Universidad de La Habana
Facultad de Matemática y Computación

Code Similarity

Autor:

María de Lourdes Choy Fernández
Alejandro Yero Valdéz

Tutores:

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

julio de 2022

github.com/Chonyyy/Code_Similarity

Resumen

Resumen en español

Abstract

Resumen en inglés

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.

0.1. Motivación

La creciente complejidad y volumen del software moderno ha generado la necesidad de herramientas más sofisticadas para analizar y comprender el código. La similitud de código es una métrica importante que puede ayudar a detectar duplicaciones, plagio y patrones reutilizables, facilitando la mejora continua del software. Además, con el avance de las técnicas de aprendizaje automático y el procesamiento del lenguaje natural, se han abierto nuevas posibilidades para realizar este análisis de manera más efectiva y precisa.

0.2. 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. La precisión y la eficiencia siguen siendo áreas críticas de mejora, especialmente en contextos donde el código es altamente variable y complejo.

0.3. 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, con el fin de 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.

0.4. 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 comparación 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 la facultad. 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

Estado del Arte

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.

Primeros comienzos: detección de plagio de código (décadas de 1970 a 1990)

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 1976. 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. Los algoritmos de coincidencia de cadenas como KMP, junto con otros como el algoritmo de Boyer-Moore, proporcio-

naron 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.

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.

Impacto y Adopción 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.

La aparición de los árboles de sintaxis abstracta (AST) (décadas de 1980 a 2000)

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.

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. 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.

Extracción de funciones y similitud de código (décadas de 2000 a 2010)

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. Esta técnica facilitó la identificación de fragmentos de código que compartían estructuras similares, aunque no fueran idénticos textualmente.

Integración del aprendizaje automático (década de 2010 al presente)

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.

Los primeros enfoques de aprendizaje automático para la similitud de código utilizaban técnicas de aprendizaje supervisado, entrenando modelos en conjuntos de datos etiquetados para aprender patrones de similitud. Estos modelos eran capaces de generalizar más allá de las comparaciones textuales y estructurales simples, identificando similitudes más profundas en la lógica y el diseño del código. Además de los métodos supervisados, los algoritmos de aprendizaje no supervisado, como el clustering, comenzaron a ganar tracción en este campo.

Los algoritmos de clustering, como K-means y DBSCAN, se utilizaron para agrupar fragmentos de código similares sin necesidad de etiquetas previas. Estos métodos

permitieron descubrir estructuras y patrones emergentes en grandes conjuntos de datos de código, facilitando la detección de similitudes y la identificación de funciones comunes entre diferentes proyectos.

El uso de redes neuronales también 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. 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.

Modelos como code2vec y code2seq

En 2018, se introdujeron modelos como code2vec y code2seq, que aprovechan los AST para transformar el código en representaciones vectoriales adecuadas para diversas aplicaciones, incluida la detección de similitudes. Estos modelos utilizan técnicas de procesamiento de lenguaje natural adaptadas al análisis de código, permitiendo una comprensión más profunda de las relaciones entre los componentes del código. Code2vec y code2seq, por ejemplo, generan representaciones vectoriales a partir de caminos entre nodos en los AST, capturando información sintáctica y semántica del código.

Tendencias e innovaciones actuales (década de 2020 al presente)

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.

Las redes neuronales gráficas (GNN) también han ganado prominencia en el análisis de similitud de código. Las GNN tratan los AST como gráficos, 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

La implementacion de este trabajo esta dividida en dos partes fundamentales: Extraccion del AST y trabajo con machine learning.

- 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 machine learning: Se utiliza el modelo ———- para determinar la semejanza entre 2 proyectos.

2.1. Extraccion del AST

ANTLR (Another Tool for Language Recognition) es una poderosa herramienta que se utiliza para generar analizadores léxicos y sintácticos a partir de gramáticas definidas por el usuario. Es ampliamente utilizada en la compilación y el procesamiento de lenguajes, ya que permite transformar el código fuente en estructuras de datos que pueden ser fácilmente manipuladas. En este caso, ANTLR se emplea para extraer Árboles de Sintaxis Abstracta (AST) a partir del código fuente de programas escritos en C.

Una vez que se ha generado el AST, se puede manipular y analizar utilizando las estructuras de datos proporcionadas por ANTLR. Por ejemplo, se pueden recorrer los nodos del árbol, extraer información específica o transformar el AST para diferentes propósitos, en este caso se utilizó el listener proporcionado por ANTLR para recorrer el árbol y extraer los featurues.

2.2. Extraccion de features

En el análisis de similitud de código y detección de patrones, es crucial extraer características relevantes que capturen la estructura y el comportamiento del código. 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 característica extraída.

Características Extraídas y su Importancia

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**$_{t}uples : N$úm$erodevariablesdetipotupla.$**lists** :N$\acute{u}merodelistasdeclaradas.$
- **dicts:** Número de diccionarios declarados.

Las variables y las constantes son fundamentales para entender el estado y el flujo de datos en el código. La variedad y el tipo de estructuras de datos utilizadas (tuplas, listas, diccionarios) también proporcionan información sobre el estilo de programación y la complejidad del código.

III. Declaraciones de Métodos y Clases:

- **methods:** Número de métodos declarados.
- **method**$_{n}ames : Conjuntodenombresdem\acute{e}todos.$**method_return**$_{\mathbf{t}}$**ypes** :$Conjuntodetip$
- **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 proporcionan información sobre la organización y modularidad del código. Los métodos y sus parámetros son esenciales para entender la funcionalidad del código, mientras que las clases y sus tipos (abstractas, selladas) indican la arquitectura de la aplicación.

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**$_b locks : Número de bloques try - catch$. 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:

- **acces**$_m odifiers\_public : Número de elementos públicos$.**access_modifiers_private** :Nú
- **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 indican características avanzadas y patrones de diseño en el código, como la concurrencia (async), la seguridad (unsafe) y la herencia (override, virtual).

VII. Llamadas a Librerías y LINQ:

- **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 consultas LINQ proporcionan información sobre el uso de funcionalidades estándar y el manejo de colecciones de datos en el código.

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 proporcionan una visión más completa de las capacidades del código, su organización y las prácticas de programación utilizadas.

La extracción de características con FeatureExtractorListener permite capturar una amplia gama de aspectos del código fuente en C, desde su estructura y complejidad hasta los patrones de diseño y las prácticas de programación. Esta información es crucial para tareas como la detección de similitudes de código, la evaluación de la calidad del código y la identificación de posibles plagios. La implementación y el análisis detallado de estas características proporcionan una base sólida para mejorar la precisión y efectividad de las herramientas de análisis de código.

2.3. Preparacion del dataset

Capítulo 3

Detalles de Implementación y Experimentos

Conclusiones

Conclusiones

Recomendaciones

Recomendaciones