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

Dedicación

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 issue of detecting similarities in source code, specifically in C projects. The process begins with the extraction of the Abstract Syntax Tree (AST) from various first-year programming projects, allowing for a structural representation of the code. From these ASTs, relevant features are extracted that describe the elements of the code. These features are modified to create variations, which are then grouped in pairs for analysis using machine learning techniques. Tests were conducted with different clustering algorithms and neural networks. Ultimately, techniques such as One-Class SVM and Isolation Forest were chosen, which demonstrated greater effectiveness in identifying similarities and anomalies in the code projects. This approach allows for a more precise and robust evaluation in our problem, providing a useful tool for plagiarism detection.

Índice general

Introducción

El análisis de similitud de código es un campo de gran escala en Ciencias de la Computación, especialmente en áreas como la detección de plagio, la revisión de código y la asistencia en programación. Este análisis permite comparar fragmentos de código para identificar similitudes y diferencias, proporcionando información valiosa para la refactorización, la mejora de la calidad del código y la promoción de buenas prácticas de programación.

Motivación

La evolución y expansión del software en la actualidad han incrementado notablemente su complejidad y volumen, creando una demanda urgente por herramientas avanzadas que permitan un análisis y comprensión más profundos del código fuente. En este contexto, la métrica de similitud de código se vuelve crucial, ya que puede identificar duplicaciones, detectar plagio y reconocer patrones reutilizables. Esta capacidad no solo ayuda a mantener la integridad del software al prevenir el plagio, sino que también fomenta prácticas de desarrollo más eficientes y sostenibles al identificar oportunidades para la refactorización y la reutilización de código.

La detección de plagio es especialmente relevante en el ámbito académico y profesional, donde la originalidad del trabajo es fundamental. Las herramientas avanzadas de análisis de similitud permiten identificar fragmentos de código que han sido copiados, modificados mínimamente o reutilizados sin atribución adecuada. Esto es esencial para mantener la equidad en la evaluación académica y proteger la propiedad intelectual en el desarrollo de software.

Con el avance continuo de las técnicas de aprendizaje automático y el procesamiento del lenguaje natural, se han abierto nuevas posibilidades para llevar a cabo este análisis de manera más eficiente y precisa. Los algoritmos de machine learning pueden aprender a identificar similitudes y diferencias en el código con una precisión antes inalcanzable, mientras que las técnicas de procesamiento del lenguaje natural permiten una comprensión más profunda y contextual del contenido del código. Estas innovaciones no solo mejoran la capacidad de detectar similitudes y plagio, sino que también pueden ofrecer comprensiones profundas sobre la estructura y el diseño del software, facilitando una mejora continua y sistemática del desarrollo de software.

Problemática

A pesar de los avances en el análisis de similitud de código, aún existen desafíos significativos. Las técnicas tradicionales, basadas en comparaciones textuales simples, son limitadas en su capacidad para capturar la estructura y semántica del código. Además, los enfoques más avanzados, como los basados en árboles de sintaxis abstracta (AST) y modelos de aprendizaje profundo, pueden ser computacionalmente costosos y difíciles de implementar a gran escala. En este proyecto se encontraron diferentes inconvenientes como la falta de datos y la falta de datos etiquetados, pues se utiliza un dataset creado con proyectos de programación de 1er año de la facultad,ademas la gramática utilizada por ANTLR no es la mas reciente, por tanto en algunos casos de proyectos donde se utiliza una sintaxis de C# moderna existe problemas para generar el AST completo, como solución a esto se hicieron unas ligeras modificaciones a la gramática de C#.

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. **Implementar y Evaluar Algoritmos de Comparación Basados en AST**
El primer objetivo es diseñar e implementar algoritmos de comparación que utilicen Árboles de Sintaxis Abstracta (AST) para identificar similitudes en el código. Este proceso implica desarrollar un método detallado para extraer subárboles de los AST y compararlos, capturando similitudes estructurales en diferentes niveles de granularidad. La efectividad de estos algoritmos se evaluará en términos de precisión y tiempo de procesamiento, comparándolos con los métodos tradicionales de similitud de código. Esto permitirá determinar si los enfoques basados en AST ofrecen ventajas significativas en el análisis de similitud de código.

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 modelos de aprendizaje supervisado y no supervisado utilizando un conjunto de datos conformado por proyectos de C# de orimer año la facultad de Ciencias de la Computación. Estos modelos deberán capturar patrones complejos y relaciones estructurales en el código, proporcionando una visión más profunda y 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

Esto tengo que mejorarlo o cambiarlo todo El análisis de similitud de código es un campo que ha evolucionado significativamente desde sus inicios en la década de 1970. La detección de similitudes en el código fuente es crucial en diversas aplicaciones, como la detección de plagio, la refactorización de código, la revisión de código y el desarrollo de herramientas de asistencia a la programación. Este estado del arte presenta una revisión exhaustiva de los desarrollos históricos, las metodologías y tecnologías utilizadas, así como los avances recientes en el análisis de similitud de código.

Los orígenes del análisis de similitud de código se remontan a la década de 1970, cuando las instituciones académicas comenzaron a buscar métodos para detectar plagio en tareas de programación. Este problema surgió debido al creciente número de cursos de programación y la necesidad de evaluar de manera justa las habilidades de los estudiantes. Durante este período, se desarrollaron algoritmos fundamentales de coincidencia de cadenas que sentaron las bases para comparar secuencias de texto, incluido el código.

Uno de los algoritmos más influyentes desarrollados durante este tiempo fue el algoritmo de Knuth-Morris-Pratt (KMP), presentado en 1970. El algoritmo KMP permite buscar patrones dentro de una cadena de texto de manera eficiente, evitando la necesidad de retroceder en el texto durante la búsqueda. Aunque originalmente diseñado para procesamiento de texto, su aplicabilidad para la comparación de secuencias de código fue reconocida rápidamente [9]. Los algoritmos de coincidencia de cadenas como KMP, junto con otros como el algoritmo de Boyer-Moore [5], proporciona-

ron herramientas básicas pero efectivas para la detección de similitudes en el código.

En la década de 1990, el análisis de similitud de códigos avanzó significativamente con el desarrollo de herramientas especializadas para la detección de plagio. Una de las herramientas más influyentes fue MOSS (Measure of Software Similarity), creada por Alex Aiken en la Universidad de Stanford. MOSS se diseñó específicamente para identificar similitudes estructurales y sintácticas en programas, superando las limitaciones de las comparaciones textuales simples [2].

MOSS funciona mediante la normalización del código antes de la comparación, eliminando comentarios y espacios en blanco, y renombrando variables a nombres genéricos. Este enfoque permite a MOSS detectar similitudes en la estructura y la lógica subyacente del código, incluso cuando los cambios superficiales han sido realizados para ocultar el plagio. La capacidad de MOSS para comparar la estructura y sintaxis del código en lugar de solo el texto hizo que fuera una herramienta pionera en el ámbito de la similitud de código.

La adopción de MOSS y herramientas similares tuvo un impacto significativo en el ámbito académico. Estas herramientas permitieron a los educadores detectar plagio en grandes conjuntos de tareas de programación de manera eficiente, manteniendo la integridad académica. La capacidad de identificar similitudes estructurales en el código contribuyó a la equidad en la evaluación de los estudiantes y a la promoción de prácticas de programación éticas.

Los árboles de sintaxis abstracta (AST) surgieron en la década de 1980, principalmente en el contexto del diseño de compiladores. Los AST representan la estructura jerárquica de la sintaxis del código fuente, permitiendo un análisis más profundo y significativo del código más allá de la simple comparación de texto [1].

Inicialmente, los AST fueron utilizados en los compiladores para representar la estructura sintáctica del código fuente de manera que fuera fácil de analizar y manipular. Un AST descompone el código en sus componentes sintácticos fundamentales, organizados jerárquicamente. Esta representación facilita la detección de errores de sintaxis y la optimización del código durante el proceso de compilación.

En la década de 2000, los AST comenzaron a ser utilizados en diversas

herramientas para el análisis, la transformación y la detección de similitudes de código [12]. Este cambio permitió comparaciones de código más sofisticadas y significativas, ya que los AST encapsulan la estructura sintáctica de los programas. En lugar de comparar solo el texto del código, el análisis basado en AST permite detectar similitudes en la estructura lógica y sintáctica del código, proporcionando una visión más detallada y precisa.

ANTLR (Another Tool for Language Recognition) es herramienta que se utiliza para generar analizadores léxicos y sintácticos a partir de gramáticas definidas por el usuario. Es utilizada en la compilación y el procesamiento de lenguajes, ya que permite transformar el código fuente en estructuras de datos que pueden ser manipuladas.??

I. Baxter, A. Yahin, L. Moura, M. Anna, y L. Bier. Presentaron métodos simples y prácticos para detectar clones exactos y casi coincidentes utilizando AST en el código fuente de programas. El método se basaba en hashing, primero se analizaba el código fuente para producir un AST, y luego se aplicaban tres algoritmos principales para encontrar clones. El primer algoritmo (algoritmo básico) detectaba clones de subárboles, el segundo (algoritmo de detección de secuencia) detectaba secuencias de tamaño variable de clones de subárboles, y el tercer algoritmo (último algoritmo) buscaba clones casi coincidentes más complejos intentando generalizar combinaciones de otros clones. Para que los algoritmos encuentren clones de subárboles funcionen, cada subárbol se comparaba con todos los demás subárboles para igualdad. El método es sencillo de implementar, pero los algoritmos realizan mejor trabajo en un gráfico de flujo de datos que en árboles, lo cual podría resultar en un alto número de falsos positivos al detectar clones casi coincidentes.

B. N. Pellin presentó una técnica para detectar autoría de código fuente. Utilizó técnicas de aprendizaje automático para transformar el código fuente en árboles de sintaxis abstracta y luego dividir el árbol en funciones. El árbol de cada función se consideraba un documento, con un autor dado. Esta colección se alimentaba a un paquete SVM usando un kernel que opera en datos estructurados en árbol. El clasificador se entrenó con código fuente de dos autores y luego era capaz de predecir cuál de los dos autores escribió una nueva función. El método logró alcanzar entre el 67% y el 88% de precisión de clasificación en el conjunto de programas exami-

nados. Sin embargo, el método es altamente vulnerable a la manipulación del código fuente, un traductor o ofuscador avanzado de código fuente podría destruir los patrones que su clasificador usa para identificar autores, y también requiere saber el conjunto de posibles autores.

J. Son, S. Park, y S. Park 6 propusieron un sistema de detección de plagio que utiliza núcleos de árboles de análisis. El papel de los núcleos de árboles de análisis en el sistema es manejar la información estructural dentro de los programas fuente y medir la similitud entre los árboles de análisis extraídos. El sistema realiza 100 % de precisión para un ataque simple y no se ve afectado por ataques estructurales. El sistema es independiente de los lenguajes de programación. Debido a la estructura de los programas copiados, que incluye mucho basura abundante, muchos sistemas de detección de plagio fallan en detectar plagio. Sin embargo, el sistema está expuesto a presentar resultados falsos positivos porque los valores de similitud de los núcleos de árboles aumentan demasiado rápido para manejarlos, y el valor de los núcleos entre dos árboles diferentes es típicamente mucho menor que el valor entre los mismos árboles. [7]

A principios de la década de 2000, la investigación se centró en extraer características de los AST para perfeccionar el análisis de similitud de código. Se desarrollaron técnicas para comparar subárboles de AST, lo que permitió detectar similitudes en diferentes niveles de granularidad de la estructura del código.

Comparar subárboles de AST permitió a los investigadores y desarrolladores detectar similitudes no solo en el nivel superficial del código, sino también en patrones más profundos y complejos [8]. Esta técnica facilitó la identificación de fragmentos de código que compartían estructuras similares, aunque no fueran idénticos textualmente.

M. Duracik, E. Kirsak, y P. Hrkut. [6] Desarrollaron un sistema que se enfoca en representar el código fuente utilizando AST para detectar plagio. El sistema representa el código fuente utilizando hashing y vectores de características. Realizaron un experimento basado en estos dos enfoques y trataron de calcular la similitud de clases así como de métodos en un conjunto de datos de código fuente; el cual consiste en 59 envíos de estudiantes. Intentaron minimizar la comparación absoluta de similitud (abordando la debilidad del algoritmo MOSS) pero su sistema no pudo

lograr escalabilidad y también se generaron algunas coincidencias falsas positivas a valores más bajos. Sin embargo, el sistema fue capaz de probar que generar vectores utilizando AST es la forma más adecuada de representar el código fuente.

Word2Vec es una técnica de aprendizaje profundo que transforma palabras en vectores de números de alta dimensión, conocidos como embeddings. Los modelos Word2Vec se entrenan utilizando grandes corpus de texto y capturan relaciones semánticas entre las palabras.??????????????????????????????????????

La integración del aprendizaje automático en el análisis de similitud de códigos comenzó en la década de 2010. Los algoritmos de aprendizaje automático mejoraron la precisión de la detección de similitudes mediante el aprendizaje de patrones a partir de grandes conjuntos de datos de código.

El uso de redes neuronales comenzó a expandirse en este período. Las redes neuronales recurrentes (RNN) y las redes neuronales convolucionales (CNN) fueron entrenadas para procesar y comparar fragmentos de código, generando representaciones vectoriales que capturaban relaciones complejas y patrones estructurales. Estas técnicas permitieron una comparación más precisa y eficiente de fragmentos de código, superando las limitaciones de los métodos tradicionales.

Alrededor de 2014, los marcos de aprendizaje profundo comenzaron a aplicarse al análisis de código. Las redes neuronales profundas (DNN) se entrenaron para generar incrustaciones vectoriales de fragmentos de código, permitiendo una comparación más precisa y eficiente de similitudes. Estas representaciones vectoriales capturaban relaciones complejas y patrones estructurales, mejorando significativamente la precisión de la detección de similitudes.

Un enfoque destacado en este contexto son las redes neuronales siamesas, que se utilizan para aprender representaciones semánticas similares entre pares de fragmentos de código [7]. Estas redes utilizan dos ramas idénticas con pesos compartidos para procesar dos entradas de código diferentes simultáneamente. Al final de cada rama, se obtienen representaciones vectoriales que luego se comparan directamente o a través de medidas de distancia como la distancia euclidiana o la similitud del coseno. Este enfoque permite capturar relaciones más sutiles y contextuales entre

fragmentos de código, superando las limitaciones de los métodos tradicionales basados en características simples.

Aqui hablar sobre las redes neuronales siamesas

En la década de 2020, los modelos avanzados de aprendizaje profundo, en particular los transformadores, se han aplicado cada vez más al análisis de código. Estos modelos han demostrado ser altamente efectivos para capturar las complejidades del código y mejorar las capacidades de detección de similitudes.

Ejemplos notables incluyen Codex de OpenAI y GitHub Copilot, que utilizan grandes cantidades de datos de código para tareas como completar código y detectar similitudes. Los transformadores son capaces de manejar secuencias largas y capturar dependencias a largo plazo en el código, lo que los hace especialmente adecuados para el análisis de similitud de código [13].

Las redes neuronales gráficas (GNN) también han ganado prominencia en el análisis de similitud de código. Las GNN tratan los AST como gráfos, capturando estructuras de código intrincadas para un análisis de similitud más matizado. Este enfoque permite una representación más rica y detallada del código, mejorando la precisión de las comparaciones.

Capítulo 2

Propuesta de solución

La implementacion de este trabajo esta dividida en tres partes fundamentales: Extraccion del AST, trabajo con los datos y explicación del modelo.

- Extraccion del AST: En esta parte se utiliza la herramienta ANTLR de C# para crear el ast de los proyectos. Luego de este ast se extraen los features para su posterior uso.

- Trabajo con los datos: Los conjuntos de features de cada proyecto se convierten en vectores de características.

- Modelo utilizado: Se implementa el modelo de Red Neuronal Siamesa para determinar la semejanza entre 2 proyectos.

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. Fue necesario hacer ligeras modificaciones sobre la gramática, pues ANTLR cuenta con la gramatica 8.0 de C#.
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 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 denominada **FeatureExtractorListener**, que extiende la funcionalidad de ANTLR 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 luego 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

bla bla bla

4.2. 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.

4.3. 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 ins-

[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.

tancias 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.

Capítulo 5

Redes Neuronales Siamesas para la Similitud de Código

Las redes neuronales siamesas han demostrado ser efectivas en el análisis de similitud de código, debido a su estructura que permite aprender relaciones semánticas profundas entre pares de datos de entrada. En este capítulo se describen los componentes fundamentales de esta arquitectura y su proceso de entrenamiento.

5.1. Estructura de una Red Neuronal Siamesa

La red neuronal siamesa consta de dos subredes idénticas que comparten los mismos parámetros y pesos. Cada subred toma como entrada un fragmento de código, que es representado a través de vectores de características derivados de su Árbol de Sintaxis Abstracta (AST). Las salidas de ambas subredes son vectores de alta dimensión que capturan características semánticas y estructurales del código procesado. Posteriormente, estos vectores se comparan mediante una función de distancia, la similitud de coseno, que cuantifica el grado de similitud entre los fragmentos de código.

Esta red esta compuesta por tres capas densas.

Al emplear una red siamesa, el modelo puede determinar qué tan "cercanos" o "distantes" son dos fragmentos de código en el espacio semánti-

co aprendido, lo cual es crucial en aplicaciones donde se busca identificar equivalencias estructurales o funcionales.

5.2. Proceso de Entrenamiento y Función de Pérdida

El entrenamiento de una red neuronal siamesa para la detección de similitud de código generalmente se lleva a cabo mediante la *pérdida de contraste* (contrastive loss) o la *pérdida de triplete* (triplet loss). Estas funciones de pérdida están diseñadas para reducir la distancia entre los vectores de fragmentos de código que cumplen la misma funcionalidad (pares similares) y aumentar la distancia entre los vectores de fragmentos de código que cumplen funciones diferentes (pares disimilares), en este trabajo se utilizará la tecnica de *pérdida de contraste*. Esta función minimiza la distancia entre representaciones de fragmentos de código similares y maximiza la distancia entre representaciones de fragmentos disímiles. Para cada par de fragmentos de código, esta función penaliza la red de acuerdo con la cercanía o lejanía de las representaciones generadas.

5.3. Aplicación de Redes Siamesas en la Similitud de Código en C#

Para aplicar redes siamesas en la detección de similitud de código en C#, cada fragmento de código se transforma en un vector de características utilizando su AST. Este árbol captura la estructura sintáctica y lógica del código, permitiendo que el modelo aprenda relaciones más profundas que las posibles mediante representaciones textuales o léxicas. Los vectores de características pueden incluir información sobre las operaciones, los flujos de control, los nombres de funciones, y otros elementos sintácticos relevantes del código en C#.

La representación vectorial del AST es crítica, ya que permite que la red siamés se enfoque en aspectos estructurales y funcionales del código, ignorando variaciones superficiales, como cambios en el nombre de variables o formato. De esta forma, el modelo se vuelve robusto frente a

modificaciones superficiales que no alteran la lógica del programa, pero susceptibles de engañar métodos más básicos de detección de similitud.

5.4. Futuras Direcciones

Se me ocurre la idea de entrenar el modelo con codigo de java(existe mas dataset con java que C#) para que vaya teniendo una base de que es una copia de codigo y luego especializarlo con codigos de C#, porque pienso que esto funcionaria? Los vectores vas a tener la misma estructura tanto para C como para Java (cantidad de if, de while etc.) . Aqui tendria que investigar las diferencias entre C# y Java, generar el AST con ANTLR para Java hacer de nuevo la extraccion de caracteristicas si los ast no tienen la misma estructura que es lo mas probable, muy pesado de hacer... y ya despues de esto tendria el vector de caracteristicas para Java y a entrenar el modelo con java, luego investigar como se reutiliza un modelo ya entrenado y especializarlo, esto tiene el nombre en ML como fine tuning. Tambien puede que aparezca magicamente el modelo ya entrenado con Java y sea cargarlo. Segun ChatGPT:

Este método implica dos fases principales: Pre-entrenamiento (Pre-training): En esta fase, el modelo se entrena con un conjunto de datos grande y general. Este entrenamiento inicial permite al modelo aprender características generales y representaciones útiles de los datos. Fine-tuning (Ajuste fino): Después del pre-entrenamiento, el modelo se vuelve a entrenar con un conjunto de datos más pequeño y específico para la tarea en cuestión. Durante este proceso, se ajustan los pesos del modelo para adaptarse mejor a la tarea específica. Características clave del transfer learning: Eficiencia: Permite aprovechar el conocimiento adquirido en tareas generales para mejorar el rendimiento en tareas específicas. Menor necesidad de datos: Requiere menos datos para la tarea específica, ya que el modelo ya ha aprendido características generales útiles. Tiempo de entrenamiento reducido: El fine-tuning suele ser más rápido que entrenar un modelo desde cero. Mejor rendimiento: A menudo resulta en un mejor rendimiento en la tarea específica, especialmente cuando los datos específicos son limitados. Versatilidad: Se puede aplicar en diversos campos como visión por computadora, procesamiento de lenguaje natural, y reconocimiento de voz. En el contexto de tu proyecto de similitud de código, podrías considerar pre-entrenar tu modelo siamés en un conjunto grande de proyectos de

código de diversos lenguajes, y luego hacer fine-tuning con un conjunto más pequeño de proyectos C específicos. Esto podría ayudar a tu modelo a capturar características generales de la estructura del código y luego especializarse en las particularidades del C.

Despues de esta maravillosa idea que no se si funcione esta seguir probando con chatGPT a ver si me quiere dar codigos plagueados bien, entrenar a algun LLM hacerle prompt ingeeniering para ensennarle a plaguear y que luego nos de el codigo que queremos.

No se me ocurre mas nada para el dataset.

Capítulo 6

Detalles de Implementación y Experimentos

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

Tabla 6.1 Resultados de Clustering

Método	**Silhouette**	**Calinski Harabaz**	**Davies Bouldin**
Agglomerative Clustering	0.2236	186.1952	1.4984
DBSCAN	0.4641	25.136239627069884	1.6503
K-Mean	-	-	
Mean-Shift	-	-	

Conclusiones

En esta investigación, se ha abordado la problemática de la detección de similitud y plagio en proyectos de C# mediante técnicas avanzadas de análisis y aprendizaje automático.

La extracción de árboles de sintaxis abstracta (AST) de diferentes proyectos de C# ha demostrado ser una técnica efectiva para capturar la estructura sintáctica y lógica de los códigos. Este enfoque ha permitido obtener características detalladas y representativas del código fuente, facilitando un análisis más profundo y preciso.

Al agrupar los proyectos en pares y calcular las diferencias entre sus vectores de características, se ha logrado incrementar significativamente la cantidad de datos disponibles para el entrenamiento de modelos de aprendizaje automático. Esta estrategia no solo ha aumentado la diversidad y riqueza del dataset, sino que también ha permitido capturar relaciones y distancias específicas entre los proyectos, mejorando así la precisión en la detección de similitudes.

Durante el proceso de experimentación, se evaluaron varios algoritmos de clustering, como K-Means, DBSCAN y Agglomerative Clustering. Sin embargo, estos métodos no resultaron efectivos para la tarea específica de detección de similitud de proyectos de C#.

Ante las limitaciones de los métodos de clustering, se decidió utilizar algoritmos de detección de anomalías como One-Class SVM e Isolation Forest. Estos métodos han demostrado ser más adecuados para describir la región de los vectores de similitud de proyectos distintos, permitiendo una identificación más precisa de proyectos similares y diferentes.

La similitud de código se ha confirmado como una métrica crucial para detectar duplicaciones, plagio y patrones reutilizables en el software. Las herramientas y técnicas desarrolladas en esta tesis contribuyen significativamente a la mejora continua del software, facilitando la identificación de áreas de optimización y refactorización.

La integración de técnicas de aprendizaje automático, como el uso modelos de detección de anomalías, ha permitido abordar el problema de la similitud de código de manera más efectiva y precisa. Estas técnicas han abierto nuevas posibilidades para el análisis de código.

En resumen, esta tesis ha demostrado que la combinación de técnicas de análisis sintáctico, generación de datasets enriquecidos y algoritmos de aprendizaje automático puede abordar de manera efectiva la detección de similitud y plagio en proyectos de C#.

Recomendaciones

A futuro, se pueden explorar modelos avanzados de aprendizaje profundo, como transformadores, redes neuronales siamesas(SNN) y redes neuronales gráficas (GNN), para mejorar aún más la precisión y eficiencia en la detección de similitud de código. Además, la generación de datasets más amplios y diversificados, así como la incorporación de técnicas de procesamiento de lenguaje natural, pueden ofrecer nuevas oportunidades para el análisis y comprensión del código fuente.

Bibliografía

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. (Citado en la página 6).

[2] Alex Aiken. Moss (measure of software similarity). *University of California, Berkeley*, 1994. `https://theory.stanford.edu/~aiken/moss/`. (Citado en la página 6).

[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 5).

[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. (Citado en la página 8).

[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. (Citado en las páginas 8 y 9).

[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. (Citado en la página 8).

[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 5).

[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 20).

[12] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. (Citado en la página 7).

[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. (Citado en la página 10).

[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.