los errores en la identificación de ejemplos positivos, incluso cuando ello conlleva un aumento en los errores al clasificar negativos como positivos. Si bien esta estrategia no es adecuada en contextos donde la precisión es prioritaria, es efectiva en problemas donde los falsos negativos representan un riesgo para el sistema.

En la Tabla 3 se presentan los resultados de los casos de pruebas donde se utilizaron las dos funciones de pérdida implementadas. Los resultados arrojaron que la función de pérdida personalizada logró mejores resultados en términos de *recall*, mientras que *binary_crossentropy* mantuvo un *recall* ligeramente inferior, pero consiguió un *accuracy* considerablemente mayor.

Tabla 3. Resultados de las métricas para los casos de prueba [*Metric results for the test cases*].

Función de pérdida	*accuracy*	*loss*	*precision*	*recall*
binary_crossentropy	0,9082	0,4948	0,9192	0,9100
assymetric_loss	0,8469	1,1607	0,8235	0,9800

El *recall* cercano a 1 de la función de pérdida personalizada resultó en un modelo capaz de detectar una inmensa mayoría de los proyectos similares al comparar dos a dos un conjunto de códigos en lenguaje C#. Además, con *binary_crossentropy* se entrenó otro modelo que captura una gran cantidad de plagios, y da, al mismo tiempo, pocos falsos positivos. Este es el principal aporte del presente trabajo y la siguiente sección sirve como validación para este resultado.

2.1 Validación de resultados

Para validar los resultados de la investigación se escogieron cuatro proyectos reales que se orientaron en la asignatura Programación que se imparte en el primer año de licenciatura en Ciencia de la Computación en la Universidad de La Habana. Para clasificar se usó el modelo que se entrenó con la función de pérdida personalizada presentada en la sección 1.3.3.

La Tabla 4 muestra los resultados alcanzados para cada uno de los cuatro proyectos tenidos en cuenta (*Moogle* [17]), *Dominó* [15], *Hulk* [16] y *WallE* [18]).

La Tabla 4 presenta cuatro columnas:

1. Parejas plagio: pares de proyectos copia que se generaron.
2. Parejas no plagio: proyectos no similares que se emparejaron dos a dos.
3. Plagios detectados: parejas de proyectos copia que se generaron y fueron detectados por el modelo.
4. Plagios no detectados: parejas de proyectos copia que se generaron y el algoritmo no fue capaz de detectar.
5. Falsos plagios: parejas de proyectos no similares que el algoritmo detectó erróneamente como copias.

En cada caso se seleccionó un conjunto de 10 proyectos implementados por estudiantes y a un subconjunto aleatorio de 5 de estos proyectos (por cada grupo de 10 se escogieron 5), se le generó una copia, simulando un posible fraude. A tales efectos, se realizaron, entre otras, las siguientes transformaciones:

- Cambios en nombres de variables, métodos y clases.
- Cambios estructurales (como, por ejemplo, cambio del orden de las declaraciones de variables) que no afectan la lógica del código.
- Intercambios de tipos similares (Como, por ejemplo, intercambios de *long* por *int* y *double* por *float*).
- Cambios sintácticos que no afectan la lógica del código (como, por ejemplo, intercambios de condicionales *Switch* por conjuntos de condicionales *if*.)

En cada tipo de proyecto se probaron, por tanto, 5 parejas de proyectos que se clasifican como *plagio* y $\binom{15}{2}$ - 5 = 100 – todas las parejas dos a dos menos las 5 parejas de proyectos clonadas – que no se clasifican como plagio.

Los resultados que se presentaron en este apartado muestran que el modelo es capaz de identificar correctamente los proyectos fraudulentos en una gran mayoría de los casos. Sin embargo, se capturaron también erróneamente varios proyectos no fraudulentos que deben ser filtrados por los expertos (profesores).

2.2 Otros resultados

La herramienta ANTLR cuenta con una gramática basada en C# 6.0. Esta gramática presentaba limitaciones significativas al trabajar con el código moderno (o sea, código que utilice versiones recientes del lenguaje). Se generan errores durante el análisis sintáctico y se genera un AST incompleto. Por tal motivo, se decide trabajar con una versión más actualizada del lenguaje C# 12.0 (introducida en noviembre de 2023) para reflejar las características que presentan versiones posteriores a C# 6. Se realizaron cambios a la gramática en dichas versiones, según lo reflejado en la documentación correspondiente de Microsoft [21] y las modificaciones que, como parte de este trabajo, se han realizado para actualizar la misma.

Tabla 4. Resultados comparativos de detección de similitud *Comparative similarity detection results*.

Sistema	Parejas plagio	Parejas nO plagio	Plagios detectados	Plagios no detectados	Falsos plagios
Moogle	5	100	5	0	18
Dominó	5	100	5	0	20
Hulk	5	100	5	0	22
WallE	5	100	5	0	25

Conclusiones

Los principales aportes científicos de esta investigación son los siguientes: (1) se desarrolló un software de detección de similitud de código en C# que extrae características de los AST y las utiliza como entrada para un modelo de Aprendizaje Automático; (2) se diseñó e implementó una herramienta práctica, y de código libre, que se puede usar con fines didácticos a partir de los experimentos realizados que validan su utilidad; y (3) se actualizó la gramática de C# de ANTLR, de la versión 6.0 a la 12.0.

Contribución de autoría

Conceptualización M.L.C.F.

Curación de datos M.L.C.F, R.G.G.

Análisis formal M.L.C.F.

Investigación M.L.C.F, R.G.G.

Metodología M.L.C.F, A.F.O.

Recursos M.L.C.F.

Software M.L.C.F.

Supervisión R.G.G, A.F.O.

Validación R.G.G, A.F.O.

Visualización M.L.C.F.

Redacción: preparación del borrador original M.L.C.F.

Redacción: revisión y edición A.F.O.

Suplementos

Este artículo no contiene información suplementaria.

Conflictos de interés

Se declara que no existen conflictos de interés.

Referencias

[1] Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Eastwood, S. Gray, D. Harvey, G. Irving, M. Isard, Y. Jia, R. M. K., J. Kratz, K. Malhotra, B. McGinnis, S. Moore, D. Murray, D. Orr, M. Schuster, J. Susskind, Z. Tu, V. V., P. Warden, X. Wu, and R. Zadeh: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2016. https://arxiv.org/abs/1603.04467.

[2] Abba, H.L., A. Roko, A.B. Muhammad, A. Usman, and A. Almu: *Enhanced Semantic Similarity Detection of Program Code Using Siamese Neural Network*. International Journal of Advanced Networking and Applications, 14(2):5353–5360, 2022. https://www.ijana.in/papers/V14I2-5.pdf.

[3] AI, Perplexity: *Perplexity AI*, 2022. https://www.perplexity.ai.

[4] Aiken, A.: *Moss (measure of software similarity)*, 1994. https://theory.stanford.edu/~aiken/moss/.

[5] Baxter, I.D., A. Yahin, L. Moura, M. Sant'Anna, and L. Bier: *Clone detection using abstract syntax trees*. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 368–377, 1998. https://doi.org/10.1109/ICSM.1998.738528.

[6] Benedikt, G., F. Matthias, K. Jens, and P. Helmut: *Jplag: A system for detecting software plagiarism*. https://github.com/jplag/jplag.

[7] Bishop, C.M. and L.J.F.H. van der Maaten: *L1 Distance in Machine Learning: Applications and Analysis*. IEEE Transactions on Neural Networks, 6(1):11–19, 1995. https:doi.org/10.1109/72.460515.

[8] Bromley, J., Y. LeCun, I. Guyon, R. Shah, L. Bottou, and Y. Hu: *Signature verification using a Siamese time delay neural network*. In *IEEE International Conference on Neural Networks, 1993*, pages 669–674. IEEE, 1993. https://dl.acm.org/doi/10.5555/2987189.2987282.

[9] Duchi, J., E. Hazan, and Y. Singer: *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*. Journal of Machine Learning Research, 12(7):2121–2159, 2011. https://dl.acm.org/doi/10.5555/1953048.2021068.

[10] Goodfellow, I., Y. Bengio, and A. Courville: *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] Hahnloser, R.H., R.S. Finkelstein, and S.H. Seung: *Sigmoid Activation Functions for Neural Networks*. Neural Computation, 12(4):909–931, 2000. https://doi.org/10.1162/089976600300015000.

[12] Hoq, M., Y. Shi, J. Leinonen, D. Babalola, C.F. Lynch, T.W. Price, and B. Akram: *Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models*. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V1*, SIGCSE 2024, pages 526–532, New York, NY, USA, 2024. Association for Computing Machinery, ISBN 9798400704239. https://doi.org/10.1145/3626252.3630826.

[13] Koch, G., R. Zemel, and R. Salakhutdinov: *Siamese neural networks for one-shot image recognition*. Proceedings of the 32nd International Conference on Machine Learning (ICML), 2015. https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf.

[14] Krizhevsky, A., I. Sutskever, and G.E. Hinton: *ImageNet Classification with Deep Convolutional Neural Networks*. Communications of the ACM, 60(6):84–90, 2017. https://doi.org/10.1145/3065386.

[15] MATCOM: *Domino*. https://github.com/matcom/domino.

[16] MATCOM: *Hulk*. https://github.com/matcom/hulk.

[17] MATCOM: *Moogle*. https://github.com/matcom/moogle.

[18] MATCOM: *Wall-E*. https://github.com/matcom/cool-compiler-base-2019.

[19] Mehrotra, N., N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare: *Modeling Functional Similarity in Source Code With Graph-Based Siamese Networks*. IEEE Transactions on Software Engineering, 48(10):3771–3789, 2022. https://doi.org/10.1109/TSE.2021.3105556.

[20] Microsoft: *GitHub Copilot*, 2021. https://github.com/features/copilot.

[21] Microsoft: *C# Version History*, 2024. https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history.

[22] Mikolov, T., K. Chen, G. Corrado, and J. Dean: *Efficient estimation of word representations in vector space*, 2013. https://doi.org/10.48550/arXiv.1301.3781.

[23] OpenAI: *GPT-4 Technical Report*. arXiv preprint arXiv:2303.08774, 2023.

[24] Parr, T.: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh, NC, 2013. https://dl.icdst.org/pdfs/files3/a91ace57a8c4c8cdd9f1663e1051bf93.pdf.

[25] Phind: *Phind*, 2023. https://www.phind.com.

[26] Qian, N.: *On the Momentum Term in Gradient Descent Learning Algorithms*. Neural Networks, 12(1):145–151, 1999. https://doi.org/10.1016/S0893-6080(98)00116-6.

[27] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov: *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Journal of Machine Learning Research, 15(56):1929–1958, 2014. http://jmlr.org/papers/v15/srivastava14a.html.

[28] Tankala, D. K., Venugopal T. and V. Boddu: *Java source code similarity detection using Siamese networks*. Journal of Theoretical and Applied Information Technology, 100(17):5507, 2022. http://www.jatit.org/volumes/Vol100No17/17Vol100No17.pdf.

[29] Wang, W., G. Li, B. Ma, X. Xia, and Z. Jin: *Code Similarity Detection Technique Based on AST Unsupervised Clustering Method*. In *Proceedings of the 2020 International Conference on Computational Communications and Networks (ICCC)*, 2020. https://doi.org/10.1109/ICCC51575.2020.9344882.

[30] Wang, W., G. Li, B. Ma, X. Xia, and Z. Jin: *Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree*. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271, 2020. https://doi.org/10.1109/SANER48275.2020.9054857.

[31] Yang, S., Cheng L. Zeng Y. Lang Z. Zhu H. and Z. Shi: *Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection*. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 224–236, 2021. https://doi.org/10.1109/DSN48987.2021.00036.