Universidad de La Habana
Facultad de Matemática y Computación

Code Similarity

Autor:
María de Lourdes Choy Fernández

Tutor:
Rodrigo García Gomez
Cotutor:

Trabajo de Diploma
presentado en opción al título de
Licenciado en Ciencia de la Computación

febrero de 2024

github.com/Chonyyy/Code_Similarity

A mi familia
A mis amigos
A mis profesores.

Agradecimientos

Agradecimientos

Opinión del tutor

Ve haciendo ya tu opinion

Resumen

Esta investigación aborda el problema de la detección de similitudes en código fuente, específicamente en proyectos de C#. El proceso comienza con la extracción del Árbol de Sintaxis Abstracta (AST) de un conjunto de proyectos, lo que permite una representación estructural del código. A partir de estos AST, se extraen características relevantes que describen los elementos del código y estas se agrupan en vectores. Estos vectores se utilizan de entrada en redes neuronales siamesas para obtener si son similares o diferentes.

Resumen

This research addresses the problem of detecting similarities in source code, specifically in C projects. The process begins with the extraction of the Abstract Syntax Tree (AST) from a set of projects, enabling a structural representation of the code. From these ASTs, relevant features are extracted to describe the code elements, and these features are grouped into vectors. These vectors are then used as input to Siamese neural networks to determine whether they are similar or different.

Índice general

Introducción

El análisis de similitud de código es un área de Ciencias de la Computación que se centra en la comparación de fragmentos de código para identificar similitudes y diferencias. Este enfoque es fundamental en la detección de plagio y en la revisión de código, ya que proporciona métricas objetivas para la evaluación de la calidad del código y la adopción de buenas prácticas de programación.

Motivación

En el contexto académico, la detección de similitud de código y el análisis de plagio representan un desafío en la evaluación de proyectos de programación. Con el crecimiento constante de los recursos de ciencias de la computación, los educadores enfrentan la tarea de garantizar la originalidad en las soluciones presentadas por los estudiantes. Este problema no solo afecta la integridad académica, sino que también dificulta evaluar de manera justa el aprendizaje y la comprensión de conceptos.

El plagio de código puede adoptar diversas formas, desde la copia literal hasta modificaciones superficiales como el cambio de nombres de variables, la reorganización de bloques de código o la reescritura con constructos equivalentes. Estas transformaciones suelen evadir los enfoques tradicionales de detección basados en comparaciones textuales o métricas básicas de similitud. Como resultado, la detección efectiva requiere sistemas que puedan identificar similitudes semánticas y estructurales, independientemente de las transformaciones realizadas.

Esta investigación surge de la necesidad de dotar a las instituciones educativas de herramientas avanzadas que puedan analizar código con precisión. Al combinar técnicas de análisis estático, modelos de representación estructural y métodos de aprendizaje automático, esta tesis propone

un marco robusto para la detección de similitud de código. Este enfoque no solo busca resolver los problemas prácticos del plagio en entornos académicos, sino también contribuir al entendimiento teórico del problema y ofrecer una base para futuras investigaciones en el área.

Objetivos Generales

I. Desarrollar un marco de análisis de similitud de código que combine la extracción de características mediante árboles de sintaxis abstracta (AST) con técnicas avanzadas de aprendizaje automático para mejorar la precisión y eficiencia en la detección de similitudes.

II. Contribuir al conocimiento académico y práctico en el campo del análisis de código, proporcionando herramientas y metodologías que puedan ser utilizadas tanto en entornos educativos como profesionales.

Objetivos Específicos

I. **Extracción y Procesamiento de AST para la Comparación de Código**
El primer objetivo es diseñar e implementar un sistema que utilice Árboles de Sintaxis Abstracta (AST) para analizar y comparar proyectos de código. En la primera etapa, se utiliza la herramienta ANTLR y se modifica su gramática para C# con el fin de generar los AST de los proyectos, a partir de los cuales se extraen los *features* relevantes para su posterior análisis.En la segunda etapa, los conjuntos de *features* extraídos de cada proyecto se convierten en vectores de características, que servirán como base para las comparaciones entre códigos. Este enfoque busca capturar similitudes estructurales y semánticas en los proyectos.

II. **Integrar Técnicas de Aprendizaje Automático para la Detección de Similitudes**
El segundo objetivo es integrar técnicas de aprendizaje automático para mejorar la detección de similitudes en el código. Esto implica

entrenar un modelo de aprendizaje supervisado utilizando un conjunto de datos conformado por proyectos de C#. Estos modelos deberán capturar patrones complejos y relaciones estructurales en el código, proporcionando una visión precisa de las similitudes. La implementación de estas técnicas permitirá comparar su desempeño con los métodos tradicionales, evaluando mejoras en precisión y eficiencia.

III. **Desarrollar y Validar una Herramienta Práctica**
El tercer objetivo es desarrollar una herramienta de software que implemente las técnicas avanzadas desarrolladas, facilitando su uso por desarrolladores y educadores. Esta herramienta deberá ser práctica y accesible, permitiendo su integración en entornos de desarrollo reales. La validación de la herramienta se llevará a cabo en escenarios prácticos, como la detección de plagio en tareas de programación. Esto no solo demostrará la efectividad de las técnicas implementadas, sino que también garantizará que la herramienta sea útil y relevante para los usuarios finales.

Capítulo 1

Preliminares

1.1. Introducción al Análisis de Similitud de Código

El análisis de similitud de código es un campo que ha evolucionado desde sus inicios en la década de 1970. Es utilizado en aplicaciones como la detección de plagio, refactorización de código, revisión de código y herramientas de asistencia a profesores de programación. En esta sección se revisan los desarrollos históricos, metodologías y tecnologías clave, así como los avances recientes en este ámbito.

Los orígenes del análisis de similitud de código se remontan a los años 70 con los primeros algoritmos de coincidencia de cadenas, cuando se buscaban métodos para detectar plagio en tareas de programación. Algoritmos como Knuth-Morris-Pratt (KMP) [9] y Boyer-Moore [5] se utilizaron inicialmente para comparar secuencias de texto, sentando las bases para la comparación de código fuente.

En los 90, herramientas como MOSS (Measure of Software Similarity) [2] fue un primer paso dentro de este ámbito. MOSS normaliza el código, eliminando comentarios y renombrando variables, para detectar similitudes estructurales y lógicas, incluso frente a modificaciones superficiales. Esta herramienta tuvo un impacto significativo en el ámbito académico, promoviendo la integridad académica.

1.2. Técnicas Basadas en Árboles de Sintaxis Abstracta (AST)

Originalmente desarrollados en los 80 para compiladores, los AST comenzaron a usarse en los 2000 para análisis y transformación de código [1]. Permiten comparaciones basadas en la estructura lógica y sintáctica del código, superando las limitaciones de las comparaciones textuales simples.

Un Árbol de Sintaxis Abstracta (AST) es una representación jerárquica y estructurada de un código fuente. Cada nodo del árbol representa una construcción en el lenguaje de programación, como operadores, declaraciones o expresiones. Esta representación abstracta facilita el análisis de alto nivel del código, ignorando detalles superficiales como el formato o los comentarios.

I. Baxter et al. [**?**] propusieron métodos que utilizan hashing para detectar clones exactos y casi coincidentes mediante subárboles de AST. Aunque efectivos, enfrentan desafíos como falsos positivos en la detección de clones complejos.

B. N. Pellin [**?**] utilizó SVM y AST para clasificar autoría en código fuente. Aunque precisa, esta técnica es vulnerable a manipulaciones avanzadas del código.

Comparar subárboles de AST permite detectar similitudes en diferentes niveles de granularidad. Esto incluye no solo similitudes textuales, sino también similitudes semánticas y estructurales que reflejan la lógica subyacente del código, incluso cuando este ha sido modificado para evitar detección directa.

1.3. Avances en Análisis de Similitud de Código

1.3.1. Integración del Aprendizaje Automático

Desde 2010, el aprendizaje automático ha mejorado la precisión del análisis de similitud de código. Técnicas como redes neuronales recurrentes (RNN) y convolucionales (CNN) generan representaciones vectoriales que capturan patrones estructurales complejos.

1.3.2. Redes Neuronales Siamesas

Las redes neuronales siamesas son un tipo de arquitectura de red diseñada para aprender similitudes entre pares de entradas. Originalmente propuestas por Bromley y LeCun [**?**], estas redes 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 similitud entre las dos representaciones vectoriales se mide usando una métrica, como la distancia euclidiana o la similitud del coseno. Este enfoque permite identificar relaciones semánticas y estructurales entre fragmentos de código, incluso cuando no son idénticos textualmente.

Aplicaciones destacadas de redes siamesas incluyen la detección de plagio y la búsqueda de fragmentos de código similares en grandes bases de datos. Estas redes son útiles cuando se necesita comparar fragmentos de código que han sido modificados, ya que se centran en la esencia lógica y estructural de los fragmentos.

1.4. Representaciones Vectoriales y Aprendizaje Profundo

Word2Vec se ha utilizado para generar representaciones vectoriales de fragmentos de código, capturando relaciones semánticas complejas. Las representaciones vectoriales permiten realizar comparaciones rápidas, ya que los fragmentos de código se transforman en puntos dentro de un espacio vectorial. Estas representaciones capturan tanto la estructura lógica como los patrones semánticos, lo que facilita la detección de similitudes en un nivel más abstracto.

Desde 2014, se han empleado redes neuronales profundas (DNN) para aprender incrustaciones vectoriales de código. Estas técnicas han mejorado la precisión en la detección de similitudes y relaciones contextuales entre fragmentos de código.

Capítulo 2

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

ANTLR (Another Tool for Language Recognition) cuenta con una gramática basada en C# 6.0, esta gramática presentaba limitaciones significativas al trabajar con el código moderno. Generaba errores durante el análisis sintáctico y resultaba en un árbol de sintaxis abstracta (AST) incompleto. Sin embargo, se decidió actualizar a una versión más actual de C# 12.0 (introducida en noviembre de 2023), para reflejar las características introducidas en versiones posteriores. A continuación, se describen los principales problemas encontrados y las modificaciones realizadas para solventarlos.

2.1. C# 7.0

Variables `out`

Problema: El uso de variables `out` requería una declaración previa. Desde C# 7.0, se permite declarar variables directamente en la declaración del método.

Solución: Se modificó la regla `argument` para incluir una declaración de variable opcional antes del modificador `out`. Además, se ajustó la regla `local_variable_declaration` para permitir el uso de `out` como modificador de variable.

Tuplas y Deconstrucción

Problema: Las tuplas no eran un tipo nativo y requerían una biblioteca externa. Desde C# 7.0, se introdujeron tuplas como un tipo nativo, permitiendo la deconstrucción de sus elementos.

Solución: Se agregó una nueva regla `tuple_type` para definir tipos de tuplas. Se modificó la regla `type` para incluir `tuple_type`. Además, se añadió una regla `tuple_literal` para la creación de tuplas y se actualizó la regla `assignment` para soportar la deconstrucción de tuplas.

Coincidencia de Patrones

Problema: No había soporte para coincidencia de patrones en C#. Desde C# 7.0, se introdujo la coincidencia de patrones que permite simplificar el código al verificar tipos y valores.

Solución: Se introdujo una nueva regla `pattern_matching_expression` que incluye subreglas para diferentes tipos de patrones (constante, tipo, var). Se actualizaron las reglas `is_expression` y `switch_section` para incorporar la nueva sintaxis de coincidencia de patrones.

Funciones Locales

Problema: Las funciones debían estar definidas a nivel de clase. Desde C# 7.0, se introdujeron funciones locales que permiten definir funciones dentro de métodos.

Solución: Se creó una nueva regla `local_function_declaration` similar a `method_declaration`, pero permitida dentro de bloques de código. Se actualizó la regla `statement` para incluir `local_function_declaration` como una opción válida.

Miembros Expresados con Cuerpo de Expresión Expandido

Problema: Los miembros debían tener un cuerpo completo. Desde C# 7.0, se introdujo la posibilidad de usar cuerpos de expresión para simplificar la definición de miembros.

Solución: Se modificaron las reglas `method_declaration`, `property_declaration`, `operator_declaration`, `constructor_declaration`, `destructor_declaration`,

y `indexer_declaration` para permitir el uso de `=>` seguido de una expresión como alternativa al cuerpo del bloque tradicional.

Locales `ref`

Problema: No era posible utilizar locales `ref` que permiten referenciar variables directamente en lugar de copiar su valor. Desde C# 7.0, se introdujeron los locales `ref`.

Solución: Se actualizó la regla `local_variable_declaration` para incluir el modificador `ref` opcional. Se agregó una nueva regla `ref_expression` para manejar expresiones que devuelven referencias.

Retornos `ref`

Problema: Los métodos no podían devolver referencias directamente. Desde C# 7.0, se introdujo el soporte para retornos `ref`, permitiendo que un método devuelva una referencia a un valor existente.

Solución: Se modificó la regla `method_declaration` para permitir el modificador `ref` antes del tipo de retorno. Se actualizó la regla `return_statement` para incluir la opción de devolver una referencia usando `ref`.

Discards

Problema: No había una forma explícita de ignorar valores en asignaciones o expresiones. Desde C# 7.0, se introdujo el uso de discards (_) que permite omitir valores que no son necesarios.

Solución: Se agregó una nueva regla `discard` que reconoce el símbolo `_`. Se actualizaron las reglas relevantes como `variable_declaration`, `foreach_statement`, y `out_variable_declaration` para permitir el uso de `discard` en lugar de identificadores.

Literales Binarios y Separadores de Dígitos

Problema: No había soporte para literales binarios ni separadores de dígitos en números. Desde C# 7.0, se introdujeron literales binarios y la capacidad de usar guiones bajos como separadores en literales numéricos.

Solución: Se modificó la regla `integer_literal` para incluir el prefijo `0b` o `0B` para literales binarios. Se actualizaron las reglas de literales numéricos para permitir guiones bajos (_) entre dígitos en cualquier posición excepto al principio o final del literal.

Expresiones `throw`

Problema: Las expresiones `throw` solo podían usarse en instrucciones completas. Desde C# 7.0, se introdujeron expresiones `throw`, permitiendo lanzar excepciones como parte de expresiones más complejas.

Solución: Se creó una nueva regla `throw_expression` que permite el uso de `throw` seguido de una expresión. Se actualizaron las reglas relevantes como `conditional_expression` y `null_coalescing_expression` para incluir `throw_expression` como una opción válida.

2.1.1. C# 7.1

Método `Main` Asíncrono

Problema: El método de entrada `Main` no podía ser asíncrono. Esto limitaba el uso de operaciones asincrónicas en la inicialización de aplicaciones. Desde C# 7.1, se permite que el método `Main` tenga el modificador `async`.

Solución: Se ajustaron las reglas gramaticales para permitir el modificador `async` en el método de entrada `Main`.

Expresiones Literales por Defecto (`default`)

Problema: En versiones anteriores, era necesario especificar explícitamente el tipo al usar valores predeterminados. Desde C# 7.1, se introdujeron las expresiones literales `default`, que infieren automáticamente el tipo cuando es posible.

Solución: Se modificó la gramática para permitir expresiones literales `default` en contextos donde el tipo puede inferirse.

Inferencia de Nombres de Elementos en Tuplas

Problema: En versiones anteriores, era necesario nombrar explícitamente los elementos de una tupla al inicializarla. Desde C# 7.1, se permite inferir los nombres de los elementos a partir de las variables utilizadas en la inicialización.

Solución: Se ajustaron las reglas gramaticales para inferir automáticamente los nombres de los elementos en inicializaciones de tuplas.

Coincidencia de Patrones en Parámetros Genéricos

Problema: No era posible realizar coincidencia de patrones en variables cuyo tipo era un parámetro genérico. Desde C# 7.1, se permite realizar coincidencias de patrones en este caso.

Solución: Se expandieron las reglas para permitir coincidencia de patrones en variables con tipos genéricos.

2.1.2. C# 7.2

Inicializadores en Arreglos Stackalloc

Problema: No era posible inicializar directamente los arreglos creados con `stackalloc`. Desde C# 7.2, se permite usar inicializadores en arreglos `stackalloc`.

Solución: Se ajustó la gramática para permitir inicializadores en arreglos creados con `stackalloc`.

Uso de Sentencias Fijas con Tipos que Soportan Patrones

Problema: Las sentencias `fixed` solo podían usarse con tipos específicos como arreglos o cadenas. Desde C# 7.2, se permite usar sentencias `fixed` con cualquier tipo que soporte un patrón.

Solución: Se modificó la regla `fixed_statement` para permitir el uso de cualquier tipo que soporte patrones.

Acceso a Campos Fijos Sin Fijación

Problema: Para acceder a campos fijos era necesario fijar la dirección de memoria explícitamente. Desde C# 7.2, se permite acceder a campos fijos sin necesidad de fijarlos.

Solución: Se ajustaron las reglas gramaticales para permitir el acceso a campos fijos sin fijación explícita.

Reasignación de Variables Locales Referenciadas

Problema: No era posible reasignar variables locales referenciadas (`ref`). Desde C# 7.2, se permite reasignar estas variables.

Solución: Se modificaron las reglas para permitir la reasignación de variables locales referenciadas.

Declaración de Tipos Struct Solo de Lectura (`readonly struct`)

Problema: No había una forma explícita de declarar estructuras inmutables. Desde C# 7.2, se introdujo el modificador `readonly struct` para indicar que una estructura es inmutable y debe pasarse como un parámetro `in`.

Solución: Se agregó soporte para el modificador `readonly` en la declaración de estructuras.

Modificador `in` en Parámetros

Problema: No era posible pasar argumentos por referencia sin permitir su modificación. Desde C# 7.2, se introdujo el modificador `in`, que permite pasar argumentos por referencia sin permitir modificaciones.

Solución: Se ajustó la regla `parameter_modifier` para incluir el modificador `in`.

Modificador `ref readonly` en Retornos de Métodos

Problema: No había una forma explícita de devolver referencias inmutables desde métodos. Desde C# 7.2, se introdujo el modificador `ref readonly`, que permite devolver referencias inmutables.

Solución: Se agregó soporte para el modificador `ref readonly` en las reglas de retorno de métodos.

Declaración de Tipos Struct Referenciados (`ref struct`)

Problema: No había una forma explícita de declarar estructuras que accedan directamente a memoria administrada y deban ser asignadas en el stack. Desde C# 7.2, se introdujo el modificador `ref struct`.

Solución: Se agregó soporte para el modificador `ref struct` en la declaración de estructuras.

Argumentos Nombrados No Finales

Problema: En versiones anteriores, los argumentos nombrados debían colocarse al final de la lista de argumentos. Desde C# 7.2, se permite mezclar argumentos posicionales y nombrados siempre que los posicionales estén primero.

Solución: Se ajustaron las reglas gramaticales para permitir argumentos nombrados no finales.

Guiones Bajos Iniciales en Literales Numéricos

Problema: Los literales numéricos no podían comenzar con guiones bajos (_). Desde C# 7.2, se permite usar guiones bajos iniciales antes de los dígitos impresos.

Solución: Se ajustó la gramática para permitir guiones bajos iniciales en literales numéricos.

Modificador `private protected`

Problema: No había un modificador que combinara las restricciones de acceso `private` y `protected`. Desde C# 7.2, se introdujo el modificador `private protected`, que permite acceso desde clases derivadas dentro del mismo ensamblado.

Solución: Se agregó soporte para el modificador `private protected` en las reglas de acceso.

Expresiones Condicionales por Referencia

Problema: En versiones anteriores, las expresiones condicionales (`?:`) no podían devolver referencias. Desde C# 7.2, se permite que el resultado sea una referencia.

Solución: Se ajustó la gramática para permitir expresiones condicionales por referencia.

2.1.3. C# 7.3

Acceso a Campos Fijos Sin Fijación

Problema: En versiones anteriores de C#, para acceder a campos fijos era necesario fijar la dirección de memoria. Desde C# 7.3, se permite acceder a campos fijos sin necesidad de fijarlos explícitamente.

Solución: Se ajustaron las reglas gramaticales para permitir el acceso a campos fijos sin fijación.

Reasignación de Variables Locales Referenciadas

Problema: No se podía reasignar variables locales referenciadas. Desde C# 7.3, se permite la reasignación de variables locales que están marcadas como `ref`.

Solución: Se modificaron las reglas para permitir la reasignación de variables locales referenciadas.

Inicializadores en Arreglos Stackalloc

Problema: No era posible utilizar inicializadores con arreglos creados con `stackalloc`. Desde C# 7.3, se permite usar inicializadores en arreglos `stackalloc`.

Solución: Se ajustaron las reglas gramaticales para permitir inicializadores en arreglos `stackalloc`.

Sentencias Fijas con Tipos que Soportan Patrones

Problema: Las sentencias fijas solo podían usarse con tipos específicos. Desde C# 7.3, se permiten sentencias fijas con cualquier tipo que soporte

patrones.

Solución: Se modificó la regla `fixed_statement` para permitir el uso con cualquier tipo que soporte patrones.

Pruebas con Tipos Tupla

Problema: No se podía usar == y `!=` directamente con tipos tupla. Desde C# 7.3, se permite comparar tuplas usando estos operadores.

Solución: Se modificaron las reglas para permitir comparaciones directas entre tipos tupla.

Uso de Variables de Expresión en Más Ubicaciones

Problema: Las variables de expresión tenían un alcance limitado. Desde C# 7.3, se permite usar variables de expresión en más ubicaciones dentro del código.

Solución: Se ajustaron las reglas gramaticales para ampliar el uso de variables de expresión en diferentes contextos.

Resolución de Métodos Mejorada al Usar Argumentos Diferentes por `in`

Problema: La resolución de métodos al usar argumentos diferentes por `in` era confusa y poco clara. Desde C# 7.3, se mejoró este aspecto.

Solución: Se ajustaron las reglas gramaticales para mejorar la resolución de métodos al usar argumentos por `in`.

2.2. C# 8

Miembros Solo de Lectura `readonly`

Problema: No se podía definir miembros de solo lectura en las clases. Desde C# 8, se introdujeron los miembros solo de lectura, que permiten que las propiedades sean inmutables después de su inicialización.

Solución: Se modificó la regla `property_declaration` para permitir miembros solo de lectura `readonly`.

Métodos de Interfaz por Defecto

Problema: No era posible definir métodos en interfaces con una implementación por defecto. Desde C# 8, se introdujeron métodos de interfaz por defecto, permitiendo que las interfaces contengan métodos implementados.

Solución: Se ajustó la regla `interface_declaration` para permitir métodos de interfaz con implementación por defecto.

Mejoras en Coincidencia de Patrones

Problema: Las versiones anteriores de C# tenían limitaciones en los patrones disponibles. Desde C# 8, se introdujeron mejoras en la coincidencia de patrones, incluyendo expresiones `switch` y patrones de propiedades.

Solución: Se expandieron las reglas `pattern` para incluir expresiones `switch` y patrones mejorados.

Patrones de Propiedades

Problema: No había soporte para patrones que coincidieran con propiedades específicas. Desde C# 8, se introdujeron patrones de propiedades para facilitar la coincidencia con las propiedades de un objeto.

Solución: Se modificó la regla `pattern` para incluir patrones que coincidan con propiedades.

Patrones Tupla

Problema: No había soporte para coincidencias basadas en tuplas. Desde C# 8, se introdujeron patrones tupla para facilitar la coincidencia con estructuras de tuplas.

Solución: Se agregó soporte para patrones tupla en la regla `pattern`.

Patrones Posicionales

Problema: No había soporte para patrones que coincidieran con la posición de los elementos. Desde C# 8, se introdujeron patrones posicionales que permiten coincidir elementos en una posición específica.

Solución: Se agregó soporte para patrones posicionales en la regla `pattern`.

Declaraciones Using

Problema: Las declaraciones `using` debían estar al principio del archivo o dentro del bloque. Desde C# 8, se permite utilizar declaraciones `using` dentro del bloque de código.

Solución: Se modificó la regla `using_declaration` para permitir declaraciones `using` más flexibles.

Funciones Locales Estáticas

Problema: Las funciones locales no podían ser estáticas antes. Desde C# 8, se permite definir funciones locales como estáticas.

Solución: Se agregó soporte para funciones locales estáticas en la gramática.

Estructuras Ref Descartables

Problema: No había un manejo específico para estructuras ref descartables. Desde C# 8, se introdujo el soporte para estructuras ref descartables que permiten un manejo más eficiente de la memoria.

Solución: Se modificó la regla `struct_declaration` para permitir estructuras ref descartables.

Tipos Nullable Referencia

Problema: Todas las referencias eran consideradas no nulas. Desde C# 8, se introdujo el concepto de tipos nullable referencia que permite indicar explícitamente si una referencia puede ser nula.

Solución: Se ajustaron las reglas gramaticales para incluir el manejo de tipos nullable referencia.

Flujos Asincrónicos

Problema: No había un manejo eficiente para flujos asincrónicos. Desde C# 8, se introdujo el soporte para flujos asincrónicos que permite trabajar con secuencias de datos asincrónicas.

Solución: Se modificaron las reglas gramaticales para permitir flujos asincrónicos y su uso en bucles.

Índices y Rangos

Problema: No había una forma sencilla de trabajar con índices y rangos en colecciones. Desde C# 8, se introdujo el soporte para índices y rangos que facilita el acceso a elementos en colecciones.

Solución: Se agregó soporte para índices y rangos en la gramática.

Asignación Null-Coalescing

Problema: No había forma concisa de asignar valores cuando una variable es nula. Desde C# 8, se introdujo el operador null-coalescing assignment (`??=`).

Solución: Se ajustó la gramática para incluir el operador null-coalescing assignment.

Tipos Construidos No Administrados

Problema: No había un manejo específico para tipos construidos no administrados. Desde C# 8, se introdujo el soporte para tipos construidos no administrados que permiten trabajar con tipos sin administración del recolector de basura.

Solución: Se agregó soporte para tipos construidos no administrados en la gramática.

Stackalloc en Expresiones Anidadas

Problema: No era posible usar `stackalloc` dentro de expresiones anidadas. Desde C# 8, se permite usar `stackalloc` dentro de expresiones anidadas.

Solución: Se ajustó la gramática para permitir el uso de `stackalloc` en expresiones anidadas.

Mejora en las Cadenas Interpoladas Verbatim

Problema: Las cadenas interpoladas verbatim tenían limitaciones en su formato. Desde C# 8, se mejoraron las cadenas interpoladas verbatim para permitir más flexibilidad al incluir expresiones complejas.

Solución: Se modificó la regla `interpolated_string_expression` para mejorar el manejo de cadenas interpoladas verbatim.

2.3. C# 9

Registros `record`

Problema: No existía un tipo de datos específico para registros. Sin embargo, desde C# 9, se introdujeron los registros, que son tipos de referencia inmutables que facilitan la creación de objetos con propiedades.

Solución: Se modificó la regla `class_declaration` para permitir la definición de `record`.

Setters Solo de Inicialización `init`

Problema: Todas las propiedades podían ser modificadas después de su inicialización. Desde C# 9, se introdujo la posibilidad de definir propiedades con setters solo de inicialización, permitiendo que las propiedades sean asignadas solo durante la creación del objeto.

Solución: Se ajustó la regla `property_declaration` para incluir el modificador `init`.

Declaraciones en Nivel Superior

Problema: El código debía estar contenido dentro de una clase o espacio de nombres. Desde C# 9, se permite escribir declaraciones en nivel superior, lo que simplifica el código.

Solución: Se modificó la gramática para permitir declaraciones en nivel superior sin necesidad de definir una clase.

Mejoras en Coincidencia de Patrones

Problema: Las versiones anteriores de C# tenían limitaciones en los patrones disponibles. Desde C# 9, se introdujeron mejoras en la coincidencia de patrones, incluyendo patrones relacionales y lógicos.

Solución: Se expandieron las reglas `pattern` para incluir patrones relacionales (>, <, $\leq$, $\geq$) y lógicos(or, and, not).

Enteros de Tamaño Nativo

Problema: No había un tipo específico para enteros que se adaptara al tamaño nativo del sistema. Desde C# 9, se introdujeron enteros de tamaño nativo (`nint` y `nuint`).

Solución: Se agregó soporte para los tipos `nint` y `nuint` en la regla `type_`.

Punteros a Funciones

Problema: No era posible trabajar con punteros a funciones directamente. Desde C# 9, se introdujo el soporte para punteros a funciones.

Solución: Se agregó soporte para punteros a funciones en las reglas correspondientes.

Inicializadores de Módulo

Problema: No había una forma específica de inicializar módulos. Desde C# 9, se introdujeron inicializadores de módulo.

Solución: Se agregó soporte para inicializadores de módulo `[ModuleInitializer]` en la gramática.

Nuevas Funciones para Métodos Parciales

Problema: Las funciones parciales tenían limitaciones en su uso. Desde C# 9, se introdujeron nuevas características para métodos parciales.

Solución: Se ajustaron las reglas relacionadas con métodos parciales para incluir nuevas características.

Expresiones Nuevas con Tipos Objetivo

Problema: No había una forma concisa de crear instancias utilizando tipos objetivo. Desde esta versión, se permite crear expresiones tipadas directamente.

Solución: Se ajustó la gramática para permitir expresiones tipadas al usar `new`.

Funciones Anónimas Estáticas

Problema: Las funciones anónimas no podían ser estáticas anteriormente. Sin embargo, desde C# 9, se permite definir funciones anónimas como estáticas.

Solución: Se agregó soporte para funciones anónimas estáticas en la gramática.

Expresiones Condicionales Tipadas por Objetivo

Problema: Las expresiones condicionales no podían ser tipadas por objetivo antes. Desde C# 9, se permite que las expresiones condicionales tengan un tipo objetivo inferido.

Solución: Se modificó la gramática para permitir expresiones condicionales tipadas por objetivo.

Tipos Retornados Covariantes

Problema: Los tipos retornados covariantes $c\,?\,e_1 : e_2$ no eran compatibles. Sin embargo, desde C# 9, se introdujo el soporte para tipos retornados covariantes en métodos virtuales o sobrescritos.

Solución: Se ajustó la regla `method_declaration` para permitir tipos retornados covariantes.

Soporte para Extensiones GetEnumerator en Bucles `foreach`

Problema: El bucle `foreach` requería que las colecciones implementaran explícitamente `IEnumerable`. Con esta versión, se permite un soporte más flexible mediante extensiones `GetEnumerator`.

Solución: Se modificó la regla correspondiente al bucle `foreach` para incluir soporte para extensiones `GetEnumerator`.

Parámetros Descartados en Lambdas

Problema: Los parámetros descartados no eran compatibles en expresiones lambda. Sin embargo, desde C# 9, se permite usar parámetros descartados (_) en lambdas.

Solución: Se ajustó la regla `lambda_expression` para permitir parámetros descartados.

Atributos en Funciones Locales

Problema: Anteriormente no era posible aplicar atributos a funciones locales. Sin embargo, desde C# 9, se permite aplicar atributos a funciones locales definidas dentro de métodos.

Solución: Se modificó la regla correspondiente a los atributos para permitir su uso en funciones locales.

Uso del operador `with`

Problema: Antes de C# 9.0, no existía una forma concisa y directa de crear copias de objetos inmutables con algunas propiedades modificadas. Esto dificultaba la manipulación de objetos inmutables, ya que requería la creación manual de nuevos objetos con las propiedades deseadas.

Solución: Se modificó la regla `object_initializer` para permitir el uso del operador `with`, que permite a los desarrolladores crear un nuevo objeto basado en uno existente y modificar solo las propiedades especificadas. Esto se implementó mediante la adición de una nueva regla `with_expression` en la gramática, que permite la sintaxis `objeto with { propiedad = valor }`.

2.4. C# 10

Estructuras de Registro `record struct`

Problema: Las estructuras de registro no estaban disponibles. Sin embargo, desde C# 10, se introdujeron las estructuras de registro, que permiten definir estructuras con características de registro.

Solución: Se modificó la regla `struct_declaration` para permitir estructuras de registro.

Manejadores de Cadenas Interpoladas

Problema: No había una forma explícita de manejar las cadenas interpoladas. Sin embargo, desde C# 10, se introdujeron los manejadores de cadenas interpoladas para mejorar su manejo.

Solución: Se ajustó la regla `interpolated_string_expression` para reconocer correctamente las expresiones dentro de cadenas interpoladas.

Directivas `using` Globales

Problema: Las directivas `using` debían estar dentro de un espacio de nombres. Sin embargo, desde C# 10, se permiten directivas `using` globales que pueden aplicarse a todo el archivo.

Solución: Se modificó la regla `using_directive` para permitir directivas `using` globales.

Declaración de Espacios de Nombres `namespace` a Nivel de Archivo

Problema: Los espacios de nombres debían declararse utilizando llaves (`{}`). Sin embargo, desde C# 10, también es posible declararlos con punto y coma (`;`), permitiendo definiciones más concisas.

Solución: Se amplió la regla `namespace_declaration` para soportar ambas sintaxis.

Patrones de Propiedades Extendidos

Problema: Los patrones de propiedades no eran tan flexibles. Sin embargo, desde C# 10, se extendieron los patrones de propiedades para permitir más flexibilidad en la coincidencia de patrones.

Solución: Se modificó la regla `pattern` para soportar patrones de propiedades extendidos `property_pattern`.

Expresiones Lambda con Tipo Natural

Problema: Las expresiones lambda debían tener un tipo explícito. Sin embargo, desde C# 10, se permite que las expresiones lambda tengan un tipo natural, donde el compilador puede inferir el tipo delegado.

Solución: Se modificó la regla `lambda_expression` para permitir tipos naturales en expresiones lambda.

Expresiones Lambda con Tipo de Retorno Explícito

Problema: Las expresiones lambda no podían declarar un tipo de retorno explícito. Sin embargo, desde C# 10, se permite que las expresiones lambda declaren un tipo de retorno cuando el compilador no puede inferirlo.

Solución: Se modificó la regla `lambda_expression` para permitir tipos de retorno explícitos.

Atributos en Expresiones Lambda

Problema: Los atributos no podían aplicarse a expresiones lambda. Sin embargo, desde C# 10, se permite aplicar atributos a expresiones lambda.

Solución: Se modificó la regla `lambda_expression` para permitir atributos.

Inicialización de Cadenas Constantes con Interpolación

Problema: Las cadenas constantes no podían inicializarse con interpolación. Sin embargo, desde C# 10, se permite inicializar cadenas constantes utilizando interpolación si todos los marcadores son cadenas constantes.

Solución: Se ajustó la regla `constant_expression` para permitir interpolación en cadenas constantes.

Modificador `sealed` en `ToString` de Registros

Problema: No había una forma explícita de sellar el método `ToString` en tipos de registro. Sin embargo, desde C# 10, se permite agregar el modificador `sealed` cuando se sobrescribe `ToString` en un tipo de registro.

Solución: Se modificó la regla `method_declaration` para permitir el modificador `sealed` en `ToString`.

Advertencias para Asignación Definida y Análisis de Estado Nulo

Problema: Las advertencias para asignación definida y análisis de estado nulo no eran tan precisas. Sin embargo, desde C# 10, se mejoraron estas advertencias para ser más precisas.

Solución: Se ajustó la regla `expression` para mejorar las advertencias relacionadas con la asignación definida y el análisis de estado nulo.

Declaración y Asignación en Deconstrucción

Problema: La deconstrucción solo permitía declarar variables. Sin embargo, desde C# 10, se permite tanto declarar como asignar valores en la misma deconstrucción.

Solución: Se modificó la regla `deconstruction_expression` para permitir tanto declaraciones como asignaciones.

Atributo `AsyncMethodBuilder`

Problema: No había una forma explícita de marcar métodos como constructores de métodos asíncronos. Sin embargo, desde C# 10, se introdujo el atributo `AsyncMethodBuilder` para indicar métodos que construyen métodos asíncronos.

Solución: Se agregó soporte para el atributo `AsyncMethodBuilder`.

Atributo `CallerArgumentExpression`

Problema: No había una forma explícita de obtener la expresión del argumento del llamador. Sin embargo, desde C# 10, se introdujo el atributo `CallerArgumentExpression` para obtener esta información.

Solución: Se agregó soporte para el atributo `CallerArgumentExpression`.

Nuevo Formato para la Directiva `#line`

Problema: La directiva `#line` no tenía un formato flexible. Sin embargo, desde C# 10, se introdujo un nuevo formato para esta directiva que permite más flexibilidad.

Solución: Se modificó la regla `line_directive` para soportar el nuevo formato.

2.5. C# 11

Literales de Cadena Raw

Problema: Las cadenas literales no podían contener caracteres especiales sin escaparlos. Sin embargo, desde C# 11, se introdujeron los literales de cadena raw, que permiten cadenas sin necesidad de escapar caracteres especiales.

Solución: Se agregó soporte para los literales de cadena raw en la regla `string_literal`.

Soporte Genérico para Operaciones Matemáticas

Problema: Las operaciones matemáticas no podían ser genéricas. Sin embargo, desde C# 11, se introdujo el soporte genérico para operaciones matemáticas, permitiendo definir métodos que trabajen con tipos numéricos genéricos.

Solución: Se modificó la regla `type_` para permitir tipos genéricos en operaciones matemáticas.

Atributos Genéricos

Problema: Los atributos no podían ser genéricos. Sin embargo, desde C# 11, se introdujo el soporte para atributos genéricos, permitiendo definir atributos que acepten tipos genéricos.

Solución: Se modificó la regla `attribute` para permitir atributos genéricos.

Literales de Cadena UTF-8

Problema: Las cadenas literales no podían especificar codificación UTF-8 explícitamente. Sin embargo, desde C# 11, se introdujo el soporte para literales de cadena UTF-8, permitiendo especificar la codificación explícitamente.

Solución: Se agregó soporte para literales de cadena UTF-8 en la regla `string_literal`.

Saltos de Línea en Expresiones Interpoladas

Problema: Las expresiones interpoladas no permitían saltos de línea dentro de la cadena. Sin embargo, desde C# 11, se permite incluir saltos de línea en expresiones interpoladas.

Solución: Se modificó la regla `interpolated_string_expression` para permitir saltos de línea.

Patrones de Lista

Problema: Los patrones no podían aplicarse a listas. Sin embargo, desde C# 11, se introdujo el soporte para patrones de lista, permitiendo comparar listas con patrones específicos.

Solución: Se agregó soporte para patrones de lista en la regla `pattern`.

Tipos Locales a Archivo

Problema: Los tipos definidos debían ser accesibles desde cualquier parte del archivo. Sin embargo, desde C# 11, se introdujo el soporte para tipos locales a archivo, permitiendo definir tipos que solo sean accesibles dentro del mismo archivo.

Solución: Se modificó la regla `class_definition` para permitir tipos locales a archivo.

Miembros Obligatorios

Problema: No había una forma explícita de marcar miembros como obligatorios. Sin embargo, desde C# 11, se introdujo el soporte para miembros obligatorios, permitiendo definir miembros que deben ser inicializados.

Solución: Se agregó soporte para miembros obligatorios en la regla `property_declaration`.

Estructuras con Valores Predeterminados

Problema: Las estructuras no tenían valores predeterminados. Sin embargo, desde C# 11, se introdujo el soporte para estructuras con valores predeterminados, permitiendo que las estructuras tengan valores por defecto.

Solución: Se modificó la regla `struct_declaration` para permitir valores predeterminados en estructuras.

Coincidencia de Patrones con `Span<char>`

Problema: Los patrones no podían aplicarse a `Span<char>`. Sin embargo, desde C# 11, se introdujo el soporte para coincidir patrones con `Span<char>`.

Solución: Se agregó soporte para patrones con `Span<char>` en la regla `pattern`.

Alcance Extendido de `nameof`

Problema: El operador `nameof` solo podía aplicarse a nombres de tipos y miembros. Sin embargo, desde C# 11, se extendió el alcance de `nameof` para permitir su uso en más contextos.

Solución: Se modificó la regla `expression` para permitir el uso extendido de `nameof`.

Tipo `IntPtr` Numérico

Problema: El tipo `IntPtr` no era numérico. Sin embargo, desde C# 11, se introdujo el soporte para que `IntPtr` sea un tipo numérico.

Solución: Se modificó la regla `type_` para permitir que `IntPtr` sea un tipo numérico.

Campos `ref` y `scoped ref`

Problema: Los campos no podían ser `ref`. Sin embargo, desde C# 11, se introdujo el soporte para campos `ref` y `scoped ref`, permitiendo definir campos que se comporten como referencias.

Solución: Se agregó soporte para campos `ref` en la regla `field_declaration`.

Mejora en la Conversión de Grupos de Métodos a Delegados

Problema: La conversión de grupos de métodos a delegados no era tan flexible. Sin embargo, desde C# 11, se mejoró esta conversión para permitir más flexibilidad en la asignación de métodos a delegados.

Solución: Se modificó la regla `expression` para mejorar la conversión de grupos de métodos a delegados.

2.6. C# 12

Constructores Primarios

Problema: Los constructores debían definirse explícitamente dentro de una clase o estructura. Desde C# 12, se permite crear constructores primarios en cualquier tipo de clase o estructura.

Solución: Se modificó la regla `class_definition` para permitir constructores primarios. Esto simplifica la definición de clases y estructuras al combinar la declaración de parámetros con la inicialización de campos.

Expresiones de Colección

Problema: Las colecciones se creaban utilizando el operador `new[]`. Desde C# 12, se introdujo una nueva sintaxis para expresiones de colección, incluyendo el elemento de propagación (`..e`).

Solución: Se agregó una nueva regla para soportar las expresiones de colección, definiendo la sintaxis de colecciones en línea y el operador de propagación (`..e`).

Arrays en Línea

Problema: Los arrays debían declararse con un tamaño fijo utilizando llaves ({). Desde C# 12, se permite declarar arrays en línea dentro de estructuras.

Solución: Se modificó la regla `type_` para permitir arrays en línea, simplificando la declaración de estructuras con arrays de tamaño fijo.

Parámetros Opcionales en Expresiones Lambda

Problema: Los parámetros en expresiones lambda no podían ser opcionales. Desde C# 12, se permite el uso de parámetros opcionales en expresiones lambda.

Solución: Se actualizó la regla `explicit_anonymous_function_parameter` para soportar parámetros opcionales.

Parámetros `ref readonly`

Problema: Los parámetros en expresiones lambda no podían ser `ref readonly`. Desde C# 12, se permite el uso de parámetros `ref readonly` en expresiones lambda.

Solución: Se modificó la regla `explicit_anonymous_function_parameter` para soportar parámetros `ref readonly`.

Alias Any Type

Problema: El alias de tipo solo se podía utilizar con tipos nombrados. Desde C# 12, se permite utilizar el alias de tipo con cualquier tipo.

Solución: Se actualizó la regla `using_directive` para permitir alias de cualquier tipo.

Atributo Experimental

Problema: No había una forma explícita de marcar características experimentales. Desde C# 12, se introdujo el atributo `Experimental`.

Solución: Se agregó soporte para el atributo `Experimental`.

Capítulo 3

Extracción de features del AST

3.1. Extraccion del AST

Se emplea ANTLR para extraer Árboles de Sintaxis Abstracta (AST) a partir del código fuente de programas escritos en C#. Convierte una gramática de lenguaje en código que puede generar un árbol de sintaxis.
A continuación, se explica su funcionamiento y las etapas principales en su flujo de trabajo:

I. **Definición de la Gramática:** Primero, se define la gramática del lenguaje en un archivo `.g4`. Este archivo incluye:

   - **Reglas léxicas (tokens):** especifican los elementos más básicos, como palabras clave, identificadores, operadores, números, etc.
   - **Reglas sintácticas:** describen cómo se combinan los tokens para formar sentencias válidas en el lenguaje.

   Cada regla léxica o sintáctica tiene un nombre y una expresión que define qué secuencias de caracteres o estructuras pueden corresponderse con esa regla.

II. **Generación del Lexer y el Parser:** ANTLR toma el archivo de gramática `.g4` y genera clases en un lenguaje de programación (como Java, Python, C#, etc.) que implementan el lexer (analizador léxico) y el parser (analizador sintáctico).

- **Lexer:** identifica tokens en el texto de entrada. Por ejemplo, en una expresión matemática, reconoce números, operadores, paréntesis, etc.
- **Parser:** usa estos tokens para construir una estructura jerárquica que representa la gramática definida, lo cual permite reconocer la estructura completa del código o texto de entrada.

III. **Análisis del Código o Texto de Entrada:** Con el lexer y parser generados, se puede analizar el código fuente. Este proceso produce un árbol de sintaxis que representa la estructura jerárquica del código según la gramática.

IV. **Creación del Árbol de Sintaxis Abstracta (AST):** ANTLR facilita la creación de un AST, una representación simplificada que retiene la estructura lógica del código, omitiendo detalles innecesarios para ciertos tipos de análisis.

V. **Recorridos y Transformaciones en el AST:** Una vez construido el AST, es posible recorrerlo para realizar análisis adicionales, transformaciones o para interpretar el código. ANTLR proporciona métodos para recorrer este árbol y manipular los nodos según las reglas definidas en la gramática, en este caso se utilizó el listener llamado **CSharpParserListener**, proporcionado por ANTLR para recorrer el árbol y extraer los features.

3.2. Extraccion de features

En el análisis de similitud de código y detección de patrones, es necesario extraer características relevantes que capturen la estructura y el comportamiento del código. Estas características, conocidas como **features**, representan los aspectos más importantes de los datos, permitiendo analizar y comparar fragmentos de código de manera efectiva. Para este propósito, se implementó una clase llamada **FeatureExtractorListener**, que extiende la funcionalidad de CSharpParserListener para analizar el código fuente en C#. A continuación, se presenta una descripción detallada del proceso de extracción de características y la importancia de cada una.

I. Estructura del AST:

   - **total_nodes:** Número total de nodos en el AST.
   - **max_depth:** Profundidad máxima del AST.

II. Declaraciones y Variables:

   - **variables:** Número de variables locales.
   - **constants:** Número de constantes declaradas.
   - **variable_names:** Conjunto de nombres de variables y sus tipos.
   - **number_of_tuples:** Número de variables de tipo tupla.
   - **lists:** Número de listas declaradas.
   - **dicts:** Número de diccionarios declarados.

   Las variables y constantes son elementos clave en cualquier programa, ya que almacenan y mantienen valores que pueden cambiar o permanecer fijos durante la ejecución. Comprender el uso de estas entidades en el código permite analizar cómo se manipulan los datos, identificar el flujo de información y observar cómo evolucionan los valores a lo largo del programa. Las variables revelan el comportamiento dinámico del código, mientras que las constantes indican valores fijos que definen parámetros o condiciones estables dentro del flujo de ejecución.

   Además, la variedad y el tipo de estructuras de datos empleadas, como tuplas, listas y diccionarios, aportan información importante sobre el enfoque y la complejidad del código. Por ejemplo, el uso de estructuras de datos más avanzadas, como diccionarios anidados o listas de objetos, puede reflejar una mayor abstracción y modularidad en el diseño, mientras que estructuras más simples pueden indicar un código directo y menos complejo. Estas elecciones también proporcionan información sobre el estilo de programación del autor y sus preferencias en cuanto a la organización y manipulación de datos.

III. Declaraciones de Métodos y Clases:

   - **methods:** Número de métodos declarados.

- **method_names:** Conjunto de nombres de métodos.
- **method_return_types:** Conjunto de tipos de retorno de métodos.
- **method_parameters:** Lista de parámetros de métodos.
- **classes:** Número de clases declaradas.
- **class_names:** Conjunto de nombres de clases.
- **abstract_classes:** Número de clases abstractas.
- **sealed_classes:** Número de clases selladas.
- **interfaces:** Número de interfaces declaradas.
- **interface_names:** Conjunto de nombres de interfaces.

La estructura y los nombres de los métodos y clases ofrecen información clave sobre la organización, modularidad y legibilidad del código. Los nombres de métodos y clases, cuando están bien definidos y siguen convenciones de nomenclatura clara, actúan como una especie de documentación implícita, ayudando a comprender la función y el propósito de cada componente sin necesidad de examinar cada detalle interno.

Los métodos y sus parámetros son esenciales para entender la funcionalidad del código. Los métodos representan acciones específicas y los parámetros definen los datos con los que esas acciones trabajan. Al analizar los métodos y los tipos de parámetros que aceptan, se puede deducir cómo las distintas partes del código interactúan y colaboran para realizar tareas. La estructura de los métodos, su nivel de abstracción, y la forma en que interactúan con otros componentes del código revelan la profundidad de la modularidad y el diseño de la aplicación, lo que facilita el análisis de patrones y la identificación de similitudes entre diferentes fragmentos de código.

Por otro lado, las clases y sus tipos (como clases abstractas o selladas) indican la arquitectura y el paradigma de diseño de la aplicación. Las clases abstractas, por ejemplo, representan conceptos generales que definen una estructura básica sin implementación completa, alentando la reutilización y la extensibilidad en el diseño del sistema. Las clases selladas (sealed) limitan la herencia, sugiriendo un diseño más controlado y dirigido a la especificidad. Estas elecciones de

diseño reflejan la intención del desarrollador en cuanto a la extensibilidad, la reutilización y la encapsulación, todos ellos principios fundamentales en la programación orientada a objetos.

IV. Estructuras de Control:

- **control_structures_if:** Número de sentencias if.
- **control_structures_switch:** Número de sentencias switch.
- **control_structures_for:** Número de bucles for.
- **control_structures_while:** Número de bucles while.
- **control_structures_dowhile:** Número de bucles do-while.
- **try_catch_blocks:** Número de bloques try-catch.

Las estructuras de control son fundamentales para comprender el flujo del programa y su lógica. Un mayor número de estructuras de control indica una lógica más compleja y ramificada.

V. Modificadores y Accesibilidad:

- **access_modifiers_public:** Número de elementos públicos.
- **access_modifiers_private:** Número de elementos privados.
- **access_modifiers_protected:** Número de elementos protegidos.
- **access_modifiers_internal:** Número de elementos internos.
- **access_modifiers_static:** Número de elementos estáticos.
- **access_modifiers_protected_internal:** Número de elementos protegidos internos.
- **access_modifiers_private_protected:** Número de elementos privados protegidos.

Los modificadores de acceso proporcionan información sobre la encapsulación y visibilidad de los componentes del código. La prevalencia de ciertos modificadores puede indicar prácticas de diseño y seguridad en el código.

VI. Modificadores Específicos:

- **modifier_readonly:** Número de elementos readonly.

- **modifier_volatile:** Número de elementos volatile.
- **modifier_virtual:** Número de elementos virtual.
- **modifier_override:** Número de elementos override.
- **modifier_new:** Número de elementos new.
- **modifier_partial:** Número de elementos partial.
- **modifier_extern:** Número de elementos extern.
- **modifier_unsafe:** Número de elementos unsafe.
- **modifier_async:** Número de elementos async.

Estos modificadores específicos reflejan patrones de diseño, control y comportamiento que van más allá de la simple estructura superficial del código. Al analizar modificadores como readonly, volatile o async, se capturan detalles sobre cómo el código maneja la concurrencia, la inmutabilidad y la asincronía, se utilizan para identificar similitudes en la lógica y el flujo de ejecución. Modificadores como virtual y override indican una arquitectura orientada a objetos, esto permite comparar el grado de extensibilidad y personalización en diferentes fragmentos de código. Además, la presencia de unsafe y extern sugiere que el código interactúa con recursos de bajo nivel o externos, lo que proporciona información sobre la complejidad y las dependencias de cada implementación.

VII. Llamadas a Librerías y LINQ(Language Integrated Query):

- **library_call_console:** Número de llamadas a la librería Console.
- **library_call_math:** Número de llamadas a la librería Math.
- **linq_queries_select:** Número de consultas LINQ Select.
- **linq_queries_where:** Número de consultas LINQ Where.
- **linq_queries_orderBy:** Número de consultas LINQ OrderBy.
- **linq_queries_groupBy:** Número de consultas LINQ GroupBy.
- **linq_queries_join:** Número de consultas LINQ Join.
- **linq_queries_sum:** Número de consultas LINQ Sum.
- **linq_queries_count:** Número de consultas LINQ Count.

Las llamadas a librerías y las consultas LINQ ofrecen información sobre cómo el código aprovecha las funcionalidades estándar y gestiona la manipulación de datos. Al utilizar llamadas a librerías, el código accede a un conjunto de herramientas predefinidas y optimizadas. Esto permite deducir la experiencia y el estilo del programador en términos de modularidad y adaptabilidad, aspectos clave en la estructura y lógica del código.

Por otro lado, el uso de consultas LINQ para manipular y consultar colecciones de datos refleja un enfoque específico en la optimización y claridad de la manipulación de datos en .NET. LINQ permite un acceso eficiente a estructuras de datos complejas, proporcionando una sintaxis uniforme para realizar operaciones como filtrado, proyección, agrupación y ordenación. La presencia de consultas LINQ en el código puede indicar la preferencia del desarrollador por una sintaxis declarativa y una gestión avanzada de colecciones, que resulta fundamental para identificar similitudes en la forma en que diferentes fragmentos de código manejan datos.

VIII. Otras Características:

- **number_of_lambdas:** Número de expresiones lambda.
- **number_of_getters:** Número de métodos get.
- **number_of_setters:** Número de métodos set.
- **number_of_namespaces:** Número de espacios de nombres.
- **enums:** Número de enumeraciones.
- **enum_names:** Conjunto de nombres de enumeraciones.
- **delegates:** Número de delegados.
- **delegate_names:** Conjunto de nombres de delegados.
- **node_count:** Conteo de nodos por tipo.

Estas características adicionales permiten analizar el código desde distintas perspectivas que revelan aspectos de diseño y organización. Por ejemplo, el número de expresiones lambda indica la frecuencia de uso de funciones anónimas, lo cual puede reflejar una orientación hacia la programación funcional. La cantidad de métodos de acceso

ofrece una idea del manejo de encapsulamiento y control de atributos en las clases. La presencia de espacios de nombres sugiere la estructura modular del código y la separación de responsabilidades, lo que facilita la organización y evita conflictos de nombres. Las enumeraciones y los delegados muestran la variedad de estructuras y tipos personalizados utilizados. Finalmente, el conteo de nodos por tipo permite una visión detallada de los elementos específicos del árbol de sintaxis abstracta, lo cual es útil para evaluar la complejidad y el tipo de construcciones empleadas.

La extracción de características con FeatureExtractorListener permite capturar aspectos relevantes del código fuente en C#, desde su estructura y complejidad hasta los patrones de diseño y las prácticas de programación. La implementación y el análisis detallado de estas características proporcionan una base sólida para mejorar la precisión de las herramientas de análisis de código.

Capítulo 4

Preparacion del dataset

Para preparar el dataset, se convierten los nombres de variables, métodos y otros identificadores de tipo string en vectores de características numéricas, lo cual permite que un modelo de machine learning procese el código de manera efectiva. Este proceso de transformación se realiza utilizando embeddings, una técnica de procesamiento del lenguaje natural, mediante el modelo Word2Vec [11]. Estos embeddings se combinan con otras características numéricas extraídas del código, como el número de métodos o la cantidad de expresiones lambda, lo cual forma un vector de características completo. Este vector integrado proporciona una descripción multidimensional del código, capturando tanto la estructura como la semántica en una única representación, lo que mejora la capacidad del modelo para detectar patrones y realizar comparaciones entre diferentes fragmentos de código.

4.1. Construcción del dataset

El dataset de códigos en C# está compuesto por pares de archivos que mantienen la misma funcionalidad pero presentan variaciones en la estructura y nomenclatura. Cada par está compuesto por un archivo original y su respectiva copia, con cambios mínimos y modificaciones que buscan simular escenarios típicos en los que el código mantiene su esencia lógica pero experimenta alteraciones en la sintaxis y organización, abarcan desde algoritmos fundamentales de programación hasta aplicaciones prácticas en la vida diaria.

Se eligieron las principales inteligencias artificiales generativas mas utilizados para la generación de código pues cuenta con 300 archivos generados por ChatGPT, 300 por Copilot, 150 con Perplexity y 90 con Phind donde la mitad son originales y la mitad son copias, teniendo un total de 1000 códigos.

I. **Creación de Código Base**: Se crearon múltiples fragmentos de código en C# que abarcan una variedad de funciones comunes, como cálculos geométricos, operaciones matemáticas, manipulación de cadenas de texto y validaciones de números. Estos fragmentos de código están diseñados para ejecutar funciones claras y específicas.

II. **Generación de Copias con Modificaciones**: Para cada archivo original, se generó una copia modificada siguiendo varias estrategias de variación:

   - *Renombramiento de Clases, Métodos y Variables*: Los nombres de clases, métodos y variables fueron modificados en las copias para simular diferencias en nomenclatura sin cambiar el objetivo funcional del código.
   - *Modificación de Estructura de Código*: Se alteraron estructuras sintácticas, como el tipo de bucles (de `for` a `while`) o la reestructuración de condiciones lógicas. Estas modificaciones permiten que la copia mantenga la misma funcionalidad que el original, pero con una apariencia diferente en términos de código.
   - *Ajustes en la Organización*: Algunos fragmentos fueron reorganizados en cuanto al orden de ejecución o el uso de estructuras de control alternativas, manteniendo los resultados finales consistentes con el código original.

4.2. Dataset de diferencias

Para maximizar la cantidad de datos disponibles y reflejar de manera efectiva la similitud entre proyectos, se creó un dataset que contiene todos los pares posibles (2 a 2) de proyectos del conjunto de datos original. En este proceso, para cada par de proyectos, se calcula y almacena la diferencia entre sus vectores correspondientes.

Este enfoque tiene varias ventajas significativas:

- **Incremento en la Cantidad de Datos:** Generar todos los pares posibles de proyectos incrementa exponencialmente el número de instancias en el dataset, proporcionando una base de datos más rica y diversa para entrenar modelos de aprendizaje automático.
- **Etiquetado Automático de Datos:** Una ventaja de calcular la distancia entre los datos es que permite disponer de algunos datos etiquetados. Aunque no se puede afirmar si un proyecto individual es original o una copia de otro proyecto, al calcular las distancias dos a dos, se puede asegurar que un par de proyectos provenientes de diferentes tipos de proyectos no son copias uno del otro. Esto proporciona etiquetas adicionales que mejoran la calidad del entrenamiento del modelo.
- **Captura de Relaciones Detalladas:** Al almacenar la diferencia entre los vectores de cada par de proyectos, se capturan las distancias y relaciones específicas entre todos los proyectos. Esto permite que el modelo de machine learning pueda aprender las sutilezas de las similitudes y diferencias entre distintos proyectos.
- **Mejora en la Precisión del Modelo:** Con un mayor volumen de datos y la inclusión de las distancias entre pares, se espera que el modelo tenga un mejor desempeño en la tarea de detección de similitudes. La precisión del modelo se ve beneficiada al disponer de más ejemplos que reflejan una amplia gama de variaciones y similitudes.
- **Refinamiento de las Métricas de Similitud:** Este método permite que se utilicen métricas de similitud precisas, ya que cada par de proyectos se compara de manera detallada.

En resumen, la creación de este dataset con todos los pares posibles y sus diferencias vectoriales no solo aumenta la cantidad de datos disponibles, sino que también enriquece la información sobre las relaciones entre proyectos, mejorando así la capacidad del modelo para detectar similitudes de manera precisa y eficiente.

4.3. Word2Vec

En el contexto del análisis de similitud de código, los nombres de variables, métodos y otros identificadores en el código fuente proporcionan información semántica sobre la funcionalidad y el propósito de diferentes partes del código. Por ejemplo, los identificadores que se usan de manera similar en diferentes contextos tendrán embeddings[1] similares. Sin embargo, los identificadores en el código no están estructurados de manera que las máquinas puedan comprender fácilmente sus relaciones semánticas, para esto se utilizó Word2Vec. El proceso involucró los siguientes pasos:

I. *Extracción de Identificadores*: Se extrajeron todos los nombres de variables, métodos, clases, interfaces, enumeraciones y delegados del código fuente utilizando la clase FeatureExtractorListener.

II. *Entrenamiento de Word2Vec*: Se utilizó un corpus de identificadores extraídos de múltiples proyectos de C# para entrenar el modelo Word2Vec. El modelo aprendió las relaciones semánticas entre los diferentes identificadores en el contexto del código.

III. *Conversión a Embeddings*: Cada identificador extraído se convirtió en un vector de características numéricas utilizando el modelo Word2Vec entrenado. Luego se halla el promedio entre todos los vectores por feature correspondiente para asegurar que todos las caracteristicas de vectores tengan la misma dimensión. Estos vectores capturan la semántica y el contexto de los identificadores en el código.

[1] Los embeddings son una técnica de procesamiento de lenguaje natural que convierte el lenguaje humano en vectores matemáticos. Estos vectores son una representación del significado subyacente de las palabras, lo que permite que las computadoras procesen el lenguaje de manera más efectiva.

Capítulo 5

Redes Neuronales Siamesas para la Similitud de Código

Las redes neuronales siamesas se emplearon para analizar y capturar tanto la estructura como el comportamiento lógico del código. Este enfoque facilita la identificación de patrones, relaciones textuales y sintácticas, proporcionando una mayor precisión en los resultados obtenidos. En este capítulo se detallan los componentes esenciales de esta arquitectura y el proceso llevado a cabo para su entrenamiento.

5.1. Concepto de Red Siamesa

Una red neuronal siamesa consta de dos subredes idénticas que procesan dos entradas en paralelo y producen representaciones vectoriales latentes. La arquitectura básica es la siguiente:

- **Entrada:** Dos vectores x_1 y x_2 de tamaño n.
- **Subred Base:** Ambas entradas se procesan mediante una red compartida f_θ, obteniendo:

$$h_1 = f_\theta(x_1), \quad h_2 = f_\theta(x_2).$$

- **Métrica de Distancia:** La similitud entre las representaciones latentes se mide mediante una distancia, como:

$$d(h_1, h_2) = ||h_1 - h_2||_p,$$

donde $p = 1$ para la distancia $L1$ o $p = 2$ para la distancia Euclidiana.

- **Clasificación Binaria:** La distancia se pasa por una capa de activación sigmoidal para obtener una probabilidad:

$$\hat{y} = \sigma(W \cdot d(h_1, h_2) + b).$$

5.2. Implementación del Modelo

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

5.2.1. Subred Base

La subred base transforma las entradas crudas $x \in \mathbb{R}^{65}$ en representaciones vectoriales compactas y abstractas. Este proceso ayuda al modelo a identificar patrones relevantes y eliminar información redundante.

I. **Capa densa (*Dense*):** Cada capa densa aplica una transformación lineal seguida de una función de activación no lineal **ReLU** (Rectified Linear Unit). 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.
- $h^{(l)}$ es la salida de la capa l.

II. **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.

III. **Arquitectura detallada:** La subred consta de:

- 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 es compartida entre las dos ramas del modelo siamesa.

5.2.2. Construcción del Modelo Siamesa

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

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

II. **Representaciones Latentes:** Cada entrada se transforma en una representación latente compacta mediante la subred base:

$$h_1 = f_\theta(x_1), \quad h_2 = f_\theta(x_2),$$

donde f_θ representa la subred base con parámetros compartidos θ.

III. **Cálculo de la Distancia L1:** Para medir la similitud entre las representaciones latentes, se calcula la distancia L1:

$$d(h_1, h_2) = |h_1 - h_2|.$$

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

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

$$\hat{y} = \sigma(W \cdot d(h_1, h_2) + b),$$

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, interpretado como la probabilidad de similitud.

5.2.3. Función de Pérdida

El modelo se entrena minimizando la entropía cruzada binaria, que mide la discrepancia entre las probabilidades predichas ($\hat{y}$) y las etiquetas verdaderas (y):

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \cdot \log(\hat{y}_i) + (1-y_i)\cdot \log(1-\hat{y}_i)\Big],$$

donde:

- $y_i \in \{0,1\}$ indica si los pares de entrada son similares (1) o diferentes (0).
- $\hat{y}_i \in [0,1]$ es la probabilidad predicha por el modelo para el par i.

Esta función penaliza predicciones incorrectas de manera proporcional a su magnitud.

5.2.4. Optimización

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

- **Momentum:** Acelera el entrenamiento en direcciones consistentes.
- **Tasas de aprendizaje adaptativas:** Ajusta la tasa de aprendizaje individual para cada parámetro.

El algoritmo de Adam se define como:

I. Estimación del primer momento (media):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t,$$

donde g_t es el gradiente en el paso t.

II. Estimación del segundo momento (varianza):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2.$$

III. Actualización de los parámetros:

$$\theta_{t+1} = \theta_t - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon}.$$

Los hiperparámetros típicos son $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$, y $\epsilon = 10^{-8}$. Adam combina estabilidad y velocidad, logrando convergencia eficiente en problemas complejos.

5.3. Proceso de Entrenamiento y Predicción

En esta sección, explicamos cómo se lleva a cabo el proceso de entrenamiento y predicción en el modelo siamesa, enfatizando cómo los pares de datos son utilizados para aprender las representaciones y cómo se realiza la inferencia posterior para predecir la similitud entre las entradas.

5.3.1. Entrenamiento

El modelo siamesa se entrena utilizando un conjunto de pares 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 todo el conjunto de entrenamiento. La optimización se realiza utilizando el algoritmo de Adam, como se mencionó anteriormente, que es capaz de actualizar los pesos de manera eficiente. El proceso de entrenamiento se realiza mediante el método `model.fit`, el cual toma los datos de entrada y las etiquetas correspondientes:

$$\text{model.fit}([x_1, x_2], y, \text{validation\_data} = ([x_1^{val}, x_2^{val}], y^{val})).$$

Aquí, $[x_1, x_2]$ son las dos entradas que representan un par de datos de entrenamiento, y y es la etiqueta correspondiente. Además, se proporciona un conjunto de validación $([x_1^{val}, x_2^{val}], y^{val})$ para evaluar el rendimiento del

modelo durante el entrenamiento y prevenir el sobreajuste. La validación permite que el modelo se ajuste y aprenda sin memorizar excesivamente los datos de entrenamiento, lo que puede llevar a una mala generalización.

El modelo procesa cada par de entradas x_1, x_2 a través de la subred base, calculando sus representaciones latentes h_1 y h_2. Luego, calcula la distancia entre estas representaciones y utiliza esta información para realizar 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 discrepancia entre las predicciones $\hat{y}$ y las etiquetas y.

5.3.2. Predicción

Una vez que el modelo ha sido entrenado, se puede utilizar para hacer predicciones sobre nuevos pares de datos (x_1, x_2). En este caso, el modelo toma las dos entradas x_1 y x_2, las pasa a través de la subred base para obtener sus representaciones latentes h_1 y h_2, y luego calcula la distancia entre ellas utilizando la métrica L1.

A continuación, el modelo predice la similitud entre los dos elementos utilizando la fórmula:

$$\text{similarity} = \text{model.predict}([x_1, x_2]).$$

El valor de `similarity` es un valor entre 0 y 1, que indica qué tan similares son las dos entradas. Un valor cercano a 0 indica que las entradas son similares, mientras que un valor cercano a 1 indica que son disímiles. Esto se debe a que el modelo genera una probabilidad, y la activación sigmoidal transforma la distancia entre las representaciones latentes en un valor de probabilidad. La salida final, $\hat{y}$, se interpreta como la probabilidad de que las dos entradas pertenezcan a la misma clase (es decir, que sean similares).

Capítulo 6

Experimentos y resultados

En este capítulo se presentan las pruebas que se hicieron para evaluar el desempeño de los modelos.

Tabla 6.1 Resultados

Método	**acc**	**loss**	**val acc**	**val loss**
L1 distance	0.9893	0.0626	0.9925	0.0577
normal	0.9922	0.0736	0.9925	0.0728

Conclusiones

El análisis de similitud de código ha avanzado desde sus inicios, convirtiéndose en una herramienta para abordar problemas en los ámbitos académico. En esta tesis, se analizan los avances en representación estructural, como el uso de Árboles de Sintaxis Abstracta (AST), y el desarrollo de modelos basados en aprendizaje profundo que permiten detectar similitudes en código más allá de comparaciones textuales. Los AST, al capturar la estructura jerárquica del código, permiten realizar comparaciones basadas en lógica y semántica, evitando la dependencia de detalles superficiales. El proceso de actualización de la gramática de lenguajes como C# garantiza la representación de nuevas características en los AST, manteniendo la aplicabilidad de las herramientas de análisis frente a la evolución de los lenguajes de programación.

Las redes neuronales siamesas permiten comparar representaciones vectoriales de fragmentos de código, incluso en casos con modificaciones superficiales. Estas redes, configuradas para procesar pares de entradas mediante ramas idénticas con pesos compartidos, miden similitudes enfocándose en las características más relevantes. Este enfoque permite detectar relaciones semánticas y estructurales que no pueden captarse mediante métodos tradicionales basados en características superficiales.

En conclusión, esta investigación evalúa las metodologías propuestas y establece un marco para el desarrollo de herramientas más precisas y adaptadas. Las contribuciones de esta tesis, que incluyen el uso de AST, el refinamiento de redes neuronales siamesas y la actualización de gramáticas, plantean oportunidades para investigaciones futuras en el análisis de similitud de código.

Recomendaciones

Optimización y Escalabilidad

- Explorar métodos avanzados para optimizar la representación y comparación de Árboles de Sintaxis Abstracta (AST), buscando reducir el tiempo de procesamiento y los requerimientos computacionales en grandes conjuntos de datos. Esto podría incluir:
  - **Paralelización de Procesos:** Implementar algoritmos paralelos o distribuidos para dividir el análisis en múltiples núcleos de procesamiento o máquinas.
  - **Compresión de Estructuras:** Diseñar representaciones más compactas de los AST, como árboles binarios optimizados, que permitan una comparación más rápida y eficiente.
  - **Preprocesamiento Inteligente:** Incorporar mecanismos de filtrado previo que identifiquen rápidamente fragmentos irrelevantes o claramente distintos, reduciendo el tamaño efectivo del conjunto de datos.

Ampliación de la Herramienta

- Expandir la compatibilidad de la herramienta para incluir múltiples lenguajes de programación. Esto requeriría:
  - Adaptación de gramáticas específicas para cada lenguaje soportado.
  - Desarrollo de un módulo de detección automática del lenguaje de entrada.

    - Evaluación de la efectividad de los análisis en lenguajes dinámicos (como Python) y lenguajes fuertemente tipados (como Java o C++).

- Incorporar módulos adicionales para analizar similitudes en paradigmas específicos, como programación funcional o programación orientada a objetos.
- Diseñar una interfaz gráfica intuitiva que facilite la interacción con la herramienta, haciendo más accesibles sus funcionalidades para usuarios no técnicos.

Educación y Ética

Utilizar los hallazgos de esta investigación para fomentar prácticas éticas en la programación. Colaboración con instituciones educativas para integrar estas herramientas y contenidos en los planes de estudio de cursos de programación y ética profesional.

Monitoreo y Actualización de la Herramienta

- Establecer un ciclo continuo de evaluación y actualización para mantener la relevancia y precisión de la herramienta. Esto incluiría:
    - Recolectar datos de uso reales para identificar patrones comunes de similitud y áreas de mejora.
    - Mantener una base de datos de gramáticas de lenguajes actualizada, incluyendo soporte para nuevos lenguajes o versiones.

Contribución Abierta

- Publicar los resultados y herramientas desarrolladas bajo licencias de código abierto, permitiendo a la comunidad académica e industrial contribuir a su mejora y expansión.
- Crear un repositorio público para compartir datasets, modelos y resultados de evaluación, fomentando investigaciones futuras en el campo.

Bibliografía

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. (Citado en la página 5).

[2] Alex Aiken. Moss (measure of software similarity). *University of California, Berkeley*, 1994. `https://theory.stanford.edu/~aiken/moss/`. (Citado en la página 4).

[3] Uri Alon, Meital Brody, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[4] Uri Alon, Matan Zilberstein, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations (ICLR)*, 2019.

[5] Robert S Boyer and J Strother Moore. A fast string searching algorithm. In *Proceedings of the 19th Annual Symposium on Switching and Automata Theory (swat 1978)*, pages 62–72. IEEE, 1977. (Citado en la página 4).

[6] M. Duracik, E. Kirsak, and P. Hrkut. Source code representations for plagiarism detection. In *Springer International Publishing AG, part of Springer Nature: CCIS 870*, pages 61–69, 2018.

[7] Aminu B. Muhammad Abba Almu Hadiza Lawal Abba, Abubakar Roko and Abdulgafar Usman. Enhanced semantic similarity detection of program code using siamese neural network. *Int. J. Advanced Networking and Applications*, 14(2):5353–5360, 2022.

[8] Timothy Kam. Analyzing and understanding software artifacts with abstract syntax trees. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, 2005.

[9] Donald E Knuth, James H Morris Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977. (Citado en la página 4).

[10] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 413–422. IEEE, 2008.

[11] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013. (Citado en la página 39).

[12] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[14] D. Zou, W. Long, and Z. Ling. A cluster-based plagiarism detection method - lab report for pan at clef. In *Proceedings of the 4th Workshop on Uncovering Plagiarism, Authorship, and Social Software Misuse*, 2010.