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

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 diversos proyectos, permitiendo una representación estructural del código. A partir de estos AST, se extraen características relevantes que describen los elementos del código. Estas características son modificadas para crear variaciones y, posteriormente, se agrupan en pares para su análisis mediante técnicas de aprendizaje automático. Se realizaron pruebas con distintos algoritmos de clustering y redes neuronales, pero los resultados no fueron satisfactorios debido a la falta de datos. Finalmente, se optó por utilizar técnicas como One-Class SVM e Isolation Forest, las cuales demostraron ser más eficaces en la identificación de similitudes y anomalías en los proyectos de código. Este enfoque permite una evaluación más precisa y robusta en nuestro problema, proporcionando una herramienta útil para la detección de plagio.

Abstract

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 various 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. Experiments were conducted with different clustering algorithms and neural networks, but the results were unsatisfactory due to the lack of data. Ultimately, techniques such as One-Class SVM and Isolation Forest were chosen, which proved to be more effective in identifying similarities and anomalies in the code projects. This approach allows for a more accurate 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.

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 Knuth y col. 1977. Los algoritmos de coincidencia de cadenas como KMP, junto con otros como el algoritmo de Boyer-Moore Boyer y Moore 1977, proporcionaron 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 Aiken 1994.

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 Aho y col. 1986.

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 Muchnick 1997. 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.

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 examinados. 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. Hadiza Lawal Abba y Usman 2022

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 Kam 2005. 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. Duracik y col. 2018 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.

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.

D. Zou, W. Long, y Z. Ling. Zou y col. 2010 describieron un método de detección de plagio basado en clustering para detectar plagio en cursos relacionados con ingeniería de redes. El método consta de 3 pasos: el primer paso es el paso de selección previa, que es encontrar una pequeña lista de documentos candidatos desde el conjunto de documentos fuente que pueden contener contenido plagiado. El siguiente paso es comparar el documento sospechoso con cada documento candidato para obtener la parte copiada del documento sospechoso, este paso se llama localización. El último paso se llama post-procesamiento, que es descartar algunos fragmentos sin plagio del resultado final. El método fue probado tanto en el conjunto de entrenamiento como en el conjunto de prueba para PAN-09 y demostró ser efectivo, pero debido a sus múltiples pasos operativos, consume mucho tiempo de ejecución. Por lo tanto, el método se dice tener una alta complejidad.

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 Hadiza Lawal Abba y Usman 2022. 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 Alon y col. 2019b y code2seq Alon y col. 2019a, 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 Vaswani y col. 2017.

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 de solución

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 estudian diferentes técnicas para determinar la semejanza entre 2 proyectos. El uso de modelos de deteccion de oulayers para agrupar los vectores de similitud de los proyectos que son distintos entre si.

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#. Fue necesario hacer ligeras modificaciones sobre la gramática, pues ANTLR tenia una gramatica desactualizada.

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.

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 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_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 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_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 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

Para preparar el dataset utilizado en el análisis de similitud de código, se requirió convertir los nombres de variables, métodos y otros identificadores en vectores de características numéricas. Este proceso se llevó a cabo utilizando la técnica de embeddings con Word2Vec. A continuación, se explica qué es Word2Vec, cómo funciona y por qué se eligió para esta tarea Mikolov y col. 2013.

Uso de Word2Vec en la Preparación del Dataset

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.

En el contexto del análisis de similitud de código, los nombres de variables, métodos y otros identificadores en el código fuente juegan un papel crucial. Estos

nombres pueden proporcionar información semántica valiosa 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 similares. Sin embargo, los identificadores en el código no están estructurados de manera que las máquinas puedan comprender fácilmente sus relaciones semánticas. Para abordar este desafío, se utilizó Word2Vec para convertir estos identificadores en embeddings. 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.

Nuevo dataset

Para maximizar la cantidad de datos disponibles y reflejar de manera efectiva la similitud entre proyectos, se creó un nuevo 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 adicional 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, 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 más refinadas y precisas, ya que cada par de proyectos se compara de manera detallada.

En resumen, la creación de este nuevo 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.

2.4. Probando diferentes modelos

SNN

La primera solución abordada fue implementar una red neuronal siamesa para comparar dos proyectos. Estas redes funcionan con dos entradas, procesando cada una a través de la misma arquitectura de red para obtener representaciones vectoriales comparables. La red neuronal siamesa mide la distancia entre estos vectores para determinar la similitud entre los proyectos.

Para esta solución, se separaron todos los posibles pares de proyectos y se les asignó una etiqueta: 0 si los proyectos no son copias (ambos originales) y 1 si los proyectos son copias. Los datos se convirtieron en vectores de características y el conjunto de datos se constituyó con los pares de vectores correspondientes y sus etiquetas.

Sin embargo, surgió el problema de la generación de copias de proyectos para etiquetar correctamente los datos. Esto llevó a la necesidad de buscar otra solución más efectiva a la carencia de datos etiquetados.

Clustering

El clustering es un proceso de aprendizaje no supervisado, utilizado para agrupar datos sin conocer de antemano la etiqueta o categoría de cada dato. En lugar de depender de etiquetas predefinidas, los algoritmos de clustering analizan la proximidad y similitud entre los datos para formar grupos o clústeres basados en características compartidas. Cada clúster agrupa datos que son similares entre sí, pero que difieren de los datos en otros clústeres. Este enfoque es particularmente útil en situaciones donde se quiere descubrir patrones, estructuras o relaciones inherentes sin la necesidad de etiquetas predefinidas.

Se aplicaron distintos enfoques de clustering, entre ellos:

- **K-Means**: Agrupa datos en k clústeres basándose en la distancia euclidiana a los centroides, que se ajustan iterativamente. Aquí se hizo uso del método del codo para determinar el parámetro k a utilizar.
- **DBSCAN (Density-Based Spatial Clustering of Applications with Noise)**: Identifica clústeres basados en la densidad de puntos, permitiendo descubrir clústeres de forma arbitraria y detectar ruido.
- **Agglomerative Clustering**: Un enfoque jerárquico que une iterativamente los puntos o clústeres más cercanos, formando un dendrograma.
- **MeanShift**: Agrupa datos desplazando iterativamente los puntos hacia la densidad máxima de puntos más cercana, identificando modos en la densidad de datos.

Aunque algunos de estos algoritmos mostraron buenas métricas en ciertos aspectos, no lograron resolver completamente el problema planteado.

2.5. Modelos utilizados

One-Class SVM

One-Class Support Vector Machine (One-Class SVM) es una variante del algoritmo SVM (Support Vector Machine) diseñada para la detección de anomalías y para identificar datos que difieren del conjunto de datos de entrenamiento. One-Class SVM es particularmente útil en escenarios donde solo se dispone de datos de

una clase (la clase "normal") y se desea detectar cualquier dato que sea diferente o anómalo.

One-Class SVM se basa en la idea de encontrar una función que sea positiva en la región donde se encuentran la mayoría de los datos de entrenamiento y negativa en otras regiones. Este enfoque ayuda a identificar las regiones del espacio de características donde se encuentran los datos ñormalesÿ marcar como anómalos los datos que caen fuera de estas regiones.

Funcionamiento:

I. **Transformación del Espacio de Características:**

- Se utiliza una función de kernel para mapear los datos de entrada a un espacio de características de mayor dimensión.
- Los kernels comunes incluyen el kernel lineal, polinómico, radial (RBF) y sigmoide.

II. **Maximización del Margen:**

- One-Class SVM trata de encontrar un hiperplano en el espacio de características transformado que maximice la distancia de los puntos de datos a este hiperplano.
- La mayoría de los datos de entrenamiento deben estar en un lado del hiperplano (considerados normales) y los puntos fuera de este margen son considerados anomalías.

Fórmula Matemática

El problema de optimización para One-Class SVM se define como:

$$\min_{\mathbf{w},\rho,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{\nu n}\sum_{i=1}^{n}\xi_i - \rho$$

sujeto a:

$$(\mathbf{w}\cdot\phi(\mathbf{x}_i)) \geq \rho - \xi_i, \quad \xi_i \geq 0, \quad i = 1,\ldots,n$$

Donde:

- $\mathbf{w}$ es el vector de pesos.
- $\phi(\mathbf{x}_i)$ es la transformación del espacio de características mediante el kernel.

- ρ es el umbral que separa los datos normales de las anomalías.
- ξ_i son las variables de holgura que permiten que algunos puntos estén en el lado equivocado del margen.
- ν es un parámetro que controla el trade-off entre la fracción de puntos de datos que son permitidos estar en el lado equivocado del margen y la regularización del modelo.

Ventajas:

- Efectivo en escenarios donde solo se dispone de datos de una clase.
- Flexible con el uso de diferentes kernels para capturar la estructura de datos no lineales.

Isolation forest

Isolation Forest es un algoritmo de aprendizaje automático no supervisado diseñado para detectar anomalías en datos. Fue introducido por Fei Tony Liu, Kai Ming Ting y Zhi-Hua Zhou en 2008 **liu2008isolation** A diferencia de muchos algoritmos de detección de anomalías que se basan en la densidad o la distancia, Isolation Forest se basa en el concepto de äislamiento.

La idea central de Isolation Forest es que las anomalías son puntos de datos que son "fáciles de aislar". Dado que los valores anómalos son raros y diferentes del resto de los datos, deben requerir menos divisiones para ser aislados.

Funcionamiento:

I. **Construcción de Árboles de Aislamiento:**
   - Se construyen múltiples árboles de aislamiento (iTrees). Cada árbol se construye seleccionando aleatoriamente una característica y luego eligiendo aleatoriamente un valor de división entre los valores mínimo y máximo de esa característica.
   - Este proceso se repite hasta que cada punto de datos esté aislado en una hoja del árbol o hasta que se alcance una profundidad máxima predefinida.

II. **Cálculo de la Longitud del Camino:**
   - La longitud del camino de un punto de datos es el número de divisiones necesarias para aislar ese punto.

- Las anomalías, siendo más fáciles de aislar, tienden a tener caminos más cortos.

III. **Puntuación de Anomalía:**

- La puntuación de anomalía se basa en la longitud del camino.
- Si un punto tiene una puntuación de anomalía alta, es considerado una anomalía.

Fórmula Matemática

Para calcular la puntuación de anomalía, se utiliza la siguiente fórmula:

$$s(x,n) = 2^{-\frac{E(h(x))}{c(n)}}$$

Donde:

- $s(x,n)$ es la puntuación de anomalía del punto x en un conjunto de datos de tamaño n.
- $E(h(x))$ es la longitud media del camino de aislamiento para el punto x calculada sobre múltiples árboles.
- $c(n)$ es una constante de normalización calculada como:

$$c(n) = 2H(n-1) - \left(\frac{2(n-1)}{n}\right)$$

Aquí, $H(i)$ es la i-ésima armónica:

$$H(i) = \sum_{k=1}^{i} \frac{1}{k}$$

Ventajas:

- Eficiente y escalable a grandes conjuntos de datos.
- No requiere un modelo previo de datos normales o anómalos.
- Puede manejar datos de alta dimensionalidad.

Por que elegir estos modelos?

Ambos métodos, Isolation Forest y One-Class SVM, fueron entrenados con pares de proyectos que son diferentes entre sí. Esto significa que aprendieron las diferencias inherentes entre proyectos que podrían contener anomalías o fraudes. Al entrenar con proyectos que representan variabilidad y diversidad en términos de estructuras de código, estilos de programación y lógica implementada, los modelos pueden capturar mejor las características que distinguen un proyecto legítimo de uno potencialmente fraudulento.

Descripción de la Región de Similitud

El objetivo de detectar fraude entre proyectos de C# de primer año implica identificar anomalías en la similitud estructural y lógica entre dos proyectos. Isolation Forest y One-Class SVM son adecuados para este propósito porque:

- **Isolation Forest:** Al modelar la estructura de los datos mediante la creación de múltiples árboles de decisión, Isolation Forest identifica anomalías como puntos que son fáciles de aislar en el espacio de características. Esto es efectivo para detectar proyectos que se desvían significativamente de la norma en términos de estructura y contenido.
- **One-Class SVM:** Al encontrar una frontera que encapsula la mayoría de los datos normales en el espacio de características, One-Class SVM puede detectar puntos de datos que están muy lejos de esta frontera. Esto es útil para identificar proyectos que muestran características inusuales o no esperadas en comparación con los proyectos normales.

El dominio específico de proyectos de C# de primer año facilita el uso de estos algoritmos. Dado que los proyectos son de un nivel académico inicial, es probable que existan variaciones predecibles pero distintivas entre los proyectos legítimos y aquellos que podrían contener fraude o anomalías. Esto hace que describir la región de los vectores de similitud entre proyectos distintos sea más accesible y factible, ya que las diferencias entre proyectos normales y anómalos pueden ser más discernibles y significativas en este contexto específico.

Capítulo 3

Detalles de Implementación y Experimentos

Conclusiones

Conclusiones

Recomendaciones

Recomendaciones

Bibliografía

Aho, A. V., Sethi, R. y Ullman, J. D. (yearmonthday). *Compilers: principles, techniques, and tools*. Addison-Wesley.

Aiken, A. (yearmonthday). Moss (measure of software similarity). *University of California, Berkeley*. `https://theory.stanford.edu/~aiken/moss/`.

Alon, U., Zilberstein, M., Levy, O. y Yahav, E. (yearmonthday). Code2seq: generating sequences from structured representations of code. En *International conference on learning representations (iclr)*. doi:`10.1023/A:1020518621128`

Alon, U., Brody, M., Levy, O. y Yahav, E. (yearmonthday). Code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, *3*(POPL), 1-29. doi:`10.1145/3290353`

Boyer, R. S. y Moore, J. S. (yearmonthday). A fast string searching algorithm. En *Proceedings of the 19th annual symposium on switching and automata theory (swat 1978)* (págs. 62-72). IEEE.

Duracik, M., Kirsak, E. y Hrkut, P. (yearmonthday). Source code representations for plagiarism detection. En *Springer international publishing ag, part of springer nature: ccis 870* (págs. 61-69).

Hadiza Lawal Abba, A. B. M. A. A., Abubakar Roko y Usman, A. (yearmonthday). Enhanced semantic similarity detection of program code using siamese neural network. *Int. J. Advanced Networking and Applications*, *14*(2), 5353-5360.

Kam, T. (yearmonthday). Analyzing and understanding software artifacts with abstract syntax trees. En *Proceedings of the 2005 international conference on software engineering research and practice*.

Knuth, D. E., Morris Jr, J. H. y Pratt, V. R. (yearmonthday). Fast pattern matching in strings. *SIAM journal on computing*, *6*(2), 323-350.

Mikolov, T., Chen, K., Corrado, G. y Dean, J. (yearmonthday). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*. doi: `10.48550/arXiv.1301.3781`

Muchnick, S. S. (yearmonthday). *Advanced compiler design and implementation*. Morgan Kaufmann Publishers.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N. y Polosukhin, I. (yearmonthday). Attention is all you need. En *Advances in neu-*

ral information processing systems (neurips) (págs. 5998-6008). doi:10.48550/arXiv.1706.03762

Zou, D., Long, W. y Ling, Z. (yearmonthday). A cluster-based plagiarism detection method - lab report for pan at clef. En *Proceedings of the 4th workshop on uncovering plagiarism, authorship, and social software misuse*.