Detección de Similitud de Código Usando Embeddings Basadas en Transformers y Aprendizaje Automático

Instituto Tecnológico y de Estudios Superiores de Monterrey

Author: Diego Vega Camacho, *A01704492@tec.mx*

I N F O R M A C I Ó N

Palabras clave:

- Detección de similitud de código
- UnixCoder
- FLAML
- AutoML
- Embeddings semánticos
- Clasificación
- TF-IDF
- Similitud coseno

A B S T R A C T O

La detección de similitud de código desempeña un papel crucial en la ingeniería de software, integridad académica y protección de propiedad intelectual. Los métodos tradicionales a menudo tienen dificultades con variaciones sintácticas o modificaciones estructurales del código. Este artículo presenta un enfoque híbrido que combina embeddings generados mediante el modelo transformer pre-entrenado UnixCoder con similitud coseno, y un clasificador supervisado entrenado mediante AutoML (FLAML) para clasificar la similitud entre fragmentos de código. Utilizando conjuntos de datos derivados del corpus FIRE 2014 SOCO, se generaron pares de código funcionalmente equivalentes (positivos) y no equivalentes (negativos), equilibrando ambos tipos de ejemplos para entrenar y evaluar el modelo. Los resultados obtenidos mostraron un desempeño competitivo, alcanzando métricas significativas en términos de precisión, recall y F1-score tanto para Java como para C. Además, se comparó contra un enfoque tradicional basado en SVM y TF-IDF, mostrando mejoras significativas en desempeño tras una validación manual en ambos lenguajes. El método propuesto destaca por su bajo costo computacional y alta eficiencia, presentando una solución práctica y escalable para aplicaciones reales en la evaluación automática de similitud de código.

1. Introducción

Detectar similitud entre fragmentos de código es una tarea crucial en diversos ámbitos, incluyendo la integridad académica, la protección de la propiedad intelectual y la detección de plagio. Con el creciente volumen de código disponible públicamente y generado por estudiantes, ha aumentado la necesidad de herramientas de evaluación de similitud que sean escalables y precisas, incluso entre distintos lenguajes de programación. Los enfoques tradicionales basados en la comparación directa de cadenas, análisis de árboles sintácticos o heurísticas suelen quedarse cortos cuando se enfrentan a código estructuralmente modificado, especialmente en casos donde la similitud textual simple no es suficiente para determinar la equivalencia semántica.

Este artículo presenta una solución innovadora basada en el uso combinado de embeddings extraídos del modelo UnixCoder (basado en la arquitectura Transformer) y un

clasificador supervisado entrenado con AutoML (FLAML). La representación vectorial generada por UnixCoder permite capturar características semánticas profundas del código, mientras que la similitud del coseno se utiliza como una métrica clave para determinar la cercanía entre fragmentos. El modelo resultante se entrenó y evaluó utilizando conjuntos de datos provenientes del corpus de código fuente SOCO FIRE 2014, específicamente en los lenguajes Java y C, integrando ejemplos positivos de código funcionalmente equivalentes y negativos generados de forma aleatoria. El objetivo final es crear un sistema robusto, escalable y eficiente que pueda ser aplicado en entornos prácticos a través de distintos lenguajes con una mínima intervención manual, ofreciendo resultados competitivos frente a las técnicas actualmente disponibles en el estado del arte.

2. Related Work

Los trabajos previos relacionados en detección de similitud abarcan una gama amplia de estrategias, incluyendo métodos léxicos (MOSS), métodos sintácticos como comparaciones de árboles sintácticos abstractos (AST), y más recientemente modelos basados en embeddings neuronales. Herramientas como MOSS han sido ampliamente utilizadas en el ambiente académico, ofreciendo algoritmos básicos de coincidencia de tokens mediante n-gramas para detectar plagio (Aiken et al. 2002). Sin embargo, estos métodos son sensibles a cambios superficiales del código y resultan cada vez menos eficaces en entornos modernos más complejos.

Enfoques más sofisticados emplean representaciones semánticas del código como Code2Vec, que genera embeddings mediante rutas en árboles AST para métodos específicos en Java (Alon et al. 2019). Otras investigaciones han explorado redes neuronales basadas en grafos (GNNs), que intentan aprender patrones estructurales y contextuales del código. Estos modelos de aprendizaje automático-profundo mejoran la comprensión semántica, pero requieren de gran capacidad capacidad de cómputo y una cantidad considerable de datos.

Modelos transformadores como CodeBERT y UnixCoder, pre entrenados con grandes corpus de código y lenguaje natural, han demostrado capacidades superiores para la generalización. UnixCoder (Swati et al. 2022), específicamente empleado en este estudio, combina la potencia del pre entrenamiento en grandes corpus de código y lenguaje natural con una eficiente generación de embeddings vectoriales, facilitando tareas como la detección de clones, resumen de código y búsqueda de fragmentos semánticamente similares. La metodología propuesta en nuestro trabajo se beneficia especialmente de la precisión semántica proporcionada por UnixCoder y simplifica notablemente mediante embeddings fácilmente interpretables con la similitud del coseno. Este enfoque destaca por su eficiencia y costo computacional significativamente menor al compararse con modelos más pesados basados en grafos.

Autor	**Año**	**Método**	**Conjunto / Tarea**	**Notas**
Aiken et al.	2002	MOSS	Varios lenguajes	Coincidencia de tokens para plagio
Alon et al.	2019	Code2Vec	Métodos en Java	Embeddings de AST

Swati et al.	2022	UnixCoder	Múltiples clases	Transformer pre entrenado código/texto

Tabla 1. Resumen de enfoques recientes basados en redes neuronales para la detección de similitud de código

3. Metodología y Recursos

3.1 Descripción del dataset

Lenguaje	**Total de pares**	**Positivos (similar = 1)**	**Negativos (similar = 0)**
Java	168	84	84
C	52	26	26

Tabla 2. Estructura de los datos generados a partir del benchmark SOCO FIRE 2014 []

- Pares positivos: Generados a partir de los archivos SOCO14-*.qrel, mediante *generate_dataset.py* (cada línea del qrel indica dos archivos funcionalmente equivalentes).
- Pares negativos: Creados con *generate_negatives.py* seleccionando archivos distintos y validando que no coincidieran con ningún par positivo conocido.
- Combinación y aleatorización: Ambos conjuntos se concatenaron y mezclaron con semilla fija (random_state=42) usando *combine_datasets.py* para evitar sesgos de orden.
- Formato final: CSV con tres columnas: *code1*, *code2* (texto completo de cada fragmento) y *similar* (0/1).

3.2 Intento inicial: SVM + TF-IDF + similitud coseno

1. Representación TF-IDF
   - Vectorizamos cada fragmento con *TfidfVectorizer*, explorando rangos de n-gramas de uno a dos tokens y distintos valores de *max_features*.
2. Similitud de Coseno
   - Para cada par, calculamos la similitud de coseno entre sus vectores TF-IDF.
3. Clasificador SVM
   - Entrenamos un *SVC* sobre la única característica de similitud, ajustando manualmente (*GridSearchCV*) hiperparámetros clave: *kernel* (lineal, RBF) y *gamma*.
4. Preprocesamiento y balanceo
   - Balanceamos las clases 1/0 para evitar sesgo.
5. Resultados
   - El F1-score máximo alcanzado con *SVM+TF-IDF* fue de 72% en Java y 68% en C, tras lo cual los intentos adicionales de ajuste manual no lograron mejorar esas cifras.
6. Limitaciones
   - Alto costo de ajuste manual
   - Sensible a cambios sintácticos y superficiales (de léxica)
   - Rendimiento máximo cerca de 72% para Java y 68% para C, lejos de benchmarks semánticos.

3.3 Transición guiada a AutoML

Para aprovechar la capacidad de FLAML sin delegar totalmente la toma de decisiones, integramos AutoML en un proceso iterativo y controlado por el equipo:

Definición de objetivos y restricciones:

Métrica de interés: F1-Score, por equilibrar precisión y recall en nuestro problema binario.

Presupuesto de tiempo: 1 hora para simular límites reales de proyecto.

Estimadores_candidatos: Para asegurar que FLAML explorara únicamente métodos consolidados en la clasificación de última generación, pre seleccionamos una lista de estimadores basada en referencias clave de los estados del arte. Concretamente, incluimos:

- **LightGBM** (Ke et al. 2017)
- **XGBoost** (Chen & Guestrin, 2016)
- **Random Forest** (Breiman, 2001)
- **Extra Trees** (Geurts et al. 2006)
- **Regresión Logística L1** (Tibshirani, 1996)

De este modo, nuestro proceso de AutoML no fue un "experimento ciego", sino una exploración dirigida sobre las técnicas más relevantes reportadas en el estado del arte.

Ingeniería de características y ajustes previos:

Nuestro objetivo fue proporcionarle a FLAML la mejor calidad en cuestión de datos, por lo que previo a usarlo, realizamos lo siguiente:

- División del conjunto de datos (70% entrenamiento, 30% prueba) con semilla fija.

Ejecución de FLAML y monitoreo:

Durante la búsqueda, revisamos los registros de log generados (*flaml_*.log*) para entender cómo evolucionaban las métricas de cada estimador y detectar si algún modelo quedaba muy por detrás o si sugería ajustes en el rango de hiperparámetros.

3.3.1 Ejecución de FLAML y monitoreo (Java / C)

A continuación, presentamos un resumen de los mejores modelos por lenguaje, extraído directamente de los archivos de log (flaml_java.log, flaml_c.log):

Java:

- Mejor modelo: *XGBClassifier*
- F1-score final (test set): 83%
- Tiempo acumulado de búsqueda: 1012.26 segundos
- Número de combinaciones exploradas: 6519
- Tamaño del conjunto de validación: 51 ejemplos
- Tiempo por predicción: 1.78e-5 segundos

Interpretación: El mejor desempeño en Java se alcanzó con un modelo basado en XGBoost, el cual fue refinado a través de múltiples iteraciones. Este modelo logró un F1-score final del 83 %. Se seleccionó por su equilibrio entre precisión, capacidad de generalización y eficiencia computacional.

C:

- Mejor modelo: *RandomForestClassifier*
- F1-score final (test set): 89%

- Tiempo acumulado de búsqueda: 0.90 segundos
- Tamaño del conjunto de validación: 16 ejemplos

Interpretación: El modelo óptimo para C fue un Random Forest con pocos árboles y profundidad limitada, lo que se ajusta al tamaño reducido del dataset. A pesar del bajo tiempo de búsqueda, la configuración encontrada proporcionó un rendimiento sobresaliente, validado manualmente sobre el conjunto de prueba.

Conclusiones metodológicas:

Una vez que FLAML identificó el mejor modelo para cada lenguaje (según curr_best_record_id en el log), realizamos un análisis manual para validar su calidad y asegurar que la decisión no fuera automática.

Primero, extrajimos el tipo de modelo (learner) y sus hiperparámetros (config) desde el log. Luego, reentrenamos ese modelo con la configuración sugerida, ahora sobre todo el set de entrenamiento, y lo evaluamos sobre el set de prueba reservado. Confirmamos que superaba en F1-score al modelo inicial basado en SVM+TF-IDF. Finalmente, revisamos los tiempos de predicción (pred_time) y validamos que el modelo también fuera eficiente.

En resumen, usamos AutoML como herramienta de apoyo, pero la elección final fue guiada por revisión técnica y decisiones fundamentadas del equipo.

3.4 Enfoque final: UnixCoder + similitud coseno

1. Modelo pre-entrenado
    - microsoft/unixcoder-base (transformer de código) cargado con AutoTokenizer y AutoModel.
2. Vectorización
    - Se tokenizó cada fragmento (máx. 512 tokens), se procesó en lotes de 16 y se aplicó mean pooling sobre la última capa para obtener un vector de dimensión fija.
3. Similitud coseno
    - Con calculate_similarity() calculamos la métrica para cada par de embeddings.

3.5 Consideraciones y mejoras futuras

- Curvas ROC y umbrales personalizados para ajustar sensibilidad/especificidad según la aplicación.
- Fine-tuning de UnixCoder para mejorar la representación de patrones específicos de código.
- Expansión a otros lenguajes.

4. Resultados

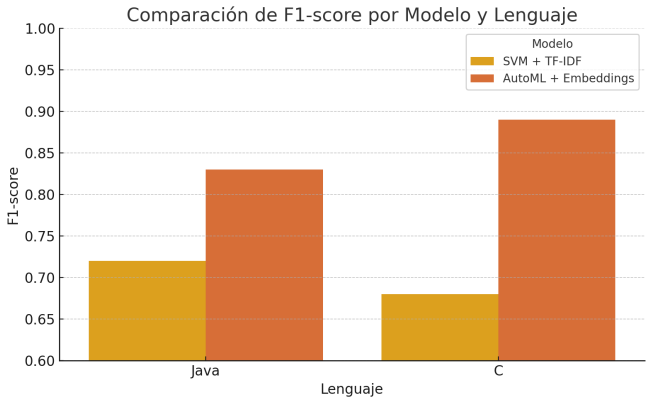
Los resultados obtenidos muestran una mejora significativa al comparar el enfoque final basado en embeddings (UnixCoder) y clasificación automática (FLAML) frente a nuestro modelo inicial SVM con características TF-IDF.

4.1 Métricas de desempeño final

Después de entrenar el modelo final con los embeddings generados por UnixCoder y clasificar con el mejor modelo determinado a partir de los *flaml_*.log*, obtuvimos los siguientes resultados sobre el conjunto de prueba:

Lenguaje	Modelo	Prec.	Recall	F1	Acc.
Java	XGBC	83%	88%	83%	82.3%
C	RF	89%	89%	89%	87.5%

4.2 Comparación con el modelo base (SVM + TF-IDF)



La mejora en el F1-score fue de aproximadamente +11 puntos para Java y +21 puntos para C, lo que valida la elección del enfoque basado en modelos por lenguaje.

5. Discusión

Los resultados obtenidos demuestran que el enfoque basado en embeddings generados por UnixCoder, combinado con un clasificador optimizado mediante AutoML, supera ampliamente a técnicas tradicionales basadas en representaciones léxicas como TF-IDF. Esta mejora refleja la capacidad del modelo para capturar la semántica profunda del código fuente, incluso cuando existen variaciones sintácticas significativas.

Además, el uso de FLAML permitió acelerar significativamente la selección de modelos, sin sacrificar el control sobre el proceso. Lejos de delegar completamente en AutoML, nosotros establecimos los modelos candidatos, el presupuesto de tiempo, la métrica de optimización, y realizamos una validación manual del mejor pipeline. Esto asegura que la solución no fue automática, sino guiada técnicamente por decisiones informadas.

Asimismo, es importante recalcar que FLAML permitió identificar modelos y configuraciones que se ajustaron de forma óptima a las características particulares de cada lenguaje y dataset, lo cual se reflejó en la variación de los estimadores y sus hiperparámetros finales. Esto indica que el comportamiento del modelo fue sensible al tamaño del conjunto, la estructura del código y la complejidad semántica en cada lenguaje, lo cual refuerza nuestro enfoque al evidenciar que distintos contextos requieren de distintos modelos y configuraciones específicas.

Sin embargo, también se identificaron algunas limitaciones:

- El modelo pre entrenado UnixCoder no fue ajustado (fine-tuned) al dominio del conjunto FIRE 2014, lo que podría limitar su capacidad para capturar convenciones específicas del dataset.
- La solución depende del uso de una sola métrica (similitud coseno), lo que puede no ser suficiente para casos más complejos de similitud parcial.

- El tamaño reducido del dataset podría haber generado sobreajuste en algunas configuraciones; esto debe explorarse con conjuntos más grandes y diversos en el futuro.

En conjunto, estos hallazgos respaldan el valor de integrar modelos pre entrenados de última generación con técnicas de selección automatizada bien guiadas, como parte de un enfoque híbrido para la detección de similitud en código fuente.

6. Conclusión

En este trabajo desarrollamos una herramienta efectiva para la detección de similitud entre fragmentos de código, combinando modelos de lenguaje pre entrenados con técnicas automatizadas de selección de clasificadores. A lo largo del proceso, evaluamos enfoques tradicionales (como SVM + TF-IDF) y progresivamente adoptamos estrategias más robustas, culminando en un pipeline basado en embeddings generados por UnixCoder y clasificados mediante AutoML con FLAML.

Los resultados obtenidos muestran mejoras sustanciales en F1-score para ambos lenguajes analizados (Java y C), superando en más de 11 y 21 puntos porcentuales, respectivamente al enfoque base. En definitiva, la combinación de representaciones semánticas profundas y selección automatizada de modelos representa una estrategia eficaz, escalable y adaptable para la detección de similitud entre fragmentos de código.

El uso de FLAML no implicó una pérdida de control por parte del equipo, sino que actuó como un acelerador para explorar múltiples estimadores y configuraciones, siempre dentro de límites definidos y validados manualmente. Esto permitió adaptar los modelos al tamaño, complejidad y estructura de cada conjunto de datos.

Como trabajo futuro, se propone extender la solución a otros lenguajes de programación y aplicar técnicas de fine-tuning para adaptar los embeddings. Asimismo, sería valioso validar el modelo en casos reales de uso.

7. References

Modelos de lenguaje y embeddings para código:

[1] Guo, D., Ren, S., Lu, S., Feng, S., Tang, D., Duan, N., & Zhou, M. (2022). UnixCoder: Unified Cross-Modal Pre-training for Code Understanding and Generation. arXiv preprint arXiv:2203.08387. https://arxiv.org/abs/2203.08387

[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is All You Need. In Advances in Neural Information Processing Systems (pp. 5998–6008). https://arxiv.org/abs/1706.03762

Automated Machine Learning (AutoML) y FLAML:

[3] Wang, Y., Shi, H., Fu, Y., & Li, Y. (2021). FLAML: A Fast and Lightweight AutoML Library. Proceedings of the 30th ACM International Conference on Information and Knowledge Management. https://doi.org/10.1145/3459637.3482231

[4] Microsoft. (2024). FLAML Documentation. https://microsoft.github.io/FLAML/

Modelos utilizados en AutoML:

[5] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., & Liu, T. Y. (2017). LightGBM: A Highly Efficient Gradient Boosting Decision Tree. Advances in Neural Information Processing Systems, 30.

[6] Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785–794).

[7] Breiman, L. (2001). Random Forests. Machine Learning, 45, 5–32.

[8] Geurts, P., Ernst, D., & Wehenkel, L. (2006). Extremely randomized trees. Machine Learning, 63, 3–42.

[9] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society: Series B (Methodological), 58(1), 267–288.

Dataset y evaluación del dominio:

[10] FIRE. (2014). Shared Task on Plagiarism Detection: Source Code (SOCO-2014).

[11] Ebrahim, F., & Joy, M. (2023). Source Code Plagiarism Detection with Pre-Trained Model Embeddings and Automated Machine Learning. In Proceedings of Recent Advances in Natural Language Processing (pp. 301–309). INCOMA Ltd. https://doi.org/10.26615/978-954-452-092-2_034