David Álvarez Fidalgo

Dirigido por Francisco Ortín Soler

Redes siamesas de transformers para el reconocimiento de programadores a partir de su código fuente

Trabajo Fin de Máster

Universidad de Oviedo

Escuela de Ingeniería Informática
Universidad de Oviedo

Máster Universitario en Ingeniería Web
Escuela de Ingeniería Informática
Universidad de Oviedo

2024

Resumen

La verificación de autoría (a veces denominada reconocimiento de autor) consiste en determinar si dos documentos fueron escritos por el mismo autor. Aplicada al código fuente, la verificación de autoría tiene varios usos, como el análisis de autoría de malware, la resolución de disputas de derechos de autor y la detección de plagio. La creación de un modelo robusto de aprendizaje profundo para la verificación de autoría de código fuente es un problema complejo porque necesita ser lo suficientemente general como para poder aplicarse a código escrito por programadores que no fueron utilizados durante su entrenamiento. La arquitectura de aprendizaje profundo Transformer ha logrado recientemente un éxito significativo en varias tareas de procesamiento de código fuente. Sin embargo, ningún modelo de aprendizaje profundo ha aprovechado esta arquitectura para la verificación de autoría de código fuente. En este trabajo, presentamos CLAVE (Contrastive Learning for Authorship Verification with Encoder representations), un nuevo modelo de aprendizaje profundo para la verificación de autoría de código fuente que aprovecha el aprendizaje contrastivo (*contrastive learning*), comúnmente utilizado a través de redes siamesas, y una arquitectura basada en el Transformer Encoder. Inicialmente, preentrenamos CLAVE con un gran conjunto de datos compuesto de archivos de código fuente Python extraídos de GitHub. Posteriormente, realizamos un *fine-tuning* de CLAVE para la verificación de autoría, utilizando aprendizaje contrastivo con código Python obtenido de las competiciones de programación Google Code Jam y Kick Start. Este enfoque permite al modelo aprender representaciones estilométricas del código fuente que pueden ser comparadas utilizando una medida de distancia vectorial, proporcionando así una métrica para la verificación de autoría. Evaluamos CLAVE con un conjunto de pares de código procedentes de estas competiciones de programación. Los resultados muestran que CLAVE supera a los modelos de verificación de autoría de código fuente más avanzados, obteniendo un AUC de 0,9782.

Palabras clave: verificación de autoría de código fuente, aprendizaje contrastivo, transformer encoder, aprendizaje profundo, representaciones estilométricas, Python

Abstract

Authorship verification involves determining whether two documents are written by the same author. When applied to source code, authorship verification has many uses, such as malware authorship analysis, copyright dispute resolution, and plagiarism detection. Creating a robust source code authorship verification model is particularly challenging because it needs to generalize to code written by programmers outside of its training data. The Transformer deep learning architecture has recently achieved significant success in various source code processing tasks. However, no previous deep learning model for source code authorship verification has taken advantage of this architecture. In this paper, we present CLAVE (Contrastive Learning for Authorship Verification with Encoder representations), a novel deep learning model for source code authorship verification that leverages contrastive learning and a Transformer Encoder-based architecture. We initially pre-train CLAVE on a large dataset of Python source code files extracted from GitHub. Subsequently, we fine-tune CLAVE for authorship verification using contrastive learning on Python submissions from the Google Code Jam and Kick Start coding competitions. This approach enables the model to learn stylometric representations of source code, which can be compared using a vector distance measure, thereby providing a metric for authorship verification. Our evaluation on a set of submission pairs from these coding competitions has shown that CLAVE outperforms state-of-the-art source code authorship verification models.

Keywords: Source code authorship verification, contrastive learning, transformer encoder, deep learning, stylometric representations, Python

Índice general

Capítulo 1

Introducción

La atribución de autoría es la tarea de identificar al autor de un documento anónimo dado un conjunto de autores conocidos [27]. La atribución de autoría tiene muchas aplicaciones, incluyendo encontrar autores de obras literarias anónimas o disputadas [51], ayudar en investigaciones criminales [12], o atribuir mensajes asociados a grupos terroristas conocidos [1].

Por otro lado, la verificación (a veces llamada reconocimiento) de autoría tiene como objetivo determinar si dos documentos están escritos por el mismo autor [42]. La verificación de autoría es una tarea compleja porque debe ser lo suficientemente general como para verificar autores no utilizados en el entrenamiento, a diferencia de la atribución de autoría, donde los documentos se clasifican en un conjunto predefinido de autores conocidos. Además, la atribución de autoría puede reducirse a verificación comparando el documento de autoría desconocida con cada documento disponible del conjunto de autores conocidos. Por lo tanto, la verificación de autoría abarca todas las aplicaciones de la atribución y las extiende, incluyendo tareas como verificar la autoría de documentos históricos [29] o detectar plagio [18].

La idea general de la verificación de autoría puede aplicarse al contexto específico del código fuente, donde el objetivo es discernir si dos fragmentos de código fueron escritos por el mismo programador. Es decir, si se dispone de código fuente escrito por un programador, la verificación de autoría permite reconocer a ese mismo programador en otros fragmentos de código. La verificación de autoría de código fuente tiene varios usos en diferentes dominios. Puede usarse para analizar la autoría de malware y vincularlo a otros fragmentos de código malicioso [32]. Además, sirve como herramienta para identificar casos de plagio de código en diversos escenarios, abarcando tanto disputas legales sobre derechos de propiedad intelectual como entornos educativos [26].

La verificación de autoría de código fuente tiene algunas características únicas que la diferencian de su aplicación en el lenguaje natural. En el código fuente, se discierne principalmente el estilo de programación del autor al escribir diferentes programas con la misma semántica general. Ejemplos incluyen la convención de nombrado utilizada para los identificadores (por ejemplo, `camelCase` o `snake_case`), la preferencia por bucles `while` o `for`, la elección entre comillas simples o dobles para cadenas de texto (en lenguajes como Python que soportan ambas), o el uso de construcciones imperativas frente a funcionales donde

```python
def fizz_buzz(max_number: int) -> None:
    for n in range(1, max_number + 1):
        if n % 3 == 0 and n % 5 == 0:
            print("fizzbuzz")
        elif n % 3 == 0:
            print("fizz")
        elif n % 5 == 0:
            print("buzz")
        else:
            print(n)
```

```python
def fizzBuzz(maxNumber):
  number = 0
  while number < maxNumber:
    number += 1
    match number:
      case number if number % 3 == 0 and number % 5 == 0:
        print('fizzbuzz')
      case number if number % 3 == 0:
        print('fizz')
      case number if number % 5 == 0:
        print('buzz')
      case number:
        print(number)
```

Figura 1.1: Dos estilos de programación distintos para una función Python con el mismo propósito.

esa elección está permitida (por ejemplo, C#, Scala y Python). En el ámbito de lenguaje natural, este estudio del estilo de escritura se conoce como estilometría.

La Figura 1.1 ilustra cómo dos funciones en Python con el mismo propósito—el problema FizzBuzz [22]—pueden escribirse en dos estilos diferentes. La primera diferencia radica en la convención de nombrado para los identificadores de las funciones y los parámetros (`fizz_buzz` y `max_number` frente a `fizzBuzz` y `maxNumber`). La primera función utiliza la convención de nombrado *snake case*, mientras que la segunda usa *camel case*. El nombre de la variable que contiene el número actual también es diferente, la primera función usa un nombre más corto (`n`) mientras que la segunda usa uno más largo (`number`). Las estructuras de control empleadas por cada función también son diferentes: la primera usa un bucle `for` con enunciados `if` para seleccionar el mensaje que se imprimirá con cada número, mientras que la segunda usa un bucle `while` y un enunciado `match`. La forma en que se escriben las cadenas de texto también es diferente: la primera función representa las cadenas dentro de comillas dobles, mientras que la segunda emplea comillas simples para este propósito. Finalmente, la primera función incluye anotaciones de tipo para declarar los tipos de los argumentos y del valor de retorno, mientras que la segunda no.

El aprendizaje profundo es una subcategoría del aprendizaje automático que consiste en el uso de redes neuronales con múltiples capas para aprender automáticamente representaciones de datos a diferentes niveles de abstracción [33]. Ha logrado un éxito significativo en varios dominios, incluyendo la visión artificial, el procesamiento de lenguaje natural y el reconocimiento de voz. Debido a sus diversas aplicaciones, varios estudios han explorado el uso del aprendizaje profundo para la verificación de autoría de código fuente. Algunos de ellos entrenan clasificadores para la atribución de autoría y los reutilizan para la verificación de autoría [25]. Otros usan el aprendizaje contrastivo (Sección 2.2) para entrenar redes *feedforward* con características seleccionadas manualmente [49], o Redes Neuronales Recurrentes (RNNs) con características aprendidas [50, 37].

La arquitectura de aprendizaje profundo Transformer [48] ha permitido numerosos avances recientes en el procesamiento de lenguaje natural. Aborda varias limitaciones clave de las RNNs, como la dificultad para modelar dependencias a largo plazo, y permite una mayor paralelización durante el entrenamiento. Diseñados inicialmente para tareas consistentes en transformar una secuencia de entrada en otra de salida, como la traducción automática, los Transformers

constan de dos redes neuronales: un Encoder y un Decoder. El Encoder procesa la secuencia de entrada y el Decoder genera la secuencia de salida condicionada por la salida del Encoder.

Al mismo tiempo, la utilización por separado del Encoder y el Decoder del Transformer ha logrado importantes éxitos en diversas tareas. Por ejemplo, los modelos de solo Decoder, como GPT [8] y LLaMA [44], han sobresalido en la generación de texto, mientras que los modelos de solo Encoder, como BERT [17] y RoBERTa [34], han mostrado su eficacia en la clasificación de texto y otras tareas de comprensión textual. La arquitectura Transformer también ha tenido éxito en tareas de procesamiento de código fuente, con modelos preentrenados como CodeBERT [20] y CuBERT [28]. A pesar de su éxito, ninguna investigación previa ha explorado la aplicación de esta arquitectura a la verificación de autoría de código fuente.

La principal contribución de este trabajo es un nuevo modelo de aprendizaje profundo para la verificación de autoría de código fuente, que aprovecha el aprendizaje contrastivo y una arquitectura basada en el Transformer Encoder. Nuestro sistema, CLAVE (Contrastive Learning for Authorship Verification with Encoder representations), está inicialmente preentrenado en un gran conjunto de datos de archivos de código fuente Python [1] extraídos de GitHub. Exploramos y personalizamos tokenizadores específicamente adaptados para Python con el objetivo de preprocesar eficazmente el código. Posteriormente, realizamos un *fine-tuning* de CLAVE para la verificación de autoría, utilizando aprendizaje contrastivo con código Python obtenido de las competiciones de programación Google Code Jam y Kick Start. Este proceso permite que el modelo aprenda representaciones estilométricas del código fuente en forma de vectores (también llamadas *embeddings*), ya que diferentes programadores escriben soluciones a los mismos problemas en sus propios estilos de programación. Como consecuencia, los *embeddings*, que representan las características estilométricas de dos fragmentos de código fuente, pueden compararse utilizando una medida de distancia vectorial, proporcionando así una métrica para la verificación de autoría. En comparación con los modelos más avanzados, los resultados muestran que CLAVE obtiene el rendimiento más alto para la verificación de autoría de código fuente.

El resto de esta memoria está estructurado de la siguiente manera. Primero, presentamos los trabajos relacionados en el Capítulo 2. Luego, el Capítulo 3 describe la arquitectura de CLAVE y su proceso de entrenamiento. La metodología seguida para obtener los datos de entrenamiento y evaluar el modelo se detalla en el Capítulo 4. En el Capítulo 5, presentamos la evaluación de CLAVE y lo comparamos con sistemas relacionados. El Capítulo 6 discute el impacto de los componentes más relevantes de CLAVE en su rendimiento, y el Capítulo 7 presenta las conclusiones y el trabajo futuro. El Capítulo 8 detalla cómo se ha llevado a cabo la dirección y la gestión del trabajo.

[1] Elegimos entrenar CLAVE con código fuente en Python porque es uno de los lenguajes más populares según diferentes índices de lenguajes de programación [43, 11].

Capítulo 2

Trabajo relacionado

En este capítulo, revisamos investigaciones previas relacionadas con el objetivo de este artículo—el análisis de autoría de código fuente—y las principales técnicas que hemos utilizado para lograrlo: el aprendizaje contrastivo y el procesamiento de lenguaje natural aplicado al código fuente.

2.1 Análisis de autoría de código fuente

El análisis de autoría abarca un conjunto de tareas centradas en determinar información sobre el autor de un documento dado [42]. Incluye dos tareas principales: la atribución de autoría y la verificación de autoría. La atribución de autoría tiene como objetivo identificar al autor de un documento a partir de un conjunto de autores conocidos, mientras que la verificación de autoría busca determinar si dos documentos fueron escritos por el mismo autor.

Se han realizado varias investigaciones sobre la verificación de autoría de código fuente. Ou *et al.* [37] proponen una arquitectura de red neuronal que combina una red Long Short-Term Memory (LSTM) bidireccional con una Generative Adversarial Network (GAN). Esta arquitectura se entrena como una red siamesa para aprender representaciones estilométricas del código fuente, con la GAN asegurando que las representaciones derivadas no contengan información sobre la funcionalidad de los archivos de código fuente originales. Estas representaciones luego se comparan usando la similitud del coseno para determinar si un par de archivos de código fuente fueron escritos por el mismo autor o no. La red supera a varios modelos de representación de código de última generación en cuatro conjuntos de datos extraídos de la competición de programación Google Code Jam 2017.

White *et al.* [50] también presentan una arquitectura de red neuronal basada en un LSTM bidireccional que aprende representaciones estilométricas del código fuente. Los autores entrenan esta red utilizando la función de pérdida NT-Xent que forma parte del *framework* SimCLR [14]. La red entrenada se evalúa para la verificación de autoría comparando la similitud del coseno entre las representaciones. También se utiliza para la atribución de autoría con un clasificador de máquina de vectores de soporte. La red alcanza una exactitud de verificación del 92,94 % en un conjunto de datos que consiste en programas C/C++ enviados entre los años 2008 y 2017 a Google Code Jam.

Hozhabrierdi *et al.* [25] proponen un método para la verificación de autoría que emplea una red neuronal originalmente entrenada para la atribución de autoría. La red recibe como entrada unas características novedosas llamadas Variable-Independent Nested Bigrams, que se extraen del árbol de sintaxis abstracta de cada programa. La red se entrena inicialmente para clasificar archivos de código fuente en un conjunto de autores conocidos. Luego, se reutiliza una parte de la red entrenada para generar representaciones del código fuente, que posteriormente se comparan para la verificación de autoría. Este método obtiene un AUC (área bajo la curva ROC) de 0,96 en un conjunto de programas Python enviados a Google Code Jam entre los años 2006 y 2014.

Wang *et al.* [49] introducen una red neuronal que acepta como entrada un conjunto de características diseñadas manualmente que son extraídas de pares de programas, y genera la probabilidad de que el par haya sido escrito por el mismo autor. Estas características incluyen tanto información estática (extraída sin ejecutar el programa) como dinámica (extraída después de ejecutar el programa), como el tiempo de ejecución o el consumo de memoria. Los autores obtienen una exactitud de verificación del 99,91 % en un conjunto de programas Python recopilado de las ediciones de Google Code Jam celebradas entre los años 2008 y 2017. Sin embargo, no proporcionan métricas adicionales ni información sobre la distribución de clases en el conjunto de datos.

Aunque menos relacionados con este trabajo, existen más estudios que se centran en la atribución de autoría de código fuente. Caliskan-Islam *et al.* [10] proponen un conjunto de características extraídas manualmente del árbol de sintaxis abstracta de cada programa. Estas características se utilizan para entrenar clasificadores que tienen como objetivo identificar al autor de un programa entre un conjunto de autores dados. Abuhamad *et al.* [2] introducen varias Redes Neuronales Convolucionales (CNNs) que toman como entrada *embeddings* de palabras o características TF-IDF y son entrenadas para la atribución de autoría.

2.2 Aprendizaje contrastivo

El aprendizaje contrastivo es una técnica autosupervisada de aprendizaje profundo que tiene como objetivo aprender representaciones de datos de manera que la distancia en el espacio de representación sea menor para instancias similares y mayor para las no similares [23]. Su idea central es aprender representaciones útiles contrastando pares de ejemplos positivos (similares) contra pares negativos (no similares).

Esta técnica ha sido aplicada con éxito en visión artificial para tareas como el reconocimiento facial y la clasificación de imágenes. Uno de los primeros estudios sobre aprendizaje contrastivo, realizado por Chopra *et al.* [15], presenta un método para aprender una métrica de similitud a partir de datos. Los autores utilizan este método para entrenar una Red Neuronal Convolucional (CNN) que aprende representaciones de imágenes faciales. La CNN se entrena con pares de rostros de la misma persona y de distintas personas, con el objetivo de reducir la distancia entre los pares coincidentes mientras se aumenta la distancia entre los no coincidentes. Durante el entrenamiento, las dos imágenes de cada par son procesadas por dos redes que comparten los mismos parámetros, razón por

la que es habitual referirse a estas redes como redes siamesas [7]. Schroff *et al.* [41] también entrenan una CNN con el mismo objetivo, pero su enfoque utiliza tripletas compuestas de un rostro de referencia, un rostro coincidente y un rostro no coincidente.

El aprendizaje contrastivo también ha tenido éxito en el procesamiento de lenguaje natural. Reimers *et al.* [38] proponen una modificación de BERT [17] que utiliza redes siamesas y de tripletas para aprender representaciones semánticas de oraciones que pueden ser comparadas usando la similitud del coseno.

Debido a la naturaleza de la verificación de autoría de código fuente, el aprendizaje contrastivo es una técnica popular para esta tarea, como se muestra en la Sección 2.1. También se ha aplicado a la verificación de autoría en documentos de lenguaje natural. Tyo *et al.* [46] también modifican BERT usando diferentes funciones objetivo para aprender representaciones de documentos que están más cerca cuando los documentos son escritos por los mismos autores, y más lejos cuando no lo son.

Finalmente, SimCLR [14] es un *framework* popular de aprendizaje contrastivo autosupervisado, diseñado originalmente para visión artificial. En SimCLR, las instancias similares se generan mediante *data augmentation*. Durante el entrenamiento, para cada instancia en un *batch*, se crean dos nuevas instancias usando *data augmentation*. Luego, el modelo se actualiza utilizando una función de pérdida llamada NT-Xent, que tiene como objetivo acercar las representaciones de instancias que fueron creadas a partir de la misma imagen original, mientras aleja las representaciones de instancias que se originaron a partir de imágenes diferentes.

2.3 Procesamiento de lenguaje natural aplicado al código fuente

Las técnicas de aprendizaje profundo para el procesamiento de lenguaje natural (NLP) han experimentado un avance significativo desde la introducción de la arquitectura Transformer [48]. Muchos de estos avances en NLP también son efectivos cuando se aplican al código fuente.

Feng *et al.* [20] proponen CodeBERT, un modelo bimodal preentrenado para código fuente y lenguaje natural basado en las arquitecturas de BERT [17] y RoBERTa [34]. Los autores preentrenan el modelo utilizando tanto código fuente como lenguaje natural, y lo entrenan para dos tareas de más bajo nivel: búsqueda de código en lenguaje natural y generación de documentación de código.

Kanade *et al.* [28] introducen CuBERT, un modelo preentrenado diseñado para generar *embeddings* contextuales a partir de código fuente. CuBERT también se basa en la arquitectura de BERT. Los autores preentrenan este modelo utilizando un conjunto de datos compuesto de archivos Python extraídos de GitHub y lo entrenan para varias tareas de comprensión de código, como la detección de mal uso de variables y la detección de discordancia entre funciones y sus descripciones.

Capítulo 3

Descripción del sistema

En esta capítulo, describimos nuestro sistema, llamado CLAVE: Contrastive Learning for Authorship Verification with Encoder representations. Detallamos su arquitectura, cómo se introduce el código fuente en el sistema, la forma en que el modelo se preentrena con código fuente de autores desconocidos y cómo se realiza el *fine-tuning* para la verificación de autoría.

3.1 Arquitectura

La Figura 3.1 muestra cómo CLAVE toma el código fuente como entrada y produce un único vector (*embedding*). El sistema se entrena (Secciones 3.3 y 3.4) de tal manera que los *embeddings* de salida contengan representaciones estilométricas del código fuente de entrada. Es decir, los *embeddings* que representan código escrito por el mismo autor estarán cerca entre sí, mientras que los que representan diferentes autores estarán más alejados. Durante la inferencia, si la distancia del coseno ($1 -$ similitud del coseno) entre los *embeddings* de dos fragmentos de código fuente generados por CLAVE es mayor que un umbral γ, predecimos que los fragmentos fueron escritos por diferentes autores. Si la distancia es menor o igual a γ, predecimos que fueron escritos por el mismo autor.

El primer componente de CLAVE (Figura 3.1) es un tokenizador que toma el código fuente de entrada y lo convierte en un vector (detallado en la Sección 3.2). Este vector se pasa a un Transformer Encoder de 6 capas, que produce una matriz que contiene un *embedding* para cada token de entrada. El Transformer Encoder sigue la arquitectura propuesta por Vaswani *et al.* [48]. Similar a BERT [17], usamos *embeddings* posicionales aprendidos en lugar de codificación sinusoidal y la función de activación GELU en lugar de ReLU, ya que GELU ha obtenido un rendimiento superior en varias tareas de procesamiento de lenguaje [24]. Configuramos el Transformer Encoder con una dimensión del modelo $d_{model} = 512$, una dimensión de la red *feedforward* $d_{ff} = 2048$ y $h = 8$ cabezas de atención.

Posteriormente, la matriz de *embeddings* de tokens producida por el Encoder se agrega para obtener un único *embedding* que representa todo el código fuente de entrada. Esto se logra calculando la media de cada posición del *embedding* entre todos los tokens, resultando en un único *embedding* con las mismas di-

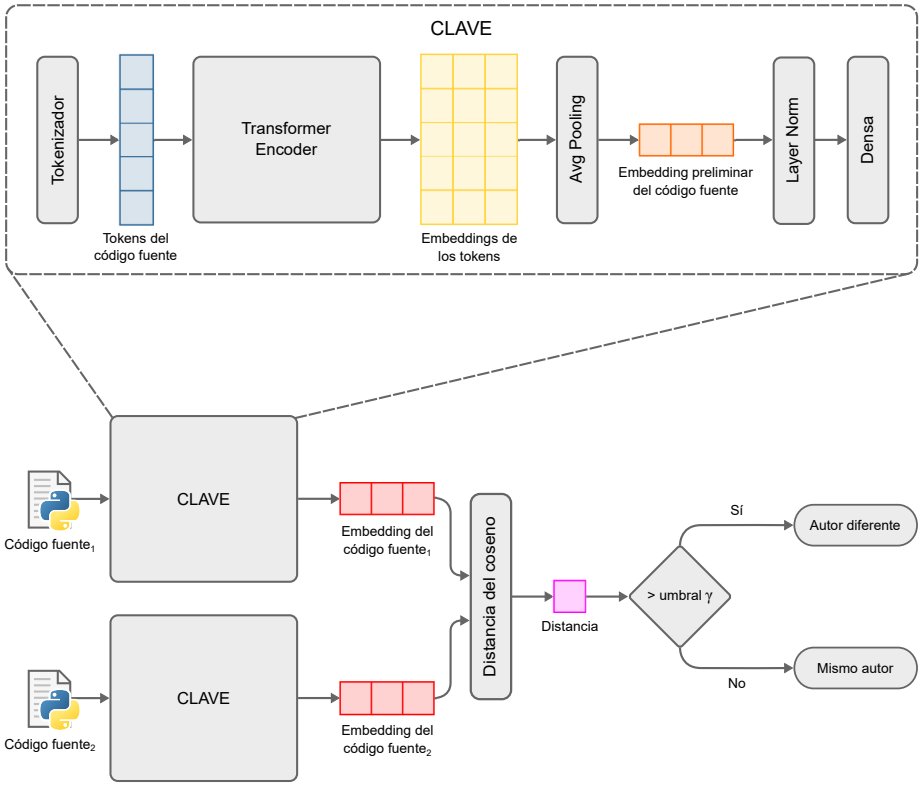


Figura 3.1: La arquitectura de CLAVE.

mensiones que los *embeddings* individuales de los tokens. Este proceso asegura que siempre obtengamos un solo vector independientemente de la longitud de la entrada.

El *embedding* del código fuente se normaliza usando Layer Normalization para mejorar la estabilidad y el rendimiento de la red durante el entrenamiento [5]. El vector normalizado se procesa finalmente por una capa densa con una función de activación ReLU. Esta última capa permite que el modelo aprenda una nueva representación basada en la información sobre toda la entrada contenida en el *embedding*. Esta representación es la salida del modelo y contiene la información estilométrica del código fuente.

3.2 Representación de la entrada

Una parte importante del sistema es cómo representar el código fuente de Python como vectores. Experimentamos con tres enfoques diferentes de tokenización: SentencePiece personalizado para código Python, un nuevo tokenizador diseñado específicamente para Python al que llamamos tokenizador estilométrico, y una combinación de SentencePiece personalizado y el tokenizador estilométrico.

3.2.1 SentencePiece para Python

SentencePiece es un tokenizador de subpalabras desarrollado por Google [31]. A diferencia de los métodos tradicionales, como la tokenización basada en palabras o caracteres, SentencePiece opera a nivel de subpalabras, dividiendo el texto en piezas más pequeñas que pueden corresponder a palabras completas o partes de palabras. Esta característica es particularmente útil para tokenizar código fuente, donde el vocabulario es típicamente ilimitado. Por ejemplo, los identificadores en Python pueden tener cualquier longitud, resultando en un vocabulario infinito.

SentencePiece requiere ser entrenado en un corpus de datos textuales antes de que pueda ser utilizado para la tokenización. Durante este entrenamiento, el tokenizador aprende un vocabulario de subpalabras basado en la frecuencia y probabilidad de diferentes secuencias de caracteres en los datos de entrenamiento. Entrenamos SentencePiece usando el conjunto de datos de preentrenamiento descrito en la Sección 4.1.1, estableciendo el tamaño del vocabulario en 16,000 tokens.

Para tokenizar eficientemente el código Python y retener parte de la estructura del programa original en su forma vectorial, instruimos a SentencePiece para que extraiga los siguientes elementos del lenguaje Python como un solo token: palabras clave, operadores, signos de igual, paréntesis, llaves, corchetes, dos puntos, punto y coma, comas, puntos y puntos suspensivos. También configuramos SentencePiece para extraer espacios, tabuladores y saltos de línea como tokens individuales, ya que pueden ser útiles para determinar el estilo de codificación (por ejemplo, espacios frente a tabuladores en la Figura 1.1).

3.2.2 Tokenizador estilométrico

Este tokenizador es una versión modificada del módulo `tokenize` de Python. Ese módulo es parte de la biblioteca estándar y proporciona funcionalidad para el análisis léxico del código Python. Descompone el código fuente de Python en tokens, que son las unidades más pequeñas de un programa, como palabras clave, identificadores, literales y operadores.

Modificamos el tokenizador para generar tokens adicionales asociados con las características estilométricas del código fuente. Estos tokens adicionales, listados en la Tabla 3.1, abarcan varios aspectos como las diferentes formas en que se pueden escribir literales numéricos y de cadenas, las diversas convenciones de nombrado para identificadores, y si los comentarios comienzan con un espacio en blanco siguiendo la guía de estilo de Python [47]. Además de los tokens listados en la Tabla 3.1, agregamos tokens para espacios, tabuladores y saltos de línea, que de otra manera sería ignorado por el módulo `tokenize`.

3.2.3 Tokenizador estilométrico con SentencePiece

Esta es una extensión del tokenizador estilométrico (Sección 3.2.2) que incorpora los tokens obtenidos al aplicar SentencePiece a los identificadores. Así, los identificadores están representados tanto con los tokens en la Tabla 3.1 como con las subpalabras obtenidas de otro tokenizador SentencePiece para Python (Sección 3.2.1).

Token	Descripción	Ejemplo
NUMBER_UNDERSCORE	Número con barras bajas	1_000
NUMBER_HEX	Número hexadecimal	0xa0
NUMBER_BIN	Número binario	0b10
NUMBER_OCT	Número octal	0o12
NUMBER_EXP	Número exponencial	1e-10
NUMBER_IMG	Número complejo	1j
NUMBER_LCASE	Número que contiene una minúscula	1e2
NUMBER_UCASE	Número que contiene una mayúscula	1E2
NUMBER_LEADING_DOT	Número que empieza por un punto	.1
NUMBER_TRAILING_DOT	Número que acaba con un punto	1.
NUMBER_EXP_PLUS	Número exponencial con un signo más	1e+2
NUMBER_LEADING_PLUS	Número que empieza por un signo más	+1
NUMBER_LEADING_ZEROS	Número que empieza con ceros	001
NUMBER_TRAILING_ZEROS	Número decimal que acaba con ceros	1.100
STRING_SQUOTE	Cadena rodeada de comillas simples	'hello'
STRING_DQUOTE	Cadena rodeada de comillas dobles	"hello"
STRING_TRIPLE_SQUOTE	Cadena rodeada de comillas simples triples	'''hello'''
STRING_TRIPLE_DQUOTE	Cadena rodeada de comillas dobles triples	"hello"
STRING_MULTILINE	Cadena de varias líneas	"he\llo
STRING_PREFIX_B	Cadena de bytes	b"Hello"
STRING_PREFIX_R	Cadena cruda	r"Hello"
STRING_PREFIX_U	Cadena Unicode	u"Hello"
STRING_PREFIX_F	Cadena formateada	f"{1 + 1}"
STRING_PREFIX_LCASE	Cadena con un prefijo en minúsculas	f"{1 + 1}"
STRING_PREFIX_UCASE	Cadena con un prefijo en mayúsculas	F"{1 + 1}"
NAME_CAMEL_CASE	Identificador escrito en *camel case*	camelCase
NAME_PASCAL_CASE	Identificador escrito en *Pascal case*	PascalCase
NAME_SNAKE_CASE	Identificador escrito en *snake case*	snake_case
NAME_UPPERCASE	Identificador escrito en mayúsculas	ALL_CAPS
NAME_MIXED_CASE	Identificador escrito de otra manera	Mixed_Case
COMMENT_LEADING_WS	Comentario con un espacio después de "#"	# Comment

Tabla 3.1: Los tokens especiales generados por el tokenizador estilométrico.

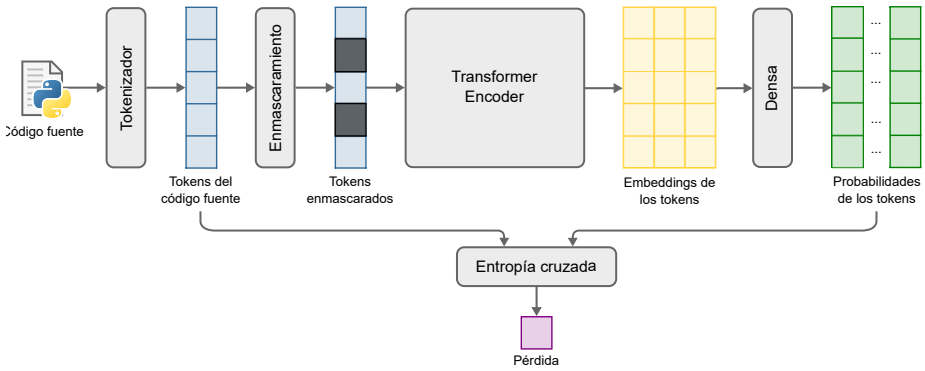


Figura 3.2: El proceso de preentrenamiento de CLAVE.

Entrenamos un tokenizador SentencePiece adicional en el conjunto de los identificadores extraídos del código fuente de preentrenamiento (Sección 4.1.1). El tamaño del vocabulario de este tokenizador SentencePiece se configura para que el número total de tokens únicos, incluidos los del tokenizador estilométrico, sea de 16,000.

3.3 Preentrenamiento

Preentrenamos CLAVE utilizando modelado de lenguaje enmascarado (MLM, por sus siglas en inglés), siguiendo el enfoque propuesto por BERT [17] y refinado por RoBERTa [34]. Para ello, tokenizamos archivos de código fuente del conjunto de datos de preentrenamiento y enmascaramos y mutamos aleatoriamente sus tokens. Luego, entrenamos el Transformer Encoder de CLAVE para predecir los tokens correctos que deberían ocupar las posiciones enmascaradas y mutadas. Optamos por MLM porque parece facilitar el aprendizaje de cómo distinguir diferentes estilos de codificación: una posición enmascarada puede ser ocupada por diferentes tokens, pero solo el que se alinea con el estilo de codificación del autor, dado el contexto del resto del código, será correcto.

La Figura 3.2 ilustra el proceso de preentrenamiento de CLAVE. Inicialmente, se enmascara el vector de tokens del código fuente. Para ello, se selecciona aleatoriamente el 15 % de los tokens. Entre estos tokens seleccionados, el 80 % se reemplaza con un token especial de máscara, el 10 % se reemplaza con un token aleatorio y el 10 % restante se deja sin cambios. Los tokens enmascarados se introducen en el Transformer Encoder de CLAVE, generando una matriz de *embeddings* de tokens (un *embedding* por token). Posteriormente, cada *embedding* se proyecta en un vector que contiene la probabilidad para cada token del vocabulario mediante una capa densa con una función de activación softmax. Finalmente, la pérdida de preentrenamiento se calcula como la entropía cruzada entre la distribución de probabilidad generada y la de los tokens originales, solo para las posiciones seleccionadas en el primer paso.

3.4 *Fine-tuning*

Después de preentrenar el Transformer Encoder, llevamos a cabo el *fine-tuning* de CLAVE para la verificación de autoría de código fuente. Para ello, integramos el Transformer Encoder preentrenado con el resto del modelo y seguimos el proceso de aprendizaje contrastivo descrito en el *framework* SimCLR [14], adaptado a la verificación de autoría por White *et al.* [50]. Aunque en un inicio habíamos propuesto utilizar una red siamesa (Sección 2.2) para realizar la verificación de autoría, finalmente decidimos seguir el enfoque del *framework* SimCLR, puesto que obtuvimos mejores resultados en esta tarea (ver Sección 6.2, la red siamesa se corresponde con la función de pérdida contrastiva).

La Figura 3.3 muestra el proceso de *fine-tuning* de CLAVE. Cada *batch* se construye a partir del conjunto de datos de *fine-tuning* muestreando aleatoriamente N autores sin reemplazo. Luego, seleccionamos aleatoriamente dos archivos de código fuente escritos por cada uno de los N autores y los introducimos en CLAVE. Las $2N$ instancias de CLAVE que procesan cada archivo del *batch* comparten los mismos pesos. Los $2N$ *embeddings* de código fuente resultantes se utilizan finalmente para calcular la pérdida. Empleamos la función de pérdida NT-Xent (Entropía Cruzada Escalada por Temperatura Normalizada), como sugiere el *framework* SimCLR.

La función de pérdida NT-Xent $\ell_{i,j}$ se define para un par de archivos de código fuente (i, j) escritos por el mismo autor con la Ecuación 3.1. En esa ecuación, $\mathbf{z}$ denota un lote de *embeddings* de código fuente, donde $\mathbf{z}_i$ y $\mathbf{z}_j$ representan los *embeddings* de los dos archivos escritos por el mismo autor (par positivo). $sim(\mathbf{z}_k, \mathbf{z}_l)$ denota la similitud del coseno entre los *embeddings* $\mathbf{z}_k$ y $\mathbf{z}_l$, y τ es un hiperparámetro de temperatura utilizado para controlar la agudeza de la distribución [14].

$$\ell_{i,j} = -\log \frac{\exp(sim(\mathbf{z}_i, \mathbf{z}_j)/\tau)}{\sum_{\substack{k=0 \\ k \neq i}}^{2N} \exp(sim(\mathbf{z}_i, \mathbf{z}_k)/\tau)} \tag{3.1}$$

La función de pérdida NT-Xent convierte la verificación de autoría en una tarea de clasificación. Concretamente, su objetivo es identificar, para cada archivo de código fuente en un *batch*, el archivo correspondiente escrito por el mismo autor.

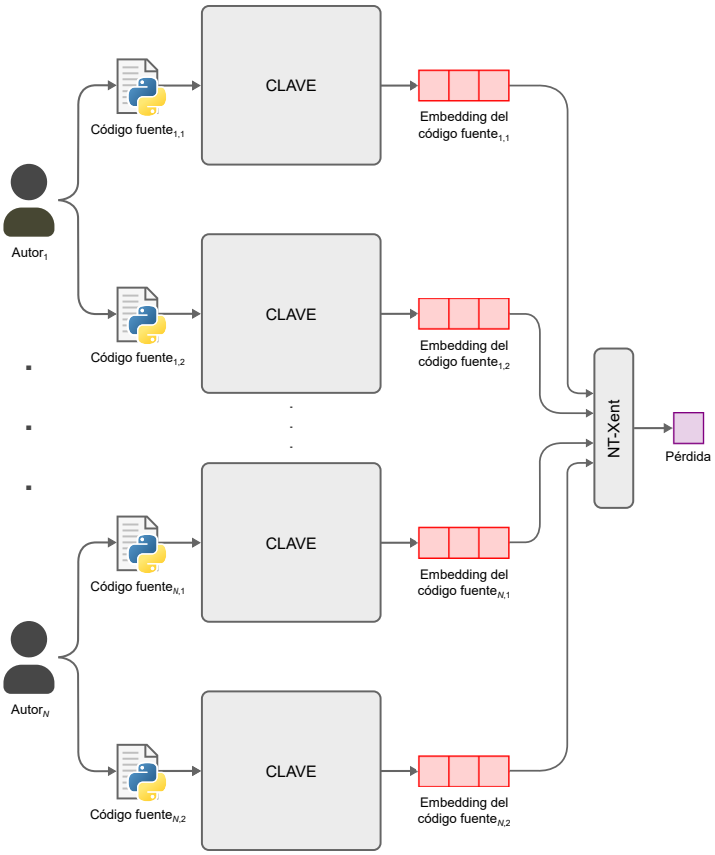


Figura 3.3: El proceso de *fine-tuning* de CLAVE.

Capítulo 4

Metodología

Esta sección describe los conjuntos de datos utilizados para el preentrenamiento y *fine-tuning* de CLAVE, los experimentos realizados para evaluar su rendimiento y el impacto de diversos factores en el mismo, y los detalles sobre cómo se llevaron a cabo esos experimentos.

4.1 Conjuntos de datos

4.1.1 Conjunto de datos de preentrenamiento

Nuestro conjunto de datos de preentrenamiento está compuesto de archivos Python obtenidos de GitHub. Utilizamos la API de GitHub para buscar repositorios de Python con licencias de código abierto permisivas (MIT y Apache), evitando así problemas de derechos de autor y respetando las preferencias de los propietarios de los repositorios. Excluimos repositorios con menos de 100 estrellas en GitHub para mantener un estándar mínimo de calidad, y aquellos identificados como malware por el antivirus del sistema (Windows Defender).

Muestreamos 6.132 repositorios de la API de GitHub. Posteriormente, clonamos los repositorios, eliminamos todos los archivos sin la extensión ".py" y procedemos con los siguientes pasos de preprocesamiento. Primero eliminamos archivos vacíos. Luego, para excluir código generado automáticamente, filtramos archivos mayores de 1 MB o con líneas de más de 1.000 caracteres, como sugieren Chen *et al.* [13]. Finalmente, eliminamos los encabezados de licencia que se pueden encontrar en muchos archivos de código fuente, ya que están escritos en lenguaje natural.

Los 270.602 archivos Python resultantes se dividen en dos particiones: 90 % (243.542 archivos) para entrenamiento y 10 % (27.060 archivos) para validación.

4.1.2 Conjunto de datos de *fine-tuning*

Realizamos el *fine-tuning* de CLAVE con archivos Python 3 enviados a Google Code Jam y Kick Start. Estas competiciones de programación, organizadas por Google desde 2003 hasta 2023, proporcionan a los participantes enunciados de problemas que deben resolver escribiendo código dentro de un margen de tiempo. Seleccionamos esta fuente porque proporciona una gran cantidad de

Partición	Número de autores	Número de archivos
Entrenamiento	61,956	655,542
Validación	7,618	81,955
Evaluación	8,129	81,908

Tabla 4.1: Estadísticas del conjunto de datos de *fine-tuning*.

archivos de código fuente asociados con sus respectivos autores. Además, ha sido utilizada consistentemente para el entrenamiento y evaluación en trabajos anteriores sobre verificación de autoría de código fuente [49, 25, 50, 37].

Dado que el repositorio de competiciones de programación de Google ya no está disponible, obtenemos el código de un archivo no oficial [45]. Este archivo contiene código fuente enviado a Google Code Jam desde 2018 hasta 2023 y a Kick Start desde 2019 hasta 2022.

Decidimos utilizar exclusivamente archivos de código Python 3 para asegurarnos de que el modelo no interprete las diferencias sintácticas entre Python 2 y 3 como diferencias estilométricas de los autores.

Finalmente, dividimos el conjunto de autores de los archivos de código fuente seleccionados en tres particiones: 80 % para entrenamiento, 10 % para validación y 10 % para evaluación. Dividir el conjunto de datos por autores asegura que el modelo no se evalúe con autores cuyos archivos fueron utilizados en el entrenamiento. La Tabla 4.1 muestra el número de autores únicos y archivos de código fuente en cada partición.

4.2 Evaluación

Para evaluar CLAVE, tratamos la verificación de autoría como una tarea de clasificación binaria. El objetivo es clasificar pares de fragmentos de código fuente en dos clases: escritos por diferentes autores (clase positiva[1]) o por el mismo autor (clase negativa). Utilizamos varias métricas de clasificación binaria para evaluar el rendimiento de CLAVE y compararlo con los sistemas de referencia (Sección 4.3).

Para obtener los pares, muestreamos aleatoriamente 100.000 pares de archivos de código fuente de la partición de evaluación del conjunto de datos de *fine-tuning* (Sección 4.1.2), asegurando un número igual de pares escritos por diferentes autores y pares escritos por el mismo autor. A continuación, filtramos los pares que contienen archivos que no se pueden analizar con el módulo `ast` de Python, ya que uno de los sistemas de referencia con los que comparamos CLAVE requiere que el código fuente sea sintácticamente correcto. Finalmente, equilibramos el conjunto de datos de modo que la mitad de los pares seleccionados sean escritos por diferentes autores y la otra mitad por el mismo autor. Para lograr esto, eliminamos pares de la clase sobrerrepresentada hasta que el conjunto de datos esté equilibrado. El conjunto final contiene 85.286 pares, con 42.643 pares para cada clase.

[1] Dado que calculamos la distancia entre los dos *embeddings* y comparamos ese valor con un umbral dado, mantenemos los valores más altos como la clase positiva (diferentes autores) y los valores más bajos como la negativa (mismos autores).

Como se menciona en la Sección 3.1, necesitamos seleccionar un umbral γ para que nuestro sistema funcione como un clasificador binario. Por lo tanto, tomamos el primer 10 % de los pares (8.528 pares) para seleccionar γ, y el 90 % restante (76.758 pares) para calcular las métricas de evaluación. Para el 10 % de pares seleccionados, elegimos el valor de γ que logre el mayor F_1-score. En caso de empate, seleccionamos el valor con la mayor exactitud.

Utilizamos las siguientes métricas de evaluación, donde TP (verdaderos positivos) es el número de pares clasificados correctamente como escritos por diferentes autores, TN (verdaderos negativos) es el número de pares clasificados correctamente como escritos por el mismo autor, FP (falsos positivos) es el número de pares clasificados incorrectamente como escritos por diferentes autores, y FN (falsos negativos) es el número de pares clasificados incorrectamente como escritos por el mismo autor:

- **AUC.** El área bajo la curva ROC [19] proporciona una medida agregada del rendimiento para todos los posibles valores de γ. Puede interpretarse como la probabilidad de que la distancia entre los *embeddings* de dos fragmentos de código fuente escritos por diferentes autores sea mayor que la distancia entre los *embeddings* de dos fragmentos escritos por el mismo autor. Esta métrica es particularmente relevante porque no depende del valor de γ seleccionado.

- **Exactitud.** El ratio de pares correctamente clasificados (Ecuación 4.1).

$$Exactitud = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.1}$$

- **Precisión.** El ratio entre las predicciones positivas correctas y el total de predicciones positivas (Ecuación 4.2). Mide la exactitud de las predicciones positivas. En nuestro problema, es el ratio de pares clasificados correctamente entre aquellos clasificados como escritos por diferentes autores.

$$Precisión = \frac{TP}{TP + FP} \tag{4.2}$$

- **Exhaustividad.** El ratio entre las predicciones positivas correctas y el total de instancias de la clase positiva (Ecuación 4.3). La exhaustividad mide la capacidad del modelo para encontrar todos los casos positivos dentro de un conjunto de datos. Es decir, el ratio entre los pares clasificados correctamente entre aquellos escritos por diferentes autores.

$$Exhaustividad = \frac{TP}{TP + FN} \tag{4.3}$$

- **F_1-score.** La media armónica de precisión y exhaustividad, que representa ambas métricas con un solo valor (Ecuación 4.4).

$$F_1\text{-}score = \frac{2TP}{2TP + FP + FN} \tag{4.4}$$

Finalmente, empleamos bootstrapping con 1.000 repeticiones para calcular intervalos de confianza del 95 % para cada métrica [16]. Este método consiste

en muestrear repetidamente con reemplazo del conjunto de pares de evaluación para crear múltiples conjuntos de datos remuestreados. Luego, calculamos los intervalos de confianza para el promedio de cada métrica en todos los conjuntos de datos remuestreados y así determinar si existen diferencias estadísticamente significativas entre los sistemas comparados [21].

4.3 Sistemas de referencia

Comparamos el rendimiento de CLAVE con los siguientes métodos de verificación de autoría de código fuente descritos en la Sección 2.1: FDR [25], SCS-Gan [37], y el modelo propuesto por White *et al.* [50]. Excluimos SUNDAE [49] debido a que su requisito de ejecutar los archivos de código fuente de entrada supone un riesgo de seguridad. Estos son, hasta donde sabemos, los modelos de aprendizaje profundo más avanzados para la verificación de autoría de código fuente. Además, comparamos CLAVE con CodeBERT [20] (Sección 2.3), un modelo popular y avanzado para código fuente que ha sido utilizado con éxito para generación de código [36], búsqueda [20], resumido [3], traducción [40], y detección de defectos [9].

Dado que el código fuente y los datos de FDR y SCS-Gan no están disponibles, y el trabajo de White *et al.* solo considera código fuente C/C++, replicamos el entrenamiento de sus modelos siguiendo los procesos e hiperparámetros que detallan en sus publicaciones. Nuestra única desviación es entrenar el modelo propuesto por White *et al.* con un tamaño *batch* de 128 en lugar de los 1.000 sugeridos debido a limitaciones de memoria. Utilizamos el mismo conjunto de datos de *fine-tuning* que con CLAVE (Sección 4.1.2) para entrenar estos modelos. Dado que la salida de CodeBERT es una matriz de *embeddings* de tokens, promediamos los *embeddings* para obtener uno solo (como hacemos con CLAVE), que luego usamos para la verificación de autoría.

En la evaluación, utilizamos como entrada los primeros 512 tokens de cada archivo Python, rellenando las secuencias más cortas con un token especial. Seleccionamos esta longitud porque es la longitud máxima de secuencia que soporta CodeBERT. FDR recibe como entrada un conjunto de bigramas anidados en lugar de una secuencia de tokens, por lo que establecemos el tamaño de entrada en 8000 bigramas como sugieren los autores.

4.4 Impacto de los componentes principales

Algunos componentes de CLAVE pueden tener un impacto significativo en su rendimiento para la verificación de autoría de código fuente. Nos gustaría no solo seleccionar los componentes que ofrecen el mejor rendimiento, sino también cuantificar sus beneficios. En esta sección, describimos los experimentos realizados para medir el impacto de estos componentes. Posteriormente, la Sección 4.5 detalla el proceso utilizado para el ajuste de hiperparámetros de CLAVE.

Para seleccionar el mejor tokenizador de los presentados en la Sección 3.2, repetimos los procesos de preentrenamiento y *fine-tuning* con cada uno de los tres tokenizadores propuestos. También incluimos un tokenizador SentencePiece no personalizado para Python para cuantificar los posibles beneficios de la personalización. Una vez que identificamos el tokenizador con el mejor rendimiento

Configuración	d_{model}	d_{ff}	h
Base	512	2048	8
Pequeña	256	2048	8
Grande	768	3072	12

Tabla 4.2: Las diferentes configuraciones del Transformer Encoder evaluadas. d_{model}, d_{ff} y h denotan, respectivamente, la dimensión del modelo (es decir, el tamaño de los *embeddings*), la dimensión de la red *feedforward* y el número de cabezas de atención.

(usando el AUC), lo utilizamos para los experimentos restantes, incluida la comparación con los sistemas de referencia.

Además, una parte clave del proceso de *fine-tuning* (Sección 3.4) es la función de pérdida. Evaluamos el impacto de reemplazar NT-Xent con dos objetivos populares de aprendizaje contrastivo descritos en la Sección 2.2: pérdida contrastiva [15] y pérdida de tripletas [41]. Además, experimentamos con dos tamaños de *batch* diferentes para NT-Xent: 16 y 32.

Otro factor importante en el proceso de entrenamiento de CLAVE es el preentrenamiento. Ningún trabajo previo sobre verificación de autoría de código fuente ha aprovechado la idea de preentrenar el modelo utilizando un conjunto de datos más grande. Evaluamos su contribución al rendimiento de nuestro modelo comparando los resultados de CLAVE con y sin preentrenamiento.

El último componente que modificamos es el Transformer Encoder. Además de la configuración base, experimentamos con otras dos configuraciones del Encoder variando los hiperparámetros d_{model}, d_{ff} y h como se detalla en la Tabla 4.2. Para cada configuración, repetimos todo el proceso de entrenamiento. Fijamos el número de capas en 6, que es el valor máximo que nos permite realizar todos los experimentos sin exceder la memoria disponible.

Además de los componentes de CLAVE, otro factor que puede influir significativamente en su rendimiento es la longitud de la entrada. Para evaluar su impacto, inicialmente preentrenamos el modelo con una longitud de entrada de 512 tokens. Luego, repetimos el proceso de *fine-tuning* con longitudes de entrada de 256, 512, 1.024 y 2.048 tokens. Solo cambiamos la longitud de entrada durante el *fine-tuning* para evitar repetir el costoso proceso de preentrenamiento.

4.5 Entrenamiento y búsqueda de hiperparámetros

Para preentrenar CLAVE, utilizamos el optimizador AdamW [35] con sus parámetros predeterminados de $\beta_1 = 0{,}9$ y $\beta_2 = 0{,}999$, y una tasa de aprendizaje de 10^{-4}. Incrementamos desde 0 la tasa de aprendizaje durante 100.000 pasos y luego la reducimos linealmente a 0 a lo largo de 1.500.000 pasos. Seleccionamos estos valores observando la evolución de la pérdida de validación durante un entrenamiento previo. CLAVE se preentrena con un tamaño de *batch* de 32, el valor más grande posible para completar todos los experimentos con la memoria disponible.

Para el *fine-tuning* de CLAVE, realizamos una búsqueda en cuadrícula para

Hiperparámetro	Rango	Resultado
Dimensión de la última capa	[d_{model}, $d_{model}/2$, $d_{model} \times 2$]	d_{model}
Prob. de *dropout* de la última capa	[0, 0,2]	0
Tasa de aprendizaje	[5×10^{-5}, 3×10^{-5}, 2×10^{-5}]	3×10^{-5}
Optimizador	[Adam [30], AdamW [35]]	AdamW

Tabla 4.3: Los hiperparámetros de CLAVE optimizados utilizando búsqueda en cuadrícula. Ambos optimizadores son evaluados con sus parámetros predeterminados de $\beta_1 = 0{,}9$ y $\beta_2 = 0{,}999$.

encontrar los hiperparámetros, detallados en la Tabla 4.3, que logran el mejor AUC en el conjunto de validación. En este caso, incrementamos desde 0 la tasa de aprendizaje durante 40.000 pasos y luego la reducimos linealmente a 0 a lo largo de 500.000 pasos. Estos valores se seleccionan de la misma manera que en el paso de preentrenamiento. Realizamos el *fine-tuning* con un tamaño de *batch* de 32 cuando utilizamos la pérdida contrastiva y la pérdida de tripleta como objetivo de entrenamiento, y con los tamaños descritos en la Sección 4.4 cuando utilizamos NT-Xent. Configuramos las funciones de pérdida contrastiva y de tripleta con su margen predeterminado de 1, y NT-Xent con una temperatura de $\tau = 0{,}005$ como sugiere [50].

Para tanto el preentrenamiento como el *fine-tuning*, recortamos la norma del gradiente a 1,0 para evitar la explosión de gradientes. Además, aplicamos *dropout* con una probabilidad del 0,1 al Transformer Encoder, siguiendo el enfoque de BERT [17].

Todos los modelos fueron entrenados utilizando el *framework* de aprendizaje profundo PyTorch. El entrenamiento se realizó en hardware compuesto de una NVIDIA RTX 3070 con 8 GB de VRAM y la cuarta parte de una NVIDIA A100 compartida con 80 GB de VRAM (20 GB disponibles).

Capítulo 5

Evaluación

En este capítulo, comparamos el rendimiento de CLAVE con los sistemas de verificación de autoría de código fuente más avanzados (Sección 4.3). El siguiente capítulo presenta una serie de experimentos (detallados en la Sección 4.4) para analizar cómo diferentes factores afectan al rendimiento de CLAVE.

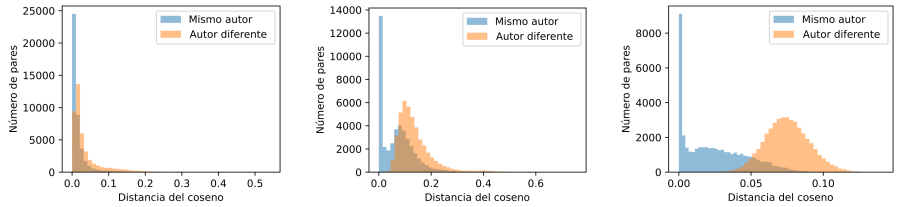
La Tabla 5.1 muestra las métricas de evaluación obtenidas por CLAVE con la siguiente configuración: tokenizador SentencePiece para Python, función de pérdida NT-Xent ($N = 16$) y la configuración base de la Tabla 4.2. Comparamos esta configuración de CLAVE con los sistemas de referencia, utilizando los pares del conjunto de evaluación. Como se describe en la Sección 4.3, todos los sistemas excepto FDR se evalúan con una longitud de entrada de 512 tokens.

Nuestro sistema, CLAVE, obtiene los mejores resultados en todas las métricas con diferencias estadísticamente significativas, logrando un AUC de 0,9782. A continuación está el sistema propuesto por White *et al.*, con un AUC de 0,9696. SCS-Gan y FDR siguen con valores de AUC de 0,9594 y 0,8112, respectivamente. CodeBERT obtiene las métricas más bajas en general, excepto en exhaustividad, con un AUC de 0,6986.

El Transformer Encoder preentrenado de CLAVE (Sección 6.4) y su tokenizador personalizado (Sección 6.1) hacen que nuestro modelo supere a los sistemas de referencia. El sistema propuesto por White *et al.* alcanza resultados

Sistema	AUC	Exactitud	F_1-score	Precisión	Exhaustividad
CodeBERT [20]	0,6986	0,6323	0,7230	0,5800	0,9598
FDR [25]	0,8112	0,7432	0,7836	0,6770	0,9301
SCS-Gan [37]	0,9594	0,8992	0,9040	0,8633	0,9488
White *et al.* [50]	0,9696	0,9129	0,9170	0,8761	0,9620
CLAVE	**0,9782**	**0,9294**	**0,9324**	**0,8945**	**0,9736**

Tabla 5.1: AUC, exactitud, F_1-score, precisión y exhaustividad de CLAVE (configuración base, tokenizador SentencePiece para Python y longitud de entrada de 512) en comparación con los sistemas de referencia (Sección 4.3) para el conjunto de pares de evaluación. Los mejores resultados, sin solapamiento de intervalos de confianza del 95 %, se destacan en negrita. Todos los intervalos de confianza del 95 % están por debajo del 0,02 %.



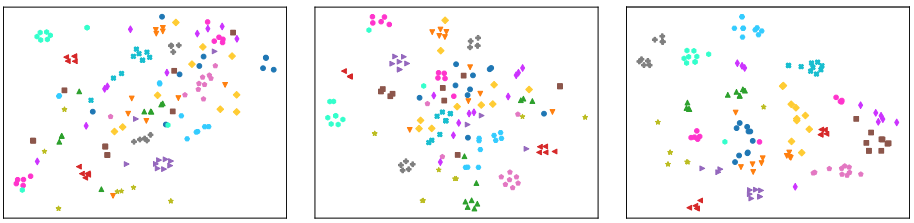
(a) CLAVE inicializado aleatoriamente (b) CLAVE preentrenado (sin *fine-tuning*) (c) CLAVE preentrenado y con *fine-tuning*

Figura 5.1: La distribución de las distancias del coseno entre los pares de evaluación estimadas por (a) CLAVE inicializado aleatoriamente, (b) CLAVE preentrenado sin *fine-tuning* y (c) CLAVE entrenado (configuración base, tokenizador SentencePiece para Python y longitud de entrada 512).

significativamente mejores que SCS-Gan, indicando que, a pesar de compartir la misma arquitectura de LSTM bidireccional, un objetivo de aprendizaje contrastivo más sofisticado (SimCLR) permite un mejor aprendizaje de representaciones estilométricas que el uso de una Generative Adversarial Network. Este enfoque también ha ayudado a CLAVE a mejorar su rendimiento (Sección 6.2). FDR tiene un rendimiento pobre en comparación con los demás sistemas, debido a su tamaño de modelo más pequeño y a la falta de entrenamiento específico para verificación. Finalmente, CodeBERT, que no ha sido entrenado para ninguna tarea relacionada con la autoría, obtiene las métricas de rendimiento más bajas.

La Figura 5.1 muestra la distribución de las distancias entre los pares de evaluación antes del entrenamiento de CLAVE, después del preentrenamiento sin *fine-tuning*, y después del preentrenamiento y *fine-tuning*. Podemos observar cómo el solapamiento entre las distribuciones de pares del mismo autor y de diferentes autores se reduce notablemente después del preentrenamiento y del *fine-tuning*. El gran número de pares del mismo autor con distancias muy cercanas a 0, incluso cuando el modelo no está preentrenado, se puede explicar por la existencia de problemas compuestos de varias partes en las competiciones de programación. En estos casos, muchos autores reutilizan su propio código para resolver diferentes partes del mismo problema, resultando en pares de código fuente muy similares.

La Figura 5.2 ilustra cómo CLAVE también aprende a agrupar fragmentos de código fuente escritos por el mismo autor en el espacio de *embedding*, proporcionando resultados significativamente mejores tanto después del preentrenamiento como después del *fine-tuning*. Los *embeddings* son los vectores mostrados en la Figura 3.1 como los "*embeddings* de código fuente"generados por CLAVE. En la Figura 5.2, esos *embeddings* se reducen a 2 dimensiones usando t-SNE. Los fragmentos se extrajeron aleatoriamente de la partición de evaluación del conjunto de datos de *fine-tuning*, donde se seleccionaron 15 autores y se eligieron 10 archivos de código fuente diferentes para cada autor.



(a) CLAVE inicializado aleatoriamente

(b) CLAVE preentrenado (sin *fine-tuning*)

(c) CLAVE preentrenado y con *fine-tuning*

Figura 5.2: *Embeddings* generados a partir de 150 fragmentos de código diferentes por (a) CLAVE inicializado aleatoriamente, (b) CLAVE preentrenado sin *fine-tuning* y (c) CLAVE entrenado (configuración base, tokenizador SentencePiece para Python y longitud de entrada 512). Los *embeddings* son reducidos a 2 dimensiones utilizando t-SNE. Los colores y marcadores indican el autor de cada fragmento de código.

Capítulo 6

Discusión

En este capítulo, realizamos una serie de experimentos (detallados en la Sección 4.4) para analizar cómo diferentes factores afectan al rendimiento de CLAVE. Estos factores incluyen la elección del tokenizador, la función de pérdida para el *fine-tuning*, el proceso de preentrenamiento, el tamaño del Transformer Encoder y la longitud de la entrada. Todos los experimentos analizan modificaciones de los ajustes evaluados en la Sección 5: tokenizador SentencePiece para Python, longitud de entrada de 512, función de pérdida NT-Xent ($N = 16$) y configuración base (Tabla 4.2).

6.1 Tokenizador

La Tabla 6.1 compara el rendimiento de CLAVE en el conjunto de pares de evaluación con cada uno de los tres tokenizadores presentados en la Sección 3.2, junto con un tokenizador SentencePiece sin personalizar. SentencePiece para Python logra el mayor AUC (0,9782), mientras que el tokenizador SentencePiece sin personalizar obtiene la mejor exactitud (0,9313) y F_1-score (0,9336). El tokenizador estilométrico obtiene el AUC más bajo (0,9718), pero cuando se combina con SentencePiece, el AUC aumenta a 0,9748.

El tokenizador estilométrico ignora el contenido de identificadores, cadenas, literales numéricos y comentarios. Aunque produce tokens especiales para algunas características estilométricas seleccionadas manualmente de estos elementos

Tokenizador	AUC	Exactitud	F_1-score	Precisión	Exhaustividad
Estilométrico	0,9718	0,9197	0,9227	0,8888	0,9593
Estilo. con SentencePiece	0,9748	0,9246	0,9277	0,8912	0,9673
SentencePiece	0,9774	**0,9313**	**0,9336**	**0,9030**	0,9665
SentencePiece para Python	**0,9782**	0,9294	0,9324	0,8945	**0,9736**

Tabla 6.1: AUC, exactitud, F_1-score, precisión, y exhaustividad obtenidos por CLAVE (configuración base y longitud de entrada 512) en el conjunto de pares de evaluación con diferentes tokenizadores. Los mejores resultados, sin solapamiento de intervalos de confianza del 95 %, se destacan en negrita. Todos los intervalos de confianza del 95 % están por debajo del 0,02 %.

Función de pérdida	N	AUC	Exactitud	F_1-score	Precisión	Exhaustividad
Pérdida contrastiva	32	0,9513	0,8846	0,8881	0,8616	0,9163
Pérdida de tripleta	32	0,9765	0,9267	0,9296	0,8939	0,9684
NT-Xent	32	0,9761	0,9291	0,9313	**0,9038**	0,9605
	16	**0,9782**	**0,9294**	**0,9324**	0,8945	**0,9736**

Tabla 6.2: AUC, exactitud, F_1-score, precisión, y exhaustividad obtenidos por CLAVE (configuración base, tokenizador SentencePiece para Python y longitud de entrada 512) en el conjunto de pares de evaluación para diferentes funciones de pérdida y tamaños de *batch* (N): pérdida contrastiva ($margen = 1$), pérdida de tripleta ($margen = 1$) y NT-Xent ($\tau = 0{,}005$). Los mejores resultados, sin solapamiento de intervalos de confianza del 95 %, se destacan en negrita. Todos los intervalos de confianza del 95 % están por debajo del 0,02 %.

(Tabla 3.1), SentencePiece permite al modelo extraer características adicionales del código fuente, lo que ayuda a mejorar su rendimiento para la verificación de autoría. Esta mejora se hace evidente cuando se incluyen los contenidos de los identificadores tokenizados por SentencePiece en el tokenizador estilométrico, lo que provoca un aumento en todas las métricas de evaluación.

Por otro lado, las personalizaciones incluidas en el tokenizador SentencePiece para Python (Sección 3.2.1) permiten al modelo alcanzar una puntuación de AUC significativamente más alta. Sin embargo, es el tokenizador SentencePiece sin personalizar el que logra la mejor exactitud y F_1-score. Atribuimos esto al umbral γ seleccionado para nuestro conjunto de pares de evaluación, que permite al modelo obtener resultados significativamente más altos para estas métricas. Como se mencionó, el AUC proporciona una representación más robusta de la capacidad de generalización del modelo, ya que no depende del γ seleccionado.

6.2 Función de pérdida para el *fine-tuning*

La Tabla 6.2 presenta el impacto de diferentes funciones de pérdida en el rendimiento de CLAVE, evaluado en el conjunto de pares de evaluación después del *fine-tuning*. Utilizamos las tres funciones de pérdida detalladas en la Sección 4.4, con dos tamaños de *batch* diferentes para NT-Xent. Los mejores resultados para todas las métricas excepto precisión se obtienen con NT-Xent y un tamaño de lote de 16 (AUC de 0,9782). La misma función de pérdida con $N = 32$ logra la mayor precisión, pero, en comparación con NT-Xent con $N = 16$, la disminución en exhaustividad es mayor que la ganancia en precisión, como muestra la reducción significativa en el F_1-score.

La función de pérdida NT-Xent logra un rendimiento superior al permitir que el modelo aprenda más de los negativos difíciles. Un negativo difícil es un par de fragmentos de código fuente escritos por diferentes autores pero cuyos *embeddings* están más cerca que los de los fragmentos del mismo autor. NT-Xent logra esto comparando cada fragmento de código fuente con un total de $2 \times (N - 1)$ instancias negativas en un lote de tamaño N. En cambio, la pérdida de tripleta compara cada fragmento con solo 1 instancia negativa, y la pérdida contrastiva lo compara con 1 o 0 instancias negativas. Además, NT-Xent pondera las instancias negativas según su dificultad relativa [14], priorizando los negativos difíciles en

Configuración	AUC	Exactitud	F_1-score	Precisión	Exhaustividad
Pequeña	0,9762	0,9279	0,9302	0,9011	0,9613
Base	**0,9782**	**0,9294**	**0,9324**	0,8945	**0,9736**
Grande	0,9762	0,9288	0,9310	**0,9028**	0,9611

Tabla 6.3: AUC, exactitud, F_1-score, precisión, y exhaustividad obtenidos por CLAVE (tokenizador SentencePiece para Python y tamaño de entrada 512) en el conjunto de pares de evaluación para cada una de las tres configuraciones. Los mejores resultados, sin solapamiento de intervalos de confianza del 95 %, se destacan en negrita. Todos los intervalos de confianza del 95 % están por debajo del 0,02 %.

el cálculo de la pérdida.

Contrario a los hallazgos de Chen *et al.* [14], quienes observaron mejoras de rendimiento con tamaños de *batch* más grandes, nuestros resultados indican que *batches* más pequeños pueden producir un mejor rendimiento. Atribuimos esto al enfoque del *framework* SimCLR, que aplica aleatoriamente *data augmentation* para cada instancia dentro de un *batch*. En cambio, el *fine-tuning* de CLAVE utiliza siempre los mismos fragmentos de código fuente sin modificar. Como resultado, un tamaño de *batch* más grande significa que CLAVE aprende más frecuentemente de los mismos negativos difíciles, lo que potencialmente conduce a un sobreajuste más rápido.

6.3 Tamaño del modelo

Las diferentes configuraciones del Transformer Encoder detalladas en la Tabla 4.2 pueden tener un impacto significativo en el rendimiento de CLAVE. La Tabla 6.3 muestra las métricas de evaluación obtenidas utilizando el conjunto de pares de evaluación para cada una de estas configuraciones. La configuración base logra el AUC más alto (0,9782), mientras que tanto las configuraciones pequeña como grande obtienen un AUC de 0,9762.

En comparación con la configuración pequeña, el mayor número de parámetros y la mayor dimensionalidad de los *embeddings* permiten que la configuración base de CLAVE aprenda representaciones estilométricas que logran un rendimiento superior en la verificación de autoría. Por otro lado, la disminución observada en el rendimiento con la configuración grande podría indicar que la cantidad de datos disponibles en el conjunto de *fine-tuning* es insuficiente para aprovechar completamente el mayor tamaño del modelo.

6.4 Preentrenamiento

La Tabla 6.4 presenta la evaluación de CLAVE con y sin preentrenamiento, siguiendo la metodología descrita en la Sección 4.4. Podemos ver cómo las medidas de AUC, exactitud y F_1-score se incrementan, respectivamente, en un 1,84 %, 3,28 % y 3,07 % con el preentrenamiento.

Estos resultados muestran que el preentrenamiento de CLAVE con modelado de lenguaje enmascarado (MLM, Sección 3.3) puede mejorar significativamente

Preentrenamiento	AUC	Exactitud	F_1-score	Precisión	Exhaustividad
No	0,9605	0,8999	0,9046	0,8642	0,9490
Sí	**0,9782**	**0,9294**	**0,9324**	**0,8945**	**0,9736**

Tabla 6.4: AUC, exactitud, F_1-score, precisión, y exhaustividad obtenidos por CLAVE (configuración base, tokenizador SentencePiece para Python y longitud de entrada 512) en el conjunto de pares de evaluación con y sin preentrenamiento. Los mejores resultados, sin solapamiento de intervalos de confianza del 95 %, se destacan en negrita. Todos los intervalos de confianza del 95 % están por debajo del 0,02 %.

Longitud de entrada	AUC	Exactitud	F_1-score	Precisión	Exhaustividad
256	0,9720	0,9169	0,9203	0,8843	0,9594
512	0,9782	0,9294	0,9324	0,8945	**0,9736**
1024	**0,9792**	**0,9340**	**0,9359**	**0,9098**	0,9634
2048	0,9770	0,9294	0,9315	0,9045	0,9602

Tabla 6.5: AUC, exactitud, F_1-score, precisión, y exhaustividad obtenidos por CLAVE (configuración base y tokenizador SentencePiece para Python) en el conjunto de pares de evaluación con diferentes longitudes de entrada. Los mejores resultados, sin solapamiento de intervalos de confianza del 95 %, se destacan en negrita. Todos los intervalos de confianza del 95 % están por debajo del 0,02 %.

su rendimiento para la verificación de autoría. Además, las Figuras 5.1 y 5.2 muestran que, después del preentrenamiento, el modelo es capaz de generar *embeddings* más cercanos para el código fuente escrito por el mismo autor y más alejados para diferentes autores. Esto apoya nuestra hipótesis de que el MLM facilita el aprendizaje para distinguir diferentes estilos de codificación, permitiéndonos aprovechar grandes cantidades de código fuente para el entrenamiento, incluso cuando se desconoce su autor.

6.5 Longitud de la entrada

La Tabla 6.5 muestra las métricas de evaluación obtenidas por CLAVE en el conjunto de evaluación con las diferentes longitudes de entrada mencionadas en la Sección 4.4. El AUC aumenta a medida que aumenta la longitud de la entrada, alcanzando un máximo de 0,9792 para entradas con 1.024 tokens. Este es el valor de AUC más alto logrado por CLAVE entre todas las configuraciones evaluadas. Para una longitud de entrada de 2.048 tokens, el AUC obtenido (0,9770) es significativamente menor que para 512 tokens (0,9782), pero mayor que para 256 tokens (0,9720).

Hasta 1.024 tokens, los aumentos en la longitud de entrada permiten que el modelo genere *embeddings* que capturan mejor las características estilométricas del código fuente. Sin embargo, como se observa en la Figura 6.1, la mayoría de los archivos de código fuente en el conjunto de datos de *fine-tuning* se transforman en vectores de 1024 o menos tokens después de ser procesados por el

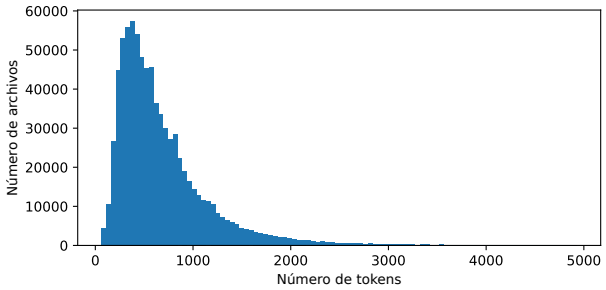


Figura 6.1: Distribución del número de tokens en el conjunto de datos de *fine-tuning* tras eliminar valores atípicos. Los archivos de código fuente del conjunto de datos son tokenizados utilizando SentencePiece para Python.

tokenizador (SentencePiece para Python). Esta falta de instancias más largas podría dificultar que el modelo aprenda representaciones efectivas a partir de los datos de entrenamiento cuando la longitud de entrada se establece en 2.048 tokens. También es importante destacar que, como se detalla en la Sección 4.4, la longitud de entrada para el preentrenamiento de CLAVE se establece en 512 tokens en todos los casos, lo que puede obstaculizar aún más el aprendizaje de representaciones para entradas más largas.

Otro enfoque para aumentar el contexto sin extender la longitud de entrada es utilizar CLAVE con diferentes fragmentos de código extraídos de una misma entrada. Un ejemplo común de este enfoque es el método de *soft voting* [6]. Esta técnica consiste en dividir la entrada, aplicar el modelo a cada fragmento por separado y luego agregar las salidas utilizando una media ponderada. Como se muestra en la Figura 6.1, la mayoría de los archivos de entrada contienen 1.024 o menos tokens. Por lo tanto, decidimos no evaluar este enfoque ya que la mayoría de los archivos caben dentro de una sola entrada.

6.6 CLAVE como clasificador

Aunque este trabajo se centra únicamente en la verificación de autoría, las representaciones estilométricas que genera CLAVE también podrían usarse para la atribución de autoría. Como se mencionó, la atribución de autoría es la tarea de identificar al autor de un documento anónimo dado un conjunto de autores conocidos. Siguiendo el enfoque de White *et al.* [50], se podría entrenar un clasificador para predecir, a partir del *embedding* generado por CLAVE, el autor de un fragmento de código fuente. Alternativamente, se podría emplear el algoritmo de los k vecinos más cercanos (kNN) sin entrenamiento adicional. kNN asignaría cada fragmento de código fuente anónimo al autor más común entre sus k *embedding* más cercanos.

Capítulo 7

Conclusiones

Hemos mostrado cómo se pueden superar los sistemas existentes para verificación de autoría de código fuente utilizando aprendizaje contrastivo con la función de pérdida NT-Xent para realizar el *fine-tuning* de un Transformer Encoder preentrenado. Nuestro sistema, llamado CLAVE, genera *embeddings* que representan las características estilométricas del código fuente, acercando los *embeddings* del código escrito por el mismo autor y alejándolos para diferentes autores. Cuando se evalúa para la verificación de autoría de código fuente utilizando archivos Python extraídos de las competiciones de programación Google Code Jam y Kick Start, CLAVE logra un AUC de 0,9782 y un F_1-score de 0,9324, superando significativamente a los sistemas existentes. También analizamos el impacto de los componentes más relevantes de CLAVE en su rendimiento, específicamente el tokenizador, la función de pérdida para el *fine-tuning*, el preentrenamiento, la configuración del Transformer Encoder y la longitud de entrada.

Como trabajo futuro, planeamos investigar si las estructuras de árbol y grafo del código fuente podrían proporcionar algún beneficio para la verificación de autoría [39]. Los datos sintácticos y semánticos podrían representarse en forma de grafos para obtener información detallada sobre el código fuente utilizando Redes Neuronales de Grafos [4].

El código fuente, los conjuntos de datos descritos en la Sección 4.1 y los pesos serializados de CLAVE listos para ser utilizados en PyTorch están disponibles para su descarga en `https://reflection.uniovi.es/bigcode/download/2024/clave`.

Capítulo 8

Dirección y gestión del TFM

Este capítulo detalla cómo se ha llevado a cabo la dirección y la gestión del trabajo. La Sección 8.1 presenta la planificación inicial del proyecto. La ejecución del proyecto se describe en la Sección 8.2. Finalmente, la Sección 8.3 contiene los documentos relativos al cierre del proyecto.

8.1 Planificación del proyecto

En esta sección se identifican los interesados del proyecto, se desglosa la organización del equipo del proyecto (OBS), los productos del proyecto (PBS) y la planificación inicial (WBS). También incluye el plan de gestión de riesgos y el presupuesto inicial.

8.1.1 Identificación de interesados

Después de un proceso exhaustivo de identificación de interesados, se han seleccionado los siguientes:

- Equipo del proyecto: David Álvarez Fidalgo y Francisco Ortín Soler.
- Comisión académica del máster.
- Tribunal de evaluación del TFM.
- Comunidad científica.

8.1.2 OBS

La organización del equipo del proyecto es sencilla, ya que está compuesto por solo dos miembros: David Álvarez Fidalgo, con rol de estudiante, y Francisco Ortín Soler, con rol de director del TFM.

8.1.3 PBS

La Figura 8.1 muestra en detalle la estructura de desglose de productos (PBS) del proyecto

8.1.4 WBS

La Tabla 8.1 contiene la estructura de desglose de trabajo (WBS) del proyecto. La Figura 8.2 muestra un diagrama de Gantt con la planificación inicial de las tareas contenidas en el WBS. El orden en el que aparecen las tareas en el diagrama de Gantt es el mismo que en el WBS. El diagrama de Gantt, además de las tareas, también incluye los hitos del proyecto.

Cabe destacar que, en el cronograma, las tareas anteriores a febrero de 2024 se realizar como parte de la asignatura Dirección y Gestión de Proyectos Web.

ID	Tarea	Nivel	Descripción
1	Proyecto	1	Desarrollo del proyecto
2	Gestión del proyecto	2	Gestión del proyecto
3	Elaboración del acta de constitución preliminar	3	Elaboración del acta de constitución preliminar
4	Elaboración de la planificación preliminar	3	Elaboración de la planificación preliminar
5	Elaboración del plan de riesgos preliminar	3	Elaboración del plan de riesgos preliminar
6	Elaboración del presupuesto preliminar	3	Elaboración del presupuesto preliminar
7	Elaboración de la planificación final	3	Elaboración de la planificación final
8	Elaboración del plan de riesgos final	3	Elaboración del plan de riesgos final
9	Elaboración del presupuesto final	3	Elaboración del presupuesto final
10	Elaboración del acta de constitución final	3	Elaboración del acta de constitución final
11	Reuniones mensuales	3	Reuniones mensuales con el director
12	Reuniones mensuales 1	4	Primera reunión mensual con el director
13	Reuniones mensuales 2	4	Segunda reunión mensual con el director
14	Cierre del proyecto	3	Todas las tareas que conlleva el cierre del proyecto
15	Revisión de trabajo relacionado	2	Revisión de trabajo relacionado
16	Procesamiento de lenguaje natural	3	Revisión de trabajo relacionado sobre procesamiento de lenguaje natural
17	Verificación de autoría	3	Revisión de trabajo relacionado sobre verificación de autoría

18	Verificación de autoría de código fuente	3 Revisión de trabajo relacionado sobre verificación de autoría de código fuente
19	Diseño de la arquitectura del modelo	2 Diseño de la arquitectura del modelo
20	Selección de tokenizador	3 Selección de tokenizador
21	Diseño de la arquitectura del modelo base	3 Diseño de la arquitectura del modelo base
22	Selección del objetivo de preentrenamiento	3 Selección del objetivo de preentrenamiento
23	Diseño de la arquitectura del modelo final	3 Diseño de la arquitectura del modelo final
24	Selección del objetivo de entrenamiento	3 Selección del objetivo de entrenamiento
25	Selección de conjuntos de datos	2 Selección de conjuntos de datos
26	Selección del conjunto de datos de preentrenamiento	3 Selección del conjunto de datos de preentrenamiento
27	Estudio de conjuntos de datos existentes	4 Estudio de conjuntos de datos existentes
28	Elaboración del conjunto de datos	4 Elaboración del conjunto de datos de preentrenamiento
29	Selección del conjunto de datos de entrenamiento	3 Selección del conjunto de datos de entrenamiento
30	Estudio de conjuntos de datos existentes	4 Estudio de conjuntos de datos existentes
31	Elaboración del conjunto de datos	4 Elaboración del conjunto de datos de entrenamiento
32	Obtención del modelo preentrenado	2 Obtención del modelo preentrenado
33	Implementación del modelo base	3 Implementación del modelo base
34	Selección de hiperparámetros	3 Selección de hiperparámetros del modelo base
35	preentrenamiento del modelo	3 preentrenamiento del modelo
36	Obtención del modelo entrenado	2 Obtención del modelo entrenado
37	Implementación del modelo final	3 Implementación del modelo final
38	Selección de hiperparámetros	3 Selección de hiperparámetros del modelo final
39	Entrenamiento del modelo	3 Entrenamiento del modelo
40	Evaluación	2 Evaluación del modelo
41	Medición del rendimiento del modelo	3 Medición del rendimiento del modelo
42	Medición de la eficiencia computacional del modelo	3 Medición de la eficiencia computacional del modelo
43	Redacción de la memoria	2 Redacción de la memoria
44	Redacción de fijación de objetivos	3 Redacción de fijación de objetivos
45	Redacción de trabajo relacionado	3 Redacción de trabajo relacionado
46	Redacción de descripción del sistema	3 Redacción de descripción del sistema

47	Redacción de metodología	3	Redacción de metodología
48	Redacción de resultados obtenidos	3	Redacción de resultados obtenidos
49	Redacción de conclusiones	3	Redacción de conclusiones
50	Redacción de introducción	3	Redacción de introducción
51	Redacción de resumen	3	Redacción de resumen
52	Redacción de apéndices	3	Redacción de apéndices
53	Revisión de la memoria	3	Revisión de la memoria

Tabla 8.1: Estructura de desglose de trabajo (WBS) del proyecto.

8.1.5 Riesgos

El Apéndice B contiene una copia del plan de gestión de riesgos del proyecto y el Apéndice C incluye una copia del registro de riesgos. Durante la elaboración del plan de gestión de riesgos se identificaron estos 15 riesgos:

- Cantidad de datos.
- Calidad de los datos.
- Recursos computacionales limitados.
- Compartición de los recursos computacionales.
- Falta de experiencia.
- Desarrollo novedoso.
- Complejidad del modelo.
- Problemas de convergencia.
- Sobrecarga de almacenamiento.
- Seguridad y privacidad.
- Mantenimiento del código.
- Sesgo en los datos.
- Problemas de derechos de autor.
- Escalabilidad.
- Dependencia de software.

8.1.6 Presupuesto inicial

El presupuesto inicial de costes del proyecto se muestra en la Tabla 8.2. El coste total del proyecto es de 23.754,10 € y está dividido en estas cuatro partidas:

- Gestión del proyecto: 5.549,14 €.
- Diseño de investigación: 9.433,53 €.

Partida	Item	Partida	Importe	Total
1		**Gestión del proyecto**		5.549,14 €
2		**Diseño de investigación**		9.433,53 €
	1	Revisión de trabajo relacionado	3.237,00 €	
	2	Diseño de la arquitectura del modelo	3.237,00 €	
	3	Selección del conjunto de datos	2.959,54 €	
3		**Entrenamiento y evaluación del modelo**		2.459,29 €
	1	Obtención del modelo preentrenado	807,40 €	
	2	Obtención del modelo entrenado	542,06 €	
	3	Evaluación	1.109,83 €	
4		**Redacción de la memoria**		6.312,14 €
			TOTAL	23.754,10 €

Tabla 8.2: Presupuesto inicial de costes del proyecto.

- Entrenamiento y evaluación del modelo: 2.459,29 €.
- Redacción de la memoria: 6.312,14 €.

8.2 Ejecución del proyecto

8.2.1 Plan seguimiento de la planificación

Para planificar el seguimiento de la planificación, se estableció como línea base la planificación inicial presentada en la Sección 8.1.4. A medida que se fueron completando las tareas, se analizó si se seguía la planificación establecida en la línea base, tomando medidas en caso de que se produjese algún desvío. Las medidas tomadas siempre consistieron en aumentar la asignación de recursos al proyecto.

Aproximadamente a la mitad del proyecto, se conoció que el equipo no podría disponer del servidor con GPU NVIDIA A100 hasta el final del periodo lectivo. Este hecho ocasionó que se estableciera una nueva línea base, retrasando las tareas asignadas a dicho recurso hasta el final de las clases. El cronograma asociado a esta nueva línea base se muestra en la Figura 8.3. Como resultado, la fecha estimada de finalización del proyecto también se vio retrasada hasta el día 21 de junio de 2024. Debido a este incremento en la fecha estimada de finalización, se añadió una nueva reunión mensual a la planificación.

Esta línea base se siguió hasta el cierre del proyecto. En la Sección 8.3.1 se detalla la planificación final del proyecto y las desviaciones que se produjeron respecto de la última línea base.

8.2.2 Bitácora de incidencias del proyecto

- **01/03/2024.** Realizando pruebas con el *framework* de aprendizaje profundo PyTorch nos encontramos con errores de ejecución. Abrimos el siguiente hilo sobre los errores encontrados en el foro de PyTorch: `https://discuss.pytorch.org/t/runtime-error-when-running-inference-on-a-compiled-nn-transformerencoder/198010`. Esta incidencia está relacionada con los riesgos “Dependencia de software” y “Complejidad del modelo”.

- **03/03/2024.** Encontramos la solución a los errores detectados en la última entrada.
- **13/03/2024.** Hoy recibimos la noticia de que no tendremos disponible la GPU NVIDIA A100 hasta que finalice el periodo lectivo. Como consecuencia, retrasamos las tareas que dependen de este recurso hasta el final de las clases. Además, establecemos una nueva línea base para el seguimiento de la planificación (Sección 8.2.1). Esta incidencia está relacionada con el riesgo "Compartición de los recursos computacionales".
- **03/04/2024.** El conjunto de datos que habíamos seleccionado para el entrenamiento (soluciones a problemas de Codeforces) solo contiene archivos de código fuente de unas pocas líneas. Decidimos buscar otro conjunto de datos de mayor calidad. Esta incidencia está relacionada con el riesgo "Calidad de los datos".
- **03/05/2024.** Al reproducir el entrenamiento del modelo de White *et al.*, observamos que la función de pérdida que utiliza, NT-Xent, podría mejorar el rendimiento de nuestro modelo. Realizamos pruebas con un modelo pequeño usando la GPU NVIDIA RTX 3070 y comprobamos que, efectivamente, el rendimiento mejora. Decidimos que cuando volvamos a tener disponibilidad de la GPU NVIDIA A100 realizaremos el entrenamiento del modelo final con NT-Xent. Este cambio no supone ningún retraso en la planificación. Esta incidencia está relacionada con el riesgo "Desarrollo novedoso".

8.2.3 Riesgos

En el Apéndice D se incluyen copias de las hojas de seguimiento de los riesgos más prioritarios identificados en el plan de gestión de riesgos (Apéndice B).

8.3 Cierre del proyecto

8.3.1 Planificación final

La Figura 8.4 muestra un diagrama de Gantt con la planificación final del proyecto. Las principales diferencias respecto de la línea base intermedia (Figura 8.3) son que se eliminó la tarea "Medición del rendimiento del modelo", puesto que se consideró que quedaba fuera de los objetivos del trabajo, y que se produjeron algunos retrasos en las tareas relativas a la redacción de esta memoria. Como consecuencia, la fecha de finalización del proyecto fue el 1 de julio de 2024.

8.3.2 Informe final de riesgos

De los 15 riesgos identificados en el plan de gestión de riesgos (Apéndice B), estos fueron los que se manifestaron a lo largo del proyecto, tal como recoge la bitácora de incidencias (Sección 8.2.2):

- **Dependencia de software.** Un error en el *framework* de aprendizaje profundo PyTorch provocó un retraso de 2 días en la implementación del

Partida	Item	Partida	Importe	Total
1		**Gestión del proyecto**		5.882,00 €
2		**Diseño de investigación**		8.455,37 €
	1	Revisión de trabajo relacionado	3.216,72 €	
	2	Diseño de la arquitectura del modelo	2.573,37 €	
	3	Selección del conjunto de datos	2.665,28 €	
3		**Entrenamiento y evaluación del modelo**		1.892,44 €
	1	Obtención del modelo preentrenado	802,34 €	
	2	Obtención del modelo entrenado	538,66 €	
	3	Evaluación	551,44 €	
4		**Redacción de la memoria**		8.524,30 €
			TOTAL	24.754,10 €

Tabla 8.3: Presupuesto final de costes del proyecto.

modelo, dentro del margen establecido en el plan de contingencia para el riesgo "Complejidad del modelo".

- **Complejidad del modelo.** El error de PyTorch se producía al combinar dos funcionalidades todavía inmaduras que ofrece el *framework*. Estas funcionalidades eran necesarias para poder ejecutar el modelo de forma eficiente.
- **Compartición de los recursos computacionales.** Este fue el riesgo que mayor impacto tuvo en el proyecto, puesto que la no disponibilidad de la GPU NVIDIA A100 entre el 13 de marzo y el 7 de mayo de 2024 provocó un retraso de 28 días en la fecha de finalización del proyecto.
- **Calidad de los datos.** El conjunto de datos de entrenamiento que habíamos seleccionado en un inicio resultó no tener la calidad suficiente para lograr el objetivo de rendimiento del modelo. Sin embargo, identificamos otro conjunto de datos que sí cumplía los requisitos sin que se produjese ningún retraso significativo.
- **Desarrollo novedoso.** Aunque inicialmente se había planteado utilizar una red siamesa, finalmente decidimos entrenar el modelo con la función de pérdida NT-Xent. Este cambio tampoco supuso ningún retraso, puesto que, cuando se tomó la decisión, el equipo del proyecto todavía se encontraba esperando a volver a tener disponible la A100.

8.3.3 Presupuesto final de costes

La Tabla 8.3 presenta el presupuesto final de costes del proyecto. El coste total del proyecto fue de 24.754,10 €, exactamente 1.000 € por encima del presupuesto inicial. Aunque los costes de las partidas 2 y 3 se redujeron al finalizar las tareas antes de lo planificado, los retrasos en la redacción de la memoria causaron el incremento final en el coste del proyecto.

8.3.4 Informe de lecciones aprendidas

Después de la realización del proyecto hemos analizado tanto los aspectos más positivos de su ejecución como las áreas de mejora.

Como aspecto positivo, destacamos la gestión de riesgos que se ha realizado. De los cinco riesgos que se manifestaron 8.3.3, cuatro de ellos se encontraban entre los cinco más prioritarios que habíamos identificado en el plan de gestión de riesgos, y para los que habíamos elaborado planes de contingencia. Estos planes permitieron que, tras haberse manifestado los riesgos, se pudiese continuar el proyecto sin retrasos ni incrementos de coste significativos. La única excepción fue la compartición de los recursos computacionales, que provocó el mayor retraso del proyecto, pero era un riesgo que habíamos decidido aceptar puesto que era necesario para lograr los objetivos propuestos.

Respecto de las áreas de mejora, consideramos que se pudo hacer una mejor estimación del esfuerzo para las tareas relativas a la redacción de la memoria. La estimación inicial resultó ser demasiado optimista, y el incremento de coste que se produjo a raíz de los retrasos en estas tareas fue el mayor de todo el proyecto.

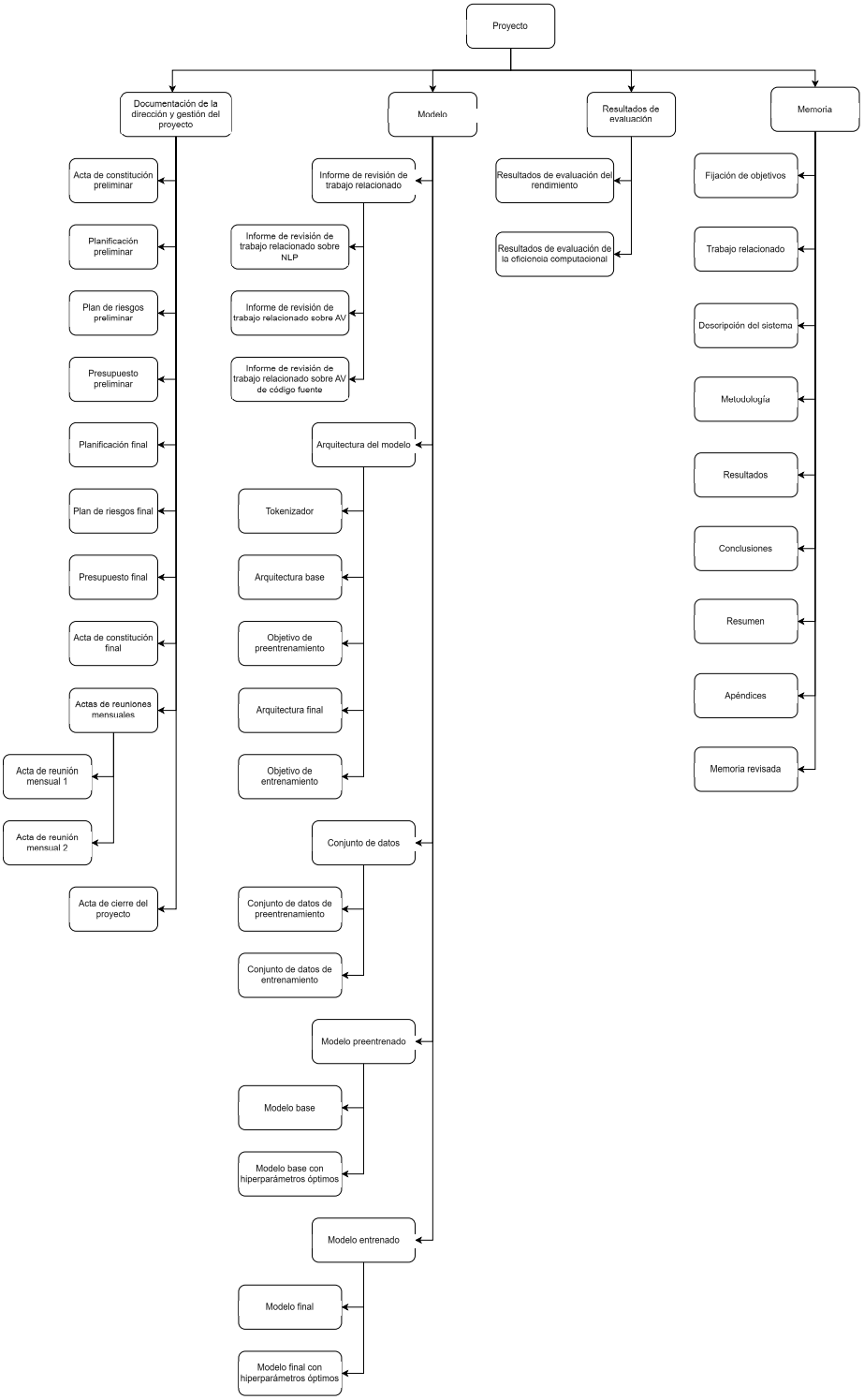


Figura 8.1: La estructura de desglose de productos (PBS) del proyecto.

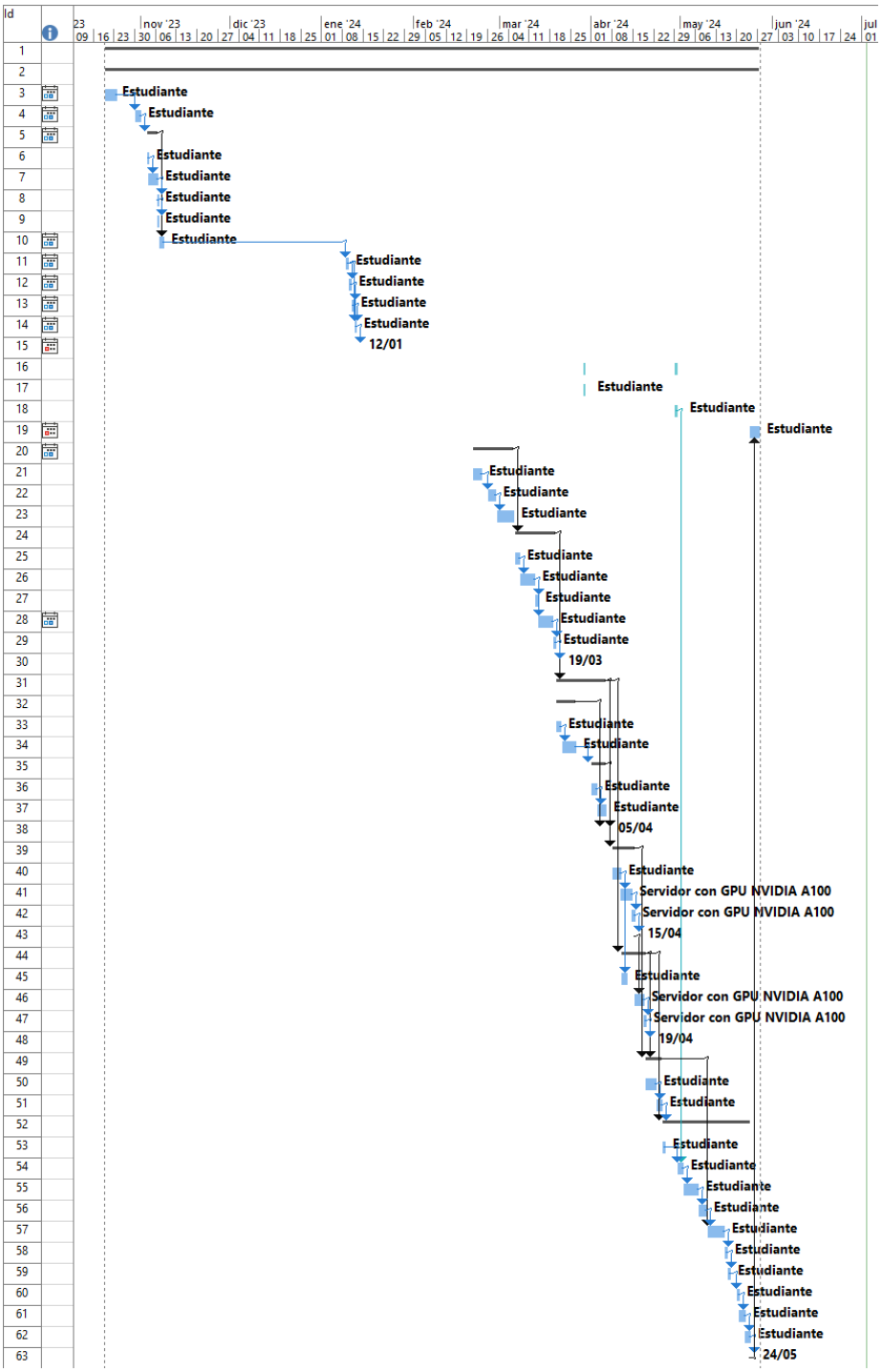


Figura 8.2: Diagrama de Gantt con la planificación inicial del proyecto.

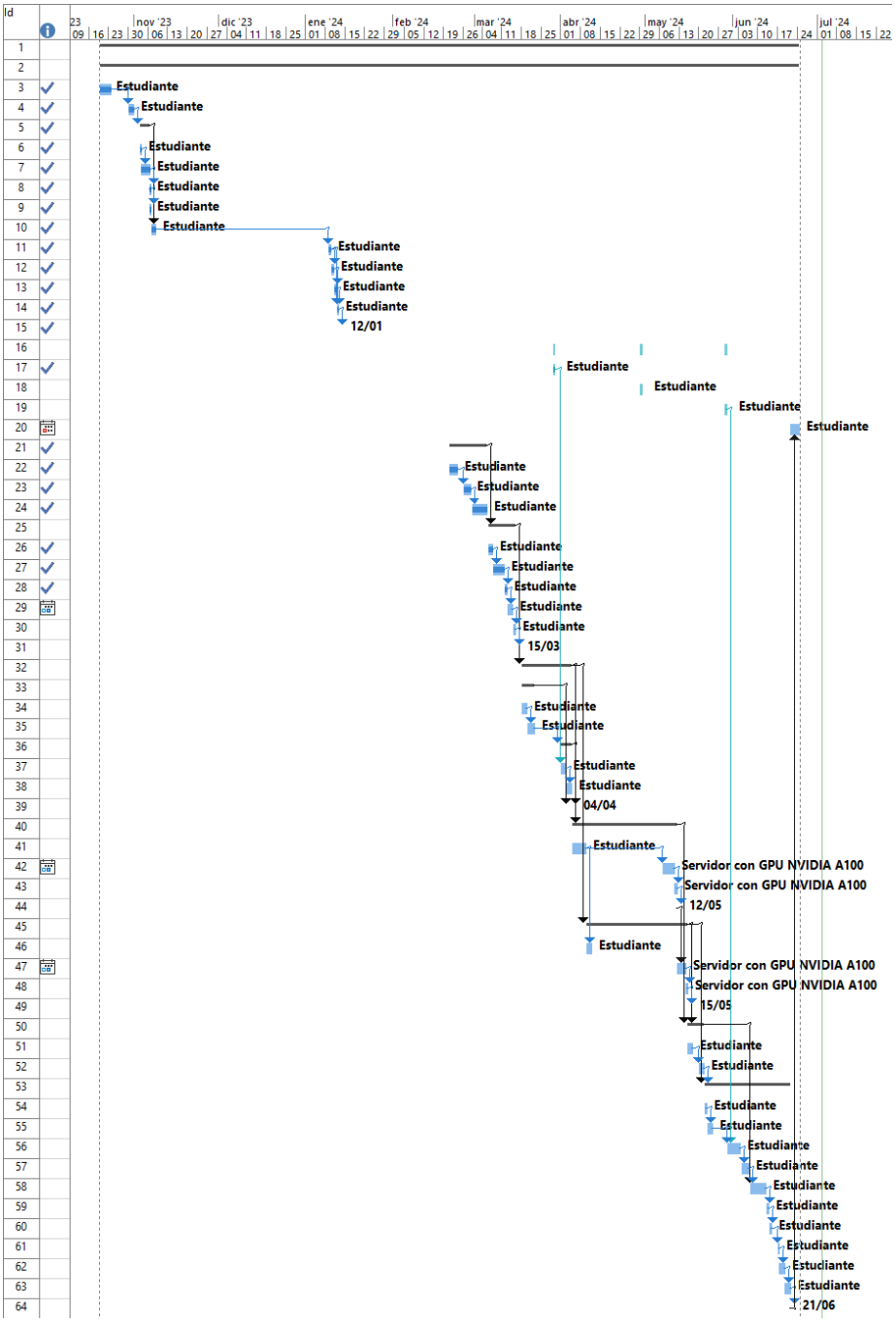


Figura 8.3: Diagrama de Gantt con la línea base establecida a mitad del proyecto.

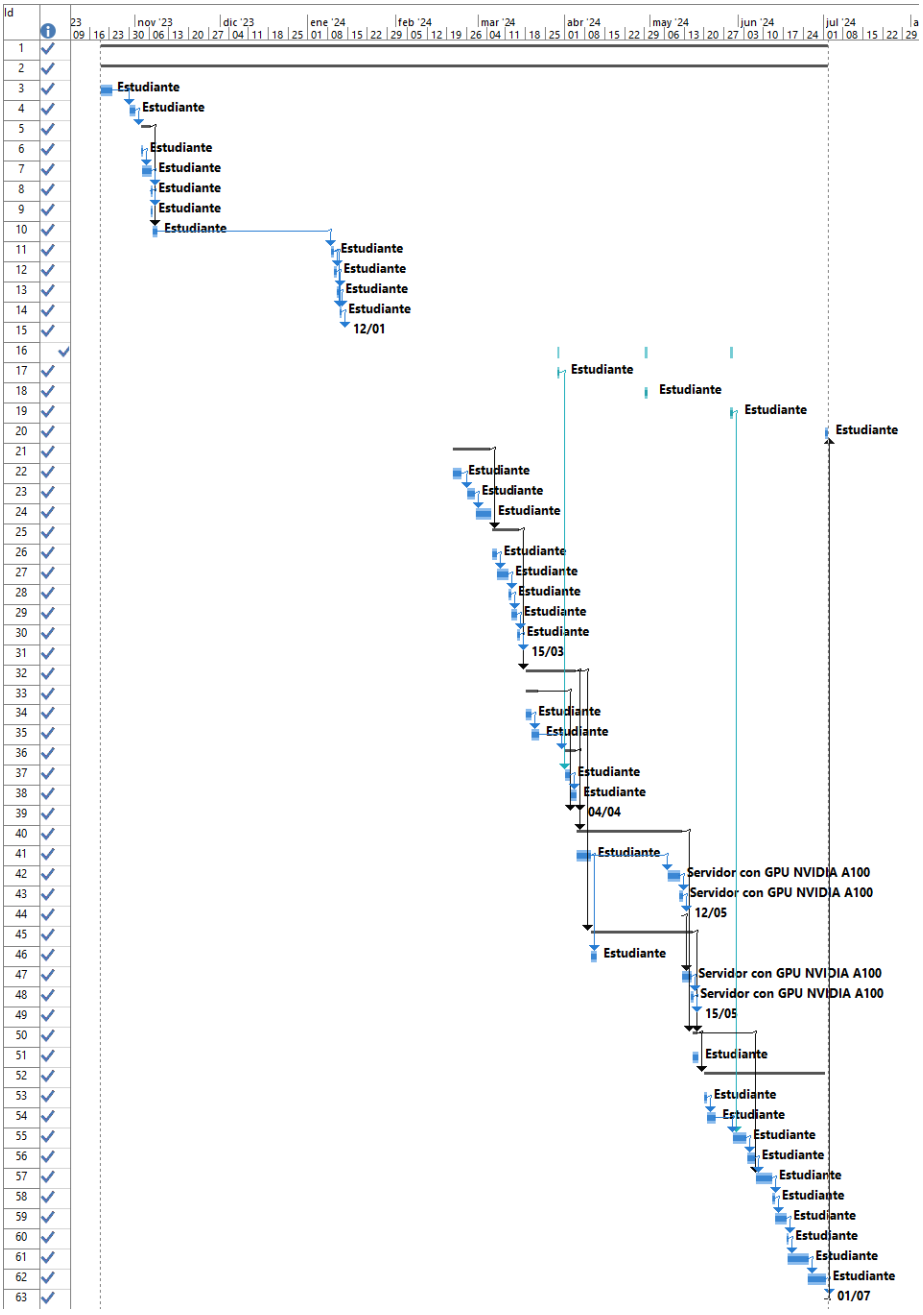


Figura 8.4: Diagrama de Gantt con la planificación final del proyecto.

Bibliografía

[1] A. Abbasi and H. Chen. Applying authorship analysis to extremist-group web forum messages. *IEEE Intelligent Systems*, 20(5):67–75, 2005.

[2] M. Abuhamad, J. su Rhim, T. AbuHmed, S. Ullah, S. Kang, and D. Nyang. Code authorship identification using convolutional neural networks. *Future Generation Computer Systems*, 95:104–115, 2019.

[3] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang. Unified pre-training for program understanding and generation. In K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tür, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668. Association for Computational Linguistics, 2021.

[4] M. Allamanis. *Graph Neural Networks in Program Analysis*, pages 483–497. Springer Nature Singapore, Singapore, 2022.

[5] L. J. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[6] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[7] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah. Signature verification using a "siamese"time delay neural network. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6. Morgan-Kaufmann, 1993.

[8] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[9] L. Buratti, S. Pujar, M. A. Bornea, J. S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang, and G. Domeniconi.

Exploring software naturalness through neural language models. *CoRR*, abs/2006.12641, 2020.

[10] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 255–270, Washington, D.C., Aug. 2015. USENIX Association.

[11] P. Carbonnelle. PYPL PopularitY of Programming Language. `https://pypl.github.io/PYPL.html`, 2024. Accessed: 2024-06-22.

[12] C. E. Chaski. Who's at the keyboard? authorship attribution in digital evidence investigations. *International journal of digital evidence*, 4(1):1–13, 2005.

[13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[14] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton. A simple framework for contrastive learning of visual representations. *CoRR*, abs/2002.05709, 2020.

[15] S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005), 20-26 June 2005, San Diego, CA, USA*, pages 539–546. IEEE Computer Society, 2005.

[16] A. C. Davison and D. V. Hinkley. *Bootstrap methods and their application*. Cambridge university press, 1997.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, and T. Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[18] D. Enriquez, G. Christensen, H. Donovan, J. Lam, N. Wong, S. Dascalu, D. Feil-Seifer, and E. Hand. Authorship verification for hired plagiarism detection. In *Proceedings of the 9th International Conference on Applied Computing & Information Technology*, ACIT '22, pages 19–24, New York, NY, USA, 2023. Association for Computing Machinery.

[19] T. Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[20] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[21] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 57–76. ACM, 2007.

[22] I. Ghory. Using FizzBuzz to Find Developers who Grok Coding. `https://web.archive.org/web/20070207060253/http://tickletux.wordpress.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/`, 2007. Accessed: 2024-06-22.

[23] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742, 2006.

[24] D. Hendrycks and K. Gimpel. Gaussian error linear units (gelus). *arXiv: Learning*, 2016.

[25] P. Hozhabrierdi, D. F. Hitos, and C. K. Mohan. Zero-shot source code author identification: A lexicon and layout independent approach. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2020.

[26] M. Joy, G. Cosma, J. Y.-K. Yau, and J. Sinclair. Source code plagiarism—a student perspective. *IEEE Transactions on Education*, 54(1):125–132, 2011.

[27] P. Juola. Authorship attribution. *Found. Trends Inf. Retr.*, 1(3):233–334, dec 2006.

[28] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. Learning and evaluating contextual embedding of source code. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR, 13–18 Jul 2020.

[29] M. Kestemont, J. Stover, M. Koppel, F. Karsdorp, and W. Daelemans. Authenticating the writings of julius caesar. *Expert Systems with Applications*, 63:86–96, 2016.

[30] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[31] T. Kudo and J. Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In E. Blanco and W. Lu, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium, Nov. 2018. Association for Computational Linguistics.

[32] R. Layton and A. Azab. Authorship analysis of the zeus botnet source code. In *2014 Fifth Cybercrime and Trustworthy Computing Conference*, pages 38–43, 2014.

[33] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[34] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[35] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017.

[36] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[37] W. Ou, S. H. H. Ding, Y. Tian, and L. Song. Scs-gan: Learning functionality-agnostic stylometric representations for source code authorship verification. *IEEE Transactions on Software Engineering*, 49(4):1426–1442, 2023.

[38] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In K. Inui, J. Jiang, V. Ng, and X. Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 3980–3990. Association for Computational Linguistics, 2019.

[39] O. Rodriguez-Prieto, A. Mycroft, and F. Ortin. An efficient and scalable platform for java source code analysis using overlaid graph representations. *IEEE Access*, 8:72239–72260, 2020.

[40] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample. Unsupervised translation of programming languages. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, pages 20601–20611, 2020.

[41] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer*

Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015, pages 815–823. IEEE Computer Society, 2015.

[42] E. Stamatatos, K. Kredens, P. Pezik, A. Heini, J. Bevendorff, B. Stein, and M. Potthast. Overview of the authorship verification task at PAN 2023. In M. Aliannejadi, G. Faggioli, N. Ferro, and M. Vlachos, editors, *Working Notes of the Conference and Labs of the Evaluation Forum (CLEF 2023), Thessaloniki, Greece, September 18th to 21st, 2023*, volume 3497 of *CEUR Workshop Proceedings*, pages 2476–2491. CEUR-WS.org, 2023.

[43] TIOBE. TIOBE Index for June 2024. `https://www.tiobe.com/tiobe-index/`, 2024. Accessed: 2024-06-22.

[44] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[45] D. Trofimov. Google Coding Competitions Archive. `https://zibada.guru/gcj/`, 2023. Accessed: 2024-06-22.

[46] J. Tyo, B. Dhingra, and Z. C. Lipton. Siamese bert for authorship verification. In G. Faggioli, N. Ferro, A. Joly, M. Maistro, and F. Piroi, editors, *Proceedings of the Working Notes of CLEF 2021 - Conference and Labs of the Evaluation Forum, Bucharest, Romania, September 21st - to - 24th, 2021*, volume 2936 of *CEUR Workshop Proceedings*, pages 2169–2177. CEUR-WS.org, 2021.

[47] G. van Rossum, B. Warsaw, and A. Coghlan. PEP 8 – Style Guide for Python Code. `https://peps.python.org/pep-0008/`, 2001. Accessed: 2024-06-22.

[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[49] N. Wang, S. Ji, and T. Wang. Integration of static and dynamic code stylometry analysis for programmer de-anonymization. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, AISec '18, pages 74–84, New York, NY, USA, 2018. Association for Computing Machinery.

[50] R. White and N. Sprague. Deep metric learning for code authorship attribution and verification. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1089–1093, 2021.

[51] Y. Zhao and J. Zobel. Searching with style: authorship attribution in classic literature. In *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*, ACSC '07, pages 59–68, AUS, 2007. Australian Computer Society, Inc.

Apéndice A

Difusión de los resultados

A partir de los resultados de este trabajo, hemos redactado un artículo de investigación titulado *CLAVE: A Deep Learning Model for Source Code Authorship Verification with Contrastive Learning and Transformer Encoders*. Para elegir la revista más apropiada a la que enviar el artículo, hemos realizado un análisis en función de su temática, factor de impacto, categoría (debe estar en una de las categorías de informática), cuartil y tendencia de su impacto. Hemos filtrado las siguientes revistas cuyo ámbito incluye la inteligencia artificial aplicada, con el desarrollo de software como uno de los ámbitos de aplicación:

- *Engineering Applications of Artificial Intelligence.* Con un factor de impacto de 8,0 en el JCR 2022, esta revista se sitúa en el primer cuartil de la categoría de inteligencia artificial. Tendencia ascendente en su factor de impacto.
- *Expert Systems with Applications.* Esta revista obtuvo un factor de impacto de 8,5 en el JCR 2022, situándose también en el primer cuartil de la categoría de inteligencia artificial. Tendencia del factor de impacto constante.
- *Information Processing & Management.* Esta revista se sitúa en el primer cuartil de la categoría de sistemas de información del JCR 2022, con un factor de impacto de 8,6 y trayectoria ascendente.

Finalmente, decidimos enviar el artículo a *Information Processing & Management* porque es la revista con los mejores resultados en el JCR 2022 de las valoradas, además de contar con una evolución muy positiva en los últimos años. Adicionalmente, ofrece un *Master Degree Paper Awards* en el que, anualmente, se elige el mejor artículo escrito por un alumno de máster.

El resto de esta sección contiene una copia del artículo de investigación enviado a la revista.

CLAVE: A Deep Learning Model for Source Code Authorship Verification with Contrastive Learning and Transformer Encoders

David Álvarez-Fidalgo[a], Francisco Ortin[a,b,*]

[a] *University of Oviedo, Computer Science Department, c/ Leopoldo Calvo Sotelo 18, 33007, Oviedo, Spain*

[b] *Munster Technological University, Computer Science Department, Rossa Avenue, Bishopstown, Cork, Ireland*

Abstract

Authorship verification involves determining whether two documents are written by the same author. When applied to source code, authorship verification has many uses, such as malware authorship analysis, copyright dispute resolution, and plagiarism detection. Creating a robust source code authorship verification model is particularly challenging because it needs to generalize to code written by programmers outside of its training data. The Transformer deep learning architecture has recently achieved significant success in various source code processing tasks. However, no previous deep learning model for source code authorship verification has taken advantage of this architecture. In this paper, we present CLAVE (Contrastive Learning for Authorship Verification with Encoder representations), a novel deep learning model for source code authorship verification that leverages contrastive learning and a Transformer Encoder-based architecture. We initially pretrain CLAVE on a large dataset of Python source code files extracted from GitHub. Subsequently, we fine-tune CLAVE for authorship verification using contrastive learning on Python submissions from the Google Code Jam and Kick Start coding competitions. This approach enables the model to learn stylometric representations of source code, which can be compared using a vector distance measure, thereby providing a metric for authorship verification. Our evaluation on a set of submission pairs from these coding competitions has shown that CLAVE outperforms state-of-the-art source code authorship verification models.

Keywords: Source code authorship verification, contrastive learning, transformer encoder, deep learning, stylometric representations, Python

1. Introduction

Authorship attribution is the task of identifying the author of an anonymous document given a set of known authors [26]. Authorship attribution has many applications, including finding authors of anonymous or disputed literary works [50], aiding in criminal investigations [11], or attributing messages associated with known terrorist groups [1].

On the other hand, authorship verification aims to determine whether two documents are written by the same author [41]. Authorship verification is a challenging task because it must generalize to unseen authors, unlike authorship attribution where the documents are classified into a predefined set of known authors. Furthermore, authorship attribution can be reduced to verification by comparing the document of unknown authorship with each document available in the set of known authors. Therefore, authorship verification encompasses all the applications of attribution and extends further, including tasks such as verifying the authorship of historical documents [28] or detecting plagiarism [17].

The general idea of authorship verification can be applied to the specific context of source code, where the objective is to discern whether two code excerpts are authored by the same programmer. Source code authorship verification has various uses across different domains. It can be used to analyze the authorship of malware and link it to other pieces of malicious source code [31]. It also serves as a tool for identifying instances of code plagiarism across various scenarios, encompassing legal disputes concerning intellectual property rights as well as educational settings [25].

Source code authorship verification has some unique characteristics that set it apart from its application in natural language. In source code, it mainly discerns the author's coding style in writing different programs with the same overall semantics. Examples include the naming convention used

*Corresponding author

Email addresses: uo270571@uniovi.es (David Álvarez-Fidalgo), ortin@uniovi.es (Francisco Ortin)

URL: http://www.reflection.uniovi.es/ortin (Francisco Ortin)

```python
def fizz_buzz(max_number: int) -> None:
    for n in range(1, max_number + 1):
        if n % 3 == 0 and n % 5 == 0:
            print("fizzbuzz")
        elif n % 3 == 0:
            print("fizz")
        elif n % 5 == 0:
            print("buzz")
        else:
            print(n)
```

```python
def fizzBuzz(maxNumber):
  number = 0
  while number < maxNumber:
    number += 1
    match number:
      case number if number % 3 == 0 and number % 5 == 0:
        print('fizzbuzz')
      case number if number % 3 == 0:
        print('fizz')
      case number if number % 5 == 0:
        print('buzz')
      case number:
        print(number)
```

Figure 1: Two sample code styles for a Python function with the same purpose.

for identifiers (e.g., `camelCase` or `snake_case`), the preference for `while` or `for` loops, the choice between single or double quotes for strings (in languages like Python that support both), or the usage of imperative versus functional constructs where that choice is allowed (e.g., C#, Scala, and Python). In the realm of natural language, this study of writing style is known as stylometry.

Figure 1 illustrates how two Python functions with the same purpose—the FizzBuzz problem [21]—can be written in two different styles. The first difference lies in the naming convention for the function and parameter names (`fizz_buzz` and `max_number` vs. `fizzBuzz` and `maxNumber`). The first function uses the snake case naming convention whereas the second uses camel case. The name of the variable that holds the current number is also different: the first function uses a shorter name (`n`) while the second uses a longer one (`number`). Regarding control structures, the code excerpt on the left uses a `for` loop with `if` statements to select the message to print for each number, whereas the code on the right uses a `while` loop and a `match` statement. The way strings are written is also different: the first function represents strings within double quotes, whereas the second function employs single quotes. Finally, the code on the left-hand side of Figure 1 includes type hints to declare parameter and return types, while the second one does not.

Deep learning is a subset of machine learning that involves the use of neural networks with multiple layers to automatically learn representations of data at different levels of abstraction [32]. It has achieved significant success in various domains, including computer vision, natural language processing, and speech recognition. Due to its diverse applications, several studies have explored using deep learning for source code authorship verification. Some of them train classifiers for authorship attribution and

repurpose them for authorship verification [24]. Others take advantage of contrastive learning (Section 2.2) to train feedforward networks with manually engineered features [48], or Recurrent Neural Networks (RNNs) with learned features [49, 36].

The Transformer deep learning architecture [47] has enabled numerous recent advancements in natural language processing. It addresses several key limitations of RNNs, such as the challenge of modeling long-term dependencies, and allows for increased parallelization during training. Initially designed for sequence-to-sequence tasks, like machine translation, Transformers comprise two networks: an Encoder and a Decoder. The Encoder processes the input sequence and the Decoder generates the output sequence conditioned by the Encoder's output.

Moreover, the individual utilization of Transformer's Encoder and Decoder has yielded significant achievements in various tasks. For instance, Decoder-only models like GPT [7] and LLaMA [43] have excelled in text generation, while Encoder-only models like BERT [16] and RoBERTa [33] have shown proficiency in text classification and other text comprehension tasks. The Transformer architecture has also been successful in source code processing tasks, with pre-trained models like CodeBERT [19] and CuBERT [27]. Despite its success, prior research has not yet explored the application of this architecture to source code authorship verification.

The main contribution of this paper is a new deep learning model for source code authorship verification, which leverages contrastive learning and a Transformer Encoder-based architecture. Our system, CLAVE (Contrastive Learning for Authorship Verification with Encoder representations), is initially pre-trained on a large dataset of Python[1] source code files extracted from GitHub. We explore and customize various tokenization approaches specifically tailored for Python to effectively preprocess the code. Then, we use contrastive learning to fine-tune CLAVE for source code authorship verification with Python submissions from the Google Code Jam and Kick Start coding competitions. This process allows the model to learn stylometric representations of source code, as different programmers write solutions to the same problems in their particular coding styles. These representations, also known as embeddings, are closer together for source code excerpts written

[1]We chose to train CLAVE with Python source code because it is one of the most popular languages according to different programming language indexes [42, 10].

by the same author and farther apart for different authors. Consequently, the embeddings can be compared using a vector distance measure, thereby providing a metric for authorship verification. Compared to the state of the art, the results show that CLAVE outperforms the existing models for source code authorship verification.

The rest of this paper is structured as follows. First, we present the related work in Section 2. Then, Section 3 describes the architecture of CLAVE and its training process. The methodology followed to obtain the training data and evaluate the model is detailed in Section 4. In Section 5, we present the evaluation of CLAVE and compare it to related systems. Section 6 discusses the impact of CLAVE's most relevant components on its performance, and Section 7 presents the conclusions and future work.

2. Related Work

In this section, we review prior work related to the objective of this paper—source code authorship analysis—and the main techniques used to achieve it: contrastive learning and natural language processing applied to source code.

2.1. Source Code Authorship Analysis

Authorship analysis encompasses a set of tasks focused on determining information about the author of a given document [41]. It includes two primary tasks: authorship attribution and authorship verification. Authorship attribution aims to identify the author of a document from a set of known authors, while authorship verification seeks to determine whether two documents were written by the same author.

Several studies focus on source code authorship verification. Ou *et al.* [36] propose a neural network architecture that combines a bidirectional Long Short-Term Memory (LSTM) with a Generative Adversarial Network (GAN). This architecture is trained as a siamese network to learn stylometric representations of source code, with the GAN ensuring that the derived representations do not contain information about the functionality of the original source code files. These representations are then compared using cosine similarity to determine whether a pair of source code files were written by the same author. The network outperforms several state-of-the-art code representation models across four datasets collected from the 2017 Google Code Jam coding competition.

White *et al.* [49] also present a neural network architecture based on a bidirectional LSTM that learns stylometric representations of source code. The authors train this network using the NT-Xent loss function that is part of the SimCLR framework [13]. The trained network is evaluated for authorship verification by comparing the cosine similarity of the representations. It is also used for authorship attribution with a support vector machine classifier. The network achieves a verification accuracy of 92.94% on a dataset consisting of C/C++ programs from the years 2008 to 2017 of Google Code Jam.

Hozhabrierdi *et al.* [24] propose a method for authorship verification that employs a neural network originally trained for authorship attribution. The network is fed novel features called Variable-Independent Nested Bigrams that are extracted from the abstract syntax tree of each program. The network is initially trained to classify source code files into a set of known authors. Then, a portion of the trained network is repurposed to generate representations from source code, which are subsequently compared for authorship verification. This method obtains an AUC (area under the ROC curve) of 0.96 on a Python dataset collected from Google Code Jam for the years 2006 to 2014.

Wang *et al.* [48] introduce a neural network that accepts a set of manually engineered features extracted from pairs of programs and outputs the probability that the pair was written by the same author. These features include both static (extracted without running the program) and dynamic (extracted after running the program) information, such as run time or memory consumption. The authors report a verification accuracy of 99.91% on a Python dataset from Google Code Jam covering the years 2008 to 2017. However, they provide no additional metrics or information about the class distribution in the dataset.

Although less related to this paper, there are more studies that focus on source code authorship attribution. Caliskan-Islam *et al.* [9] propose a set of manually extracted features from the abstract syntax tree of each program. These features are used to train classifiers to identify the author of a program from a set of given authors. Likewise, Abuhamad *et al.* [2] introduce several Convolutional Neural Networks (CNNs) that take word embeddings or TF-IDF features as input and are trained for authorship attribution.

2.2. Contrastive Learning

Contrastive learning is a self-supervised deep learning technique that aims to learn representations of data such that the distance in representation space is smaller for similar instances and larger for dissimilar ones [22]. Its core idea is to learn meaningful representations by contrasting positive (similar) pairs of data points against negative (dissimilar) pairs.

This technique has been successfully applied in computer vision for tasks such as face recognition and image classification. One of the earliest studies on contrastive learning, by Chopra *et al.* [14], presents a method for learning a similarity metric from data. The authors use this method to train a Convolutional Neural Network (CNN) that learns representations of face images. The CNN is trained on pairs of matching and non-matching faces with the objective of reducing the distance between matching pairs while increasing the distance between non-matching ones. Schroff *et al.* [40] also train a CNN with the same objective, but their approach uses triplets consisting of a reference face, a matching face, and a non-matching face.

Contrastive learning has also been successful in natural language processing. Reimers *et al.* [37] propose a modification of BERT [16] that uses siamese and triplet networks to learn semantic sentence representations that can be compared using cosine similarity.

Due to the nature of source code authorship verification, contrastive learning is a popular technique for this task, as shown in Section 2.1. It has also been applied to authorship verification in natural language documents. Tyo *et al.* [45] modify BERT using different objective functions to learn document representations that are closer when the documents are written by the same authors, and farther apart when they are not.

Finally, SimCLR [13] is a popular contrastive self-supervised learning framework originally designed for computer vision. In SimCLR, the similar instances are generated through data augmentation. During training, for each instance in a minibatch, two new augmented ones are created. The model is then updated using a loss function called NT-Xent, which aims to bring closer together the representations of instances that were augmented from the same original image, while pushing farther apart the representations of instances that originated from different ones.

2.3. Natural Language Processing Applied to Source Code

Deep learning techniques for Natural Language Processing (NLP) have experienced significant growth since the introduction of the Transformer ar-

chitecture [47]. Many of these advancements in NLP have also proven effective when applied to source code.

Feng *et al.* [19] propose CodeBERT, a bimodal pre-trained model for source code and natural language based on the architectures of BERT [16] and RoBERTa [33]. The authors pre-train the model using both source code and natural language, and fine-tune it in two downstream tasks: natural language code search and code documentation generation.

Kanade *et al.* [27] introduce CuBERT, a pre-trained model designed to generate contextual embeddings from source code. CuBERT is also based on the architecture of BERT. The authors pre-train this model using a dataset of Python files from GitHub and fine-tune it in several code understanding tasks, such as variable-misuse classification and function-docstring mismatch.

3. CLAVE

In this section, we describe our system, called CLAVE: Contrastive Learning for Authorship Verification with Encoder representations. We detail its architecture, how source code is fed into the system, the way the model is pre-trained with source code from unknown authors, and how it is fine-tuned for authorship verification.

3.1. Architecture

Figure 2 shows how CLAVE takes source code as input and outputs a single vector (embedding). The system is trained (Sections 3.3 and 3.4) such that output embeddings hold stylometric representations of the input source code. That is, embeddings representing code written by the same author will be close together, whereas those representing different authors will be far apart. At inference time, if the cosine distance ($1-$cosine similarity) between the embeddings of two source code excerpts obtained by feeding them into CLAVE, is greater than a threshold γ, we predict that the excerpts were written by different authors. If the distance is less than or equal to γ, we predict that they were written by the same author.

The first component of CLAVE (Figure 2) is a tokenizer that takes the input source code and converts it into a vector (detailed in Section 3.2). This vector is then passed to a 6-layer Transformer Encoder, which outputs a matrix containing an embedding for each input token. The Transformer Encoder follows the architecture proposed by Vaswani *et al.* [47]. Similar to

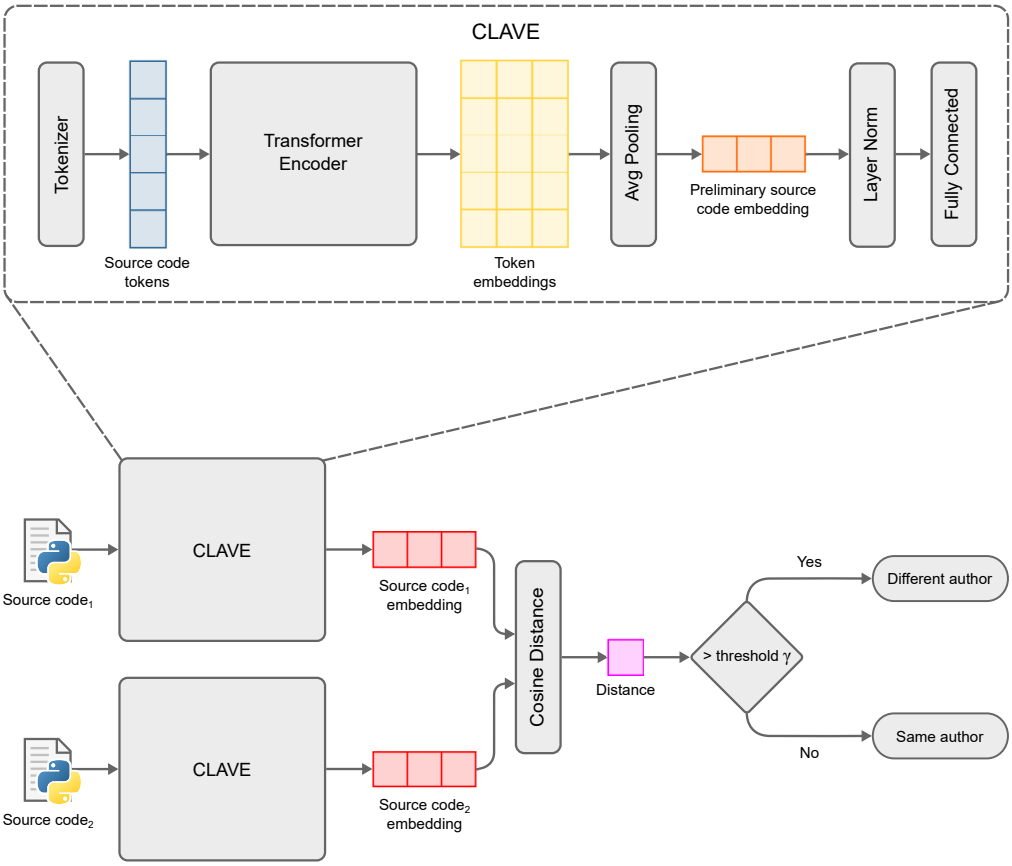


Figure 2: The architecture of CLAVE.

BERT [16], we use learned positional embeddings instead of sinusoidal encoding and the GELU activation function instead of ReLU, as GELU has shown superior performance in various language processing tasks [23]. We configure the Transformer Encoder with model dimension $d_{model} = 512$, feedforward network dimension $d_{ff} = 2048$, and $h = 8$ attention heads.

The token embedding matrix produced by the Encoder is then pooled to obtain a single embedding representing the entire input source code. This is achieved by averaging each embedding feature across all the tokens, resulting in a single source code embedding with the same dimensions as the individual token embeddings. This approach ensures that we always obtain a single vector regardless of the input length.

The source code embedding then is normalized using Layer Normalization

to address internal covariate shift and improve the stability and performance of the network during training [5]. The normalized vector is finally processed by a fully connected layer with a ReLU activation function. This final layer allows the model to learn a new representation based on the information about the entire input contained in the pooled embedding. This representation serves as the output of the model and holds the stylometric information of the source code.

3.2. Input Representation

An important part of the system is how to represent Python source code as vectors. We experiment with three different tokenization approaches: SentencePiece customized for Python code, a new tokenizer designed specifically for Python that we call the Stylometric Tokenizer, and a combination of the customized SentencePiece and the Stylometric Tokenizer.

3.2.1. SentencePiece for Python

SentencePiece is a subword tokenizer developed by Google, designed to tokenize text into smaller subword units [30]. Unlike traditional methods, such as word-based or character-based tokenization, SentencePiece operates at the subword level, breaking text into smaller pieces that may correspond to whole words or parts of words. This feature is particularly helpful for tokenizing computer programs, where the vocabulary is typically unlimited. For instance, identifiers in Python can be of any length, resulting in an infinite vocabulary.

SentencePiece requires training on a corpus of text data before it can be used for tokenization. During this training, the tokenizer learns a vocabulary of subword units based on the frequency and likelihood of different character sequences in the training data. We train SentencePiece using the pre-training dataset described in Section 4.1.1, setting the vocabulary size to 16,000 tokens.

To efficiently tokenize Python source code, and to retain part of the structure of the original program in its vector form, we instruct SentencePiece to extract the following Python language elements as a single token: keywords, operators, equals signs, parentheses, braces, brackets, colons, semicolons, commas, dots, and ellipses. We also configure SentencePiece to extract whitespace characters (spaces, tabs, and newlines) as single tokens, as they can be useful to determine the coding style (e.g., spaces versus tabs in Figure 1).

3.2.2. Stylometric Tokenizer

This tokenizer is a modified version of Python's `tokenize` module. That module is part of the standard library and provides functionality for lexical analysis of Python source code. It breaks down Python source code into tokens, which are the smallest units of a program, such as keywords, identifiers, literals, and operators.

We modify the tokenizer to generate additional tokens associated with the stylometric features of the input source code. These additional tokens, listed in Table 1, encompass various aspects such as the different ways numeric and string literals can be written, the various naming conventions for identifiers, and whether comments start with a single whitespace following Python's style guide [46]. In addition to the tokens listed in Table 1, we add one token for each whitespace character (space, tab, and newline) that would otherwise be ignored by Python's `tokenize` module.

3.2.3. Stylometric Tokenizer with SentencePiece

This is an extension of the Stylometric tokenizer (Section 3.2.2) that incorporates the tokens obtained by applying SentencePiece to identifiers. Thus, identifiers are both represented with the tokens in Table 1 and with subwords obtained from another SentencePiece for Python tokenizer (Section 3.2.1).

We train a separate SentencePiece tokenizer exclusively on the set of identifiers extracted from the pre-training dataset (Section 4.1.1). The vocabulary size of this SentencePiece tokenizer is configured so that the total number of unique tokens, including those from the Stylometric tokenizer, amounts to 16,000.

3.3. Pre-Training

We pre-train CLAVE using Masked Language Modeling (MLM), following the approach proposed by BERT [16] and refined by RoBERTa [33]. We tokenize source code files from the pre-training dataset and randomly mask and mutate their tokens. Then, we train CLAVE's Transformer Encoder to predict the correct tokens for the masked and mutated positions. We opt for MLM because it appears to facilitate learning how to distinguish different coding styles: a masked position may be filled with different tokens, but only the one that aligns with the author's coding style, given the context of the rest of the code, will be correct.

Figure 3 illustrates the pre-training procedure of CLAVE. A masking process is applied to the vector of source code tokens. Initially, 15% of the

Token	Description	Example
NUMBER_UNDERSCORE	Number written with underscores	1_000
NUMBER_HEX	Hexadecimal number	0xa0
NUMBER_BIN	Binary number	0b10
NUMBER_OCT	Octal number	0o12
NUMBER_EXP	Exponential number	1e-10
NUMBER_IMG	Complex number	1j
NUMBER_LCASE	Number that contains a lowercase letter	1e2
NUMBER_UCASE	Number that contains an uppercase letter	1E2
NUMBER_LEADING_DOT	Number with a leading dot	.1
NUMBER_TRAILING_DOT	Number with a trailing dot	1.
NUMBER_EXP_PLUS	Exponential number with a plus sign	1e+2
NUMBER_LEADING_PLUS	Number with a leading plus sign	+1
NUMBER_LEADING_ZEROS	Number with leading zeros	001
NUMBER_TRAILING_ZEROS	Decimal number with trailing zeros	1.100
STRING_SQUOTE	String enclosed in single quotes	'hello'
STRING_DQUOTE	String enclosed in double quotes	"hello"
STRING_TRIPLE_SQUOTE	String enclosed in triple single quotes	"'hello"'
STRING_TRIPLE_DQUOTE	String enclosed in triple double quotes	"""hello"""
STRING_MULTILINE	Multi-line string	"""he\llo""
STRING_PREFIX_B	Byte string	b"Hello"
STRING_PREFIX_R	Raw string	r"Hello"
STRING_PREFIX_U	Unicode string	u"Hello"
STRING_PREFIX_F	Formatted string	f"{1 + 1}"
STRING_PREFIX_LCASE	String with a lowercase prefix	f"{1 + 1}"
STRING_PREFIX_UCASE	String with an uppercase prefix	F"{1 + 1}"
NAME_CAMEL_CASE	Identifier written in camel case	camelCase
NAME_PASCAL_CASE	Identifier written in Pascal case	PascalCase
NAME_SNAKE_CASE	Identifier written in snake case	snake_case
NAME_UPPERCASE	Identifier written in all caps	ALL_CAPS
NAME_MIXED_CASE	Identifier that doesn't follow any other case	Mixed_Case
COMMENT_LEADING_WS	Comment with a whitespace after the "#"	# Comment

Table 1: The special tokens used by the stylometric tokenizer.

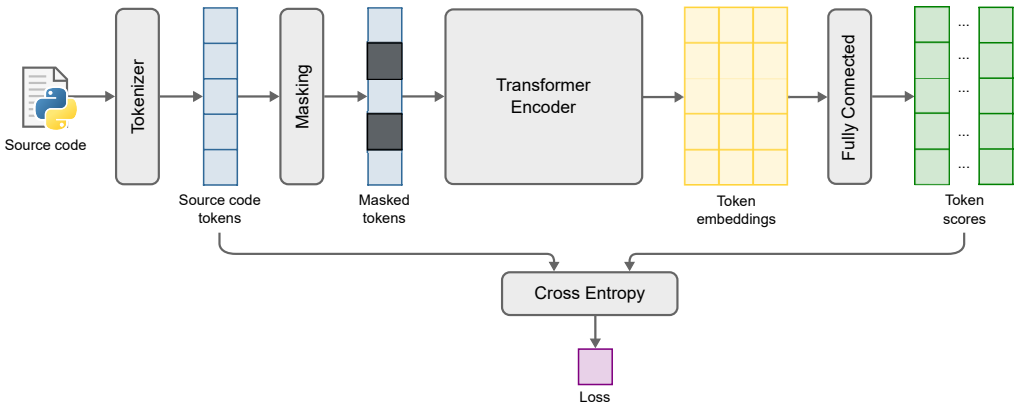


Figure 3: The pre-training process of CLAVE.

tokens are randomly selected. Among these selected tokens, 80% of them are replaced with a special mask token, 10% are replaced with a random token, and the remaining 10% are left unchanged. The masked tokens are then fed into CLAVE's Transformer Encoder, yielding a matrix of token embeddings (one embedding per token). Subsequently, each embedding is projected by a fully connected layer with a softmax activation function into a vector containing the probability for each vocabulary token. Finally, the pre-training loss is computed as the cross entropy between the output token probabilities and the original tokens, only for the 15% selected positions.

3.4. Fine-Tuning

To fine-tune CLAVE for source code authorship verification, we integrate the pre-trained Transformer Encoder with the rest of the model and follow the contrastive learning process described in the SimCLR framework [13], adapted to authorship verification by White *et al.* [49].

Figure 4 shows the fine-tuning process of CLAVE. Each training batch is built from the fine-tuning dataset by randomly sampling N authors without replacement. We then randomly select two source code files written by each of the N authors and feed them into CLAVE. The $2N$ instances of CLAVE that process each file in the batch all share the same weights. The resulting $2N$ source code embeddings are finally used to compute the training loss. We employ the NT-Xent (Normalized Temperature-Scaled Cross-Entropy) loss function, as suggested by the SimCLR framework.

The NT-Xent loss function $\ell_{i,j}$ is defined for a pair of source code files (i,

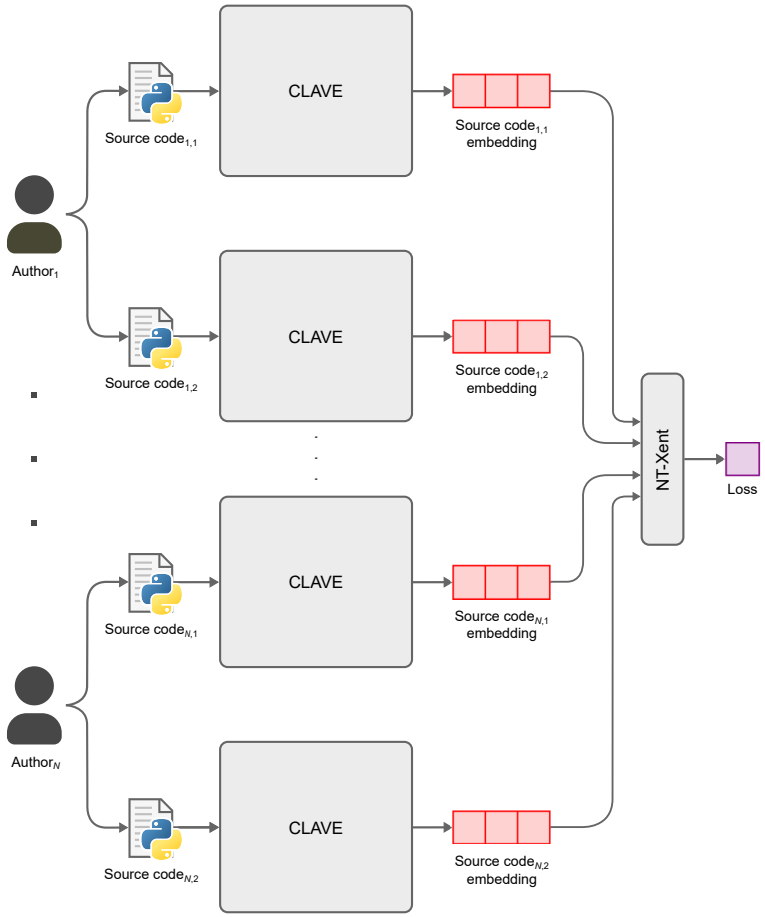


Figure 4: The fine-tuning process of CLAVE.

j) written by the same author with Equation 1. In that equation, $\mathbf{z}$ denotes a batch of source code embeddings, where $\mathbf{z}_i$ and $\mathbf{z}_j$ represent the embeddings of the two files written by the same author (positive pair). $sim(\mathbf{z}_k, \mathbf{z}_l)$ denotes the cosine similarity between the $\mathbf{z}_k$ and $\mathbf{z}_l$ embeddings, and τ is a temperature hyperparameter used to control the sharpness of the distribution [13].

$$\ell_{i,j} = -\log \frac{\exp(sim(\mathbf{z}_i, \mathbf{z}_j)/\tau)}{\sum_{\substack{k=0 \\ k \neq i}}^{2N} \exp(sim(\mathbf{z}_i, \mathbf{z}_k)/\tau)} \tag{1}$$

The NT-Xent loss function turns authorship verification into a classifi-

cation task. Specifically, it aims to identify, for each source code file in a training batch, the corresponding file written by the same author.

4. Methodology

This section describes the datasets used for pre-training and fine-tuning CLAVE, the experiments conducted to evaluate its performance and the impact of various training choices, and the details on how those experiments were carried out.

4.1. Datasets

4.1.1. Pre-Training Dataset

Our pre-training dataset comprises Python files sourced from GitHub. We use GitHub's API to search for Python repositories with open-source permissive licenses (MIT and Apache) to avoid copyright issues and respect the preferences of the repository owners. We exclude repositories with fewer than 100 GitHub stars to maintain a minimum quality standard, and those identified as malware by the system's antivirus (Windows Defender).

We sample 6,132 repositories from GitHub's API. Afterwards, we clone the repositories, remove every file without a ".py" extension, and proceed with the subsequent pre-processing steps. We first remove empty files. Next, to exclude automatically generated code, we filter out files larger than 1MB or with lines longer than 1000 characters, as suggested by Chen *et al.* [12]. Finally, we remove license headers found in many source code files since they are actually written in natural languages.

The resulting 270,602 Python files are split into two partitions: 90% (243,542 files) for training and 10% (27,060 files) for validation.

4.1.2. Fine-Tuning Dataset

We fine-tune CLAVE with Python 3 files collected from submissions to Google Code Jam and Kick Start. These programming contests, held by Google from 2003 to 2023, provide participants with problem statements they must solve by writing code within a predefined time frame. We select this source because it provides a large amount of source code files associated with their respective authors. Moreover, it has been consistently used for training and evaluation in prior works on source code authorship verification [48, 24, 49, 36].

Partition	Number of authors	Number of files
Training	61,956	655,542
Validation	7,618	81,955
Test	8,129	81,908

Table 2: Statistics of the fine-tuning dataset.

As Google's repository of programming competitions is no longer accessible, we obtain the submissions from an unofficial archive [44]. This archive contains source code files for Google Code Jam from 2018 to 2023 and for Kick Start from 2019 to 2022.

We decide to exclusively use Python 3 source code files to ensure that the model does not interpret the syntactic differences between Python 2 and 3 as authors' stylometric differences.

Finally, we divide the set of authors of the selected source code files into three partitions: 80% for training, 10% for validation, and 10% for testing. Splitting the dataset by authors ensures that the model is not evaluated on authors whose files were used for training. Table 2 shows the number of unique authors and source code files in each partition.

4.2. Evaluation

To evaluate CLAVE, we approach authorship verification as a one-shot binary classification task, where no source code of the author to verify was used in the training process. The goal is to classify pairs of source code excerpts as either written by different authors (positive class[2]) or by the same author (negative class). We use various binary classification metrics to assess the performance of CLAVE and compare it with the baseline systems (Section 4.3).

To obtain the pairs, we randomly sample 100,000 source code file pairs from the test partition of the fine-tuning dataset (Section 4.1.2), ensuring an equal number of pairs written by different authors and pairs written by the same author. Next, we filter out pairs containing files that cannot be parsed by Python's `ast` module, as one of the baselines we compare CLAVE

[2]Since we compute the distance of the two embeddings and compare that value with a given threshold, we keep the higher values as the positive class (different authors) and the closer values as the negative label (same authors).

against requires source code to be syntactically correct. Finally, we balance the dataset so that half of the selected pairs are written by different authors and the other half by the same programmer. To achieve this, we remove pairs from the overrepresented class until the dataset is balanced. The final set contains 85,286 pairs, with 42,643 pairs for each class.

As mentioned in Section 3.1, we need to select a threshold γ to make our system behave as a binary classifier. Therefore, we take the first 10% of the pairs (8,528 pairs) to select γ and the remaining 90% (76,758 pairs) to compute the evaluation metrics. For the selected 10% of pairs, we choose the value of γ that achieves the highest F_1-score. In case of a tie, we select the value with the highest accuracy.

We use the following evaluation metrics, where TP (true positives) is the number of pairs correctly classified as written by different authors, TN (true negatives) is the number of pairs correctly classified as written by the same author, FP (false positives) is the number of pairs incorrectly classified as written by different authors, and FN (false negatives) is the number of pairs incorrectly classified as written by the same author:

- **AUC.** The area under the ROC curve [18] provides an aggregate measure of performance across all possible γ values. It can be interpreted as the probability of the distance between the embeddings of two source code excerpts written by different authors being greater than the distance between the embeddings of two source code excerpts written by the same author. This metric is particularly relevant because it does not depend on the selected γ value.

- **Accuracy.** The ratio of correctly classified pairs (Equation 2).

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{2}$$

- **Precision.** The ratio of correctly predicted positive observations to the total predicted positive observations (Equation 3). It measures the accuracy of the positive predictions. In our problem, it is the ratio of correctly classified pairs among those classified as written by the different authors.

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

– **Recall.** The ratio of correctly predicted positive observations relative to all the observations in the positive class (Equation 4). Recall measures the ability of the model to find all the positive cases within a dataset. That is, the ratio of correctly classified pairs among those written by the different authors.

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

– **F_1-score.** The harmonic mean of precision and recall, which represents both metrics with a single value (Equation 5).

$$F_1\text{-}score = \frac{2TP}{2TP + FP + FN} \tag{5}$$

Finally, we employ bootstrapping with 1,000 repetitions to calculate 95% confidence intervals for each metric [15]. This method involves repeatedly sampling with replacement from the set of test pairs to create multiple resampled datasets. We then compute the confidence intervals for the average of each metric across all resampled datasets to determine whether there are statistically significant differences among the compared systems [20].

4.3. Baseline Systems

We compare the performance of CLAVE against the following source code authorship verification methods described in Section 2.1: FDR [24], SCS-Gan [36], and the model proposed by White *et al.* [49]. We exclude SUNDAE [48] due to its requirement of running input source code files, which poses a security risk. These are, to the best of our knowledge, the state-of-the-art deep learning models for source code authorship verification. Additionally, we compare CLAVE against CodeBERT [19] (Section 2.3), a popular state-of-the-art model for source code that has been successfully used for code generation [35], search [19], summarization [3], translation [39], and defect detection [8].

Since the source code and data for FDR and SCS-Gan are not available, and the work by White *et al.* only considers C/C++ source code, we replicate the training of their models following the processes and hyperparameters they outline in their articles. Our only deviation is training the model proposed by White *et al.* with a batch size of 128 instead of the suggested 1000, due to memory constraints. We use the same fine-tuning dataset as with

CLAVE (Section 4.1.2) to train these models. Since the output of CodeBERT is a matrix of token embeddings, we employ average pooling to obtain a single source code embedding (as we do for CLAVE), which we then use for authorship verification.

In the evaluation, we use as input the first 512 tokens from each Python file, padding shorter sequences with a special token. We select this length because it is the maximum sequence length that CodeBERT supports. FDR takes a set of nested bigrams as input instead of a sequence of tokens, thus we set the input size to 8000 bigrams as suggested by the authors.

4.4. Impact of the Main Components

Some key components of CLAVE may significantly impact its performance for source code authorship verification. We would like not only to select the components that offer the best performance but also to quantify their benefits. In this section, we describe the experiments conducted to measure the impact of these components. Subsequently, Section 4.5 outlines the process used for the hyperparameter tuning of CLAVE.

To select the best tokenizer from those presented in Section 3.2, we repeat the pre-training and fine-tuning processes with each of the three tokenizers proposed. We also include a SentencePiece tokenizer not customized for Python to quantify the potential benefits of customization. Once we identify the tokenizer with the best performance (using the AUC score), we use it for the remaining experiments, including the comparison with the baselines.

We conduct other studies to analyze the impact of modifying or removing other relevant components of CLAVE. A key part of the fine-tuning process (Section 3.4) is the loss function. We evaluate the impact of replacing NT-Xent with two popular contrastive learning objectives described in Section 2.2: contrastive loss [14] and triplet loss [40]. Additionally, we experiment with two different batch sizes for NT-Xent: 16 and 32.

Another important factor in CLAVE's training process is the pre-training. No previous work on source code authorship verification has leveraged the idea of pre-training the model using a larger dataset. We evaluate its contribution to the performance of our model by comparing CLAVE's results with and without pre-training.

The last component we modify is the Transformer Encoder. Besides the base configuration, we experiment with two other configurations of the Transformer Encoder by varying the hyperparameters d_{model}, d_{ff}, and h as detailed in Table 3. For each configuration, we repeat the entire training process. We

Configuration	d_{model}	d_{ff}	h
Base	512	2048	8
Small	256	2048	8
Large	768	3072	12

Table 3: The different Transformer Encoder configurations evaluated. d_{model}, d_{ff} and h denote, respectively, model dimension (i.e., embedding size), feedforward network dimension and the number of attention heads.

fix the number of layers at 6, which is the maximum value that allows us to conduct all experiments within our memory constraints.

In addition to the components of CLAVE itself, another factor that can significantly influence its performance is the length of the input. To assess its impact, we initially pre-train the model with an input length of 512 tokens. Then, we repeat the fine-tuning process with input lengths of 256, 512, 1024, and 2048 tokens. We only change the input length during fine-tuning to avoid repeating the expensive pre-training process.

4.5. Training an Hyperparameter Search

To pre-train CLAVE, we use the AdamW [34] optimizer with its default parameters of $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and a learning rate of 10^{-4}. We warm up the learning rate for 100,000 steps and then linearly decay it to 0 over 1,500,000 steps. We select these values by observing the evolution of the validation loss during a previous training run. CLAVE is pre-trained with a batch size of 32, the largest value possible to complete all experiments with the memory available.

To fine-tune CLAVE, we conduct a grid search to find the hyperparameters, detailed in Table 4, that achieve the best validation AUC. In this case, we warm up the learning rate for 40,000 steps and then linearly decay it to 0 over 500,000 steps. These values are selected in the same manner as with the pre-training step. We fine-tune with a batch size of 32 when using contrastive and triplet loss as the training objective, and with the sizes described in Section 4.4 when using NT-Xent. We configure the contrastive and triplet loss functions with their default margin of 1, and NT-Xent with a temperature of $\tau = 0.005$ as suggested by [49].

For both pre-training and fine-tuning, we clip the gradient norm to 1.0 in order to avoid exploding gradients. Additionally, we apply dropout with

Hyperparameter	Range	Result
Final layer dimension	$[d_{model}, d_{model}/2, d_{model} \times 2]$	d_{model}
Final layer dropout probability	[0, 0.2]	0
Learning rate	$[5 \times 10^{-5}, 3 \times 10^{-5}, 2 \times 10^{-5}]$	3×10^{-5}
Optimizer	[Adam [29], AdamW [34]]	AdamW

Table 4: The hyperparameters of CLAVE optimized using grid search. Both optimizers were evaluated with their default parameters of $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

a probability of 0.1 to the Transformer Encoder, following the approach of BERT [16].

All the models were trained using the PyTorch deep learning framework. The training was conducted on hardware comprising a NVIDIA RTX 3070 with 8 GB of VRAM and one fourth of a shared NVIDIA A100 with 80 GB of VRAM (20 GB available).

5. Evaluation

In this section, we compare the performance of CLAVE against the state-of-the-art source code authorship verification systems (Section 4.3). The next section presents a series of experiments (detailed in Section 4.4) to analyze how different factors affect CLAVE's performance.

Table 5 shows the evaluation metrics obtained by CLAVE with the following settings: SentencePiece for Python tokenizer, the NT-Xent loss function ($N = 16$) and the base configuration in Table 3. We compare this CLAVE model with the baseline systems, employing the pairs in the test set. As described in Section 4.3, all the systems but FDR are evaluated with an input length of 512 tokens.

Our system, CLAVE, achieves the best results across all the metrics with statistically significant differences, obtaining an AUC of 0.9782. Next is the system proposed by White *et al.*, with 0.9696 AUC. SCS-Gan and FDR follow with, respectively, 0.9594 and 0.8112 AUC. CodeBERT obtains the lowest metrics overall, except for recall, with an AUC of 0.6986.

CLAVE's pre-trained Transformer Encoder (Section 6.4) and its customized tokenizer (Section 6.1) make our model outperform the baselines. The system proposed by White *et al.* achieves significantly better results than SCS-Gan, indicating that, despite sharing the same bidirectional-LSTM encoder architecture, a more sophisticated contrastive learning objective (Sim-

System	AUC	Accuracy	F_1-score	Precision	Recall
CodeBERT [19]	0.6986	0.6323	0.7230	0.5800	0.9598
FDR [24]	0.8112	0.7432	0.7836	0.6770	0.9301
SCS-Gan [36]	0.9594	0.8992	0.9040	0.8633	0.9488
White *et al.* [49]	0.9696	0.9129	0.9170	0.8761	0.9620
CLAVE	**0.9782**	**0.9294**	**0.9324**	**0.8945**	**0.9736**

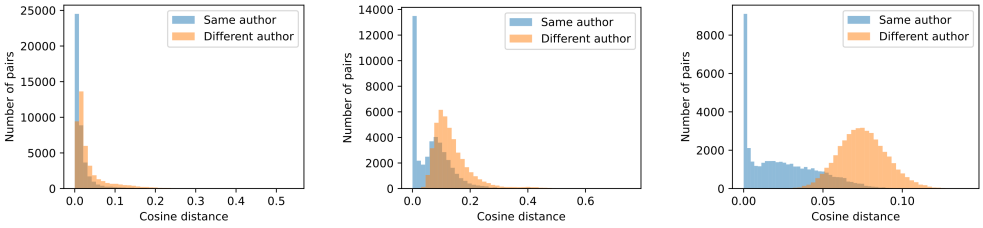
Table 5: AUC, accuracy, F_1-score, precision, and recall of CLAVE (base configuration, SentencePiece for Python tokenizer, and 512 input length) compared to the baselines (Section 4.3) for the set of test pairs. Best results, without overlapping 95% confidence intervals, are highlighted in bold. All 95% confidence intervals are below 0.02%.

CLR) allows for better learning of stylometric representations than the use of a Generative Adversarial Network. That approach has also helped CLAVE improve its performance (Section 6.2).

FDR performs poorly compared to other systems, given its smaller model size and lack of specific training for verification. Finally, CodeBERT, not trained for any authorship-related task, obtains the lowest performance metrics.

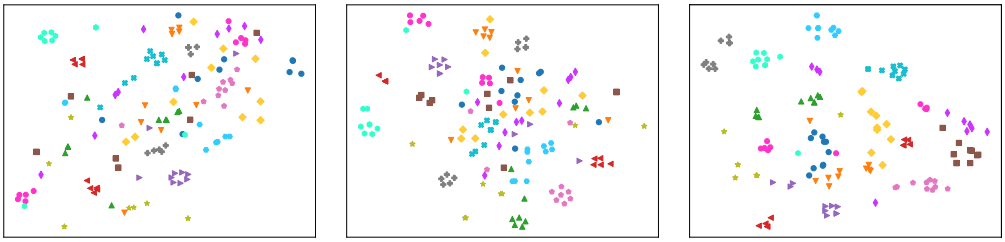
Figure 5 shows the distribution of distances between the test pairs before training CLAVE, after pre-training with no fine-tuning, and after pre-training and fine-tuning. We can see how the overlap between the distributions of same-author pairs and different-author pairs is noticeably reduced after CLAVE is pre-trained and fine-tuned. The large number of same-author pairs with distances very close to 0, even when the model is not pre-trained, can be explained by the existence of multi-part problems in the programming competitions. In these cases, many authors reuse their own code to solve different parts of the same problem, causing code duplication and hence resulting in very similar source code pairs.

Figure 6 illustrates how CLAVE also learns to cluster source code excerpts written by the same author in the embedding space, providing significantly better results when the model is both pre-trained and fine-tuned. The embeddings are the vectors shown in Figure 2 as the “source code embeddings" generated by CLAVE. In Figure 6, those embeddings are reduced to 2 dimensions using t-SNE. The excerpts are randomly sampled from the test partition of the fine-tuning dataset, where 15 authors were selected, and 10 different source code files were chosen for each author.



(a) Randomly initialized CLAVE (b) Pre-trained CLAVE (no fine-tuning) (c) Pre-trained and fine-tuned CLAVE

Figure 5: The distribution of the cosine distances between the test pairs estimated by (a) randomly initialized CLAVE, (b) pre-trained CLAVE with no fine-tuning, and (c) trained CLAVE (base configuration, SentencePiece for Python tokenizer, and 512 input length).



(a) Randomly initialized CLAVE (b) Pre-trained CLAVE (no fine-tuning) (c) Pre-trained and fine-tuned CLAVE

Figure 6: Embeddings generated from 150 different source code excerpts by (a) randomly initialized CLAVE, (b) pre-trained CLAVE with no fine-tuning, and (c) trained CLAVE (base configuration, SentencePiece for Python tokenizer, and 512 input length). The embeddings are reduced to 2 dimensions using t-SNE. Colors and markers indicate the author of each source code excerpt.

6. Discussions

In this section, we conduct a series of experiments (detailed in Section 4.4) to analyze how different factors affect CLAVE's performance. These factors include the choice of a tokenizer, the fine-tuning loss function, the pre-training process, the size of the Transformer Encoder and the input length. All the experiments analyze modifications of the settings evaluated in Section 5: SentencePiece for Python tokenizer, 512 input length, NT-Xent loss function ($N = 16$) and base configuration (Table 3).

Tokenizer	AUC	Accuracy	F_1-score	Precision	Recall
Stylometric	0.9718	0.9197	0.9227	0.8888	0.9593
Stylo. with SentencePiece	0.9748	0.9246	0.9277	0.8912	0.9673
SentencePiece	0.9774	**0.9313**	**0.9336**	**0.9030**	0.9665
SentencePiece for Python	**0.9782**	0.9294	0.9324	0.8945	**0.9736**

Table 6: AUC, accuracy, F_1-score, precision, and recall obtained by CLAVE (base configuration and 512 input length) on the set of test pairs with different tokenizers. Best results, without overlapping 95% confidence intervals, are highlighted in bold. All 95% confidence intervals are below 0.02%.

6.1. Tokenizer

Table 6 compares the performance of CLAVE on the set of test pairs with each of the three tokenizers presented in Section 3.2, together with a plain SentencePiece tokenizer. SentencePiece for Python achieves the highest AUC (0.9782), while the unmodified SentencePiece tokenizer obtains the best accuracy (0.9313) and F_1-score (0.9336). The Stylometric tokenizer obtains the lowest AUC (0.9718), but when combined with SentencePiece, the AUC increases to 0.9748.

The Stylometric tokenizer ignores the contents of identifiers, strings, numeric literals, and comments. Although it produces special tokens for some manually selected stylometric features of these elements (Table 1), SentencePiece allows the model to extract additional features from the raw source code, which helps improve its performance for authorship verification. This improvement becomes evident when the contents of the identifiers tokenized by SentencePiece are included in the Stylometric tokenizer, causing an increase in all the evaluation metrics.

On the other hand, the customizations included in the SentencePiece for Python tokenizer (Section 3.2.1) enable the model to achieve a significantly higher AUC score. However, it is the unmodified SentencePiece tokenizer the one that achieves the best accuracy and F_1-score. We attribute this to the threshold γ selected for our set of test pairs, which allows the model to obtain significantly higher results for these metrics. As mentioned, AUC provides a more robust representation of the model's generalization ability, as it does not depend on the selected γ.

Training objective	N	AUC	Accuracy	F_1-score	Precision	Recall
Contrastive loss	32	0.9513	0.8846	0.8881	0.8616	0.9163
Triplet loss	32	0.9765	0.9267	0.9296	0.8939	0.9684
NT-Xent	32	0.9761	0.9291	0.9313	**0.9038**	0.9605
	16	**0.9782**	**0.9294**	**0.9324**	0.8945	**0.9736**

Table 7: AUC, accuracy, F_1-score, precision, and recall obtained by CLAVE (base configuration, SentencePiece for Python tokenizer, and 512 input length) on the set of test pairs for different loss functions and batch sizes (N): contrastive loss ($margin = 1$), triplet loss ($margin = 1$) and NT-Xent ($\tau = 0.005$). Best results, without overlapping 95% confidence intervals, are highlighted in bold. All 95% confidence intervals are below 0.02%.

6.2. Fine-Tuning Loss Function

Table 7 presents the impact of different loss functions on the performance of CLAVE, evaluated on the set of test pairs after fine-tuning. We use the three loss functions detailed in Section 4.4, with two different batch sizes for NT-Xent. The best results for all the metrics but precision are obtained by NT-Xent with a batch size of 16 (AUC of 0.9782). The same loss function with $N = 32$ achieves the highest precision, but, compared to NT-Xent with $N = 16$, the drop in recall is greater than the gain in precision—as evidenced by the significant decrease in F_1-score.

The NT-Xent loss function achieves superior performance as it enables the model to learn more from hard negatives. A hard negative is a pair of source code excerpts authored by different individuals but whose embeddings are closer than those from excerpts by the same author. NT-Xent achieves this by comparing each source code excerpt to a total of $2 \times (N - 1)$ negative instances in a batch of size N. In contrast, triplet loss compares each excerpt to only 1 negative instance, and contrastive loss compares it to either 1 or 0 negative instances. Moreover, NT-Xent weights the negative instances by their relative hardness [13], prioritizing hard negatives in the loss computation.

Contrary to the findings by Chen *et al.* [13], who observed performance improvements with larger batch sizes, our results indicate that smaller batches may yield better performance. We attribute this to the SimCLR framework's approach of randomly performing data augmentation for each instance within a training batch. On the contrary, we consistently train CLAVE using the same unmodified source code excerpts. As a result, a larger batch size means that CLAVE learns from the same hard negatives more often, which poten-

Configuration	AUC	Accuracy	F_1-score	Precision	Recall
Small	0.9762	0.9279	0.9302	0.9011	0.9613
Base	**0.9782**	**0.9294**	**0.9324**	0.8945	**0.9736**
Large	0.9762	0.9288	0.9310	**0.9028**	0.9611

Table 8: AUC, accuracy, F_1-score, precision, and recall obtained by CLAVE (SentencePiece for Python tokenizer and 512 input length) on the set of test pairs with each of its three configurations. Best results, without overlapping 95% confidence intervals, are highlighted in bold. All 95% confidence intervals are below 0.02%.

tially leads to faster overfitting.

6.3. Model Size

The different Transformer Encoder configurations detailed in Table 3 may have a significant impact on CLAVE's performance. Table 8 lists the evaluation metrics obtained using the set of test pairs for these configurations. The base configuration achieves the highest AUC score of 0.9782, while both the small and large configurations obtain an AUC of 0.9762.

Compared to the small configuration, the higher number of parameters and the larger dimensionality of the output embeddings allow CLAVE's base configuration to learn stylometric representations that achieve superior performance in authorship verification. On the other hand, the observed decrease in performance with the large configuration might indicate that the amount of data available in the fine-tuning dataset is insufficient to fully leverage the increased model size.

6.4. Pre-Training

Table 9 presents the evaluation of CLAVE with and without pre-training, following the method described in Section 4.4. We can see how the AUC, accuracy and F_1-score measures are increased, respectively, 1.84%, 3.28% and 3.07% with pre-training.

These results show that pre-training CLAVE with Masked Language Modeling (MLM) (Section 3.3) can effectively improve its performance for authorship verification. Furthermore, Figures 5 and 6 show that after pre-training alone, the model can generate embeddings closer together for source code written by the same author and farther apart for different authors. This

Pre-training	AUC	Accuracy	F_1-score	Precision	Recall
No	0.9605	0.8999	0.9046	0.8642	0.9490
Yes	**0.9782**	**0.9294**	**0.9324**	**0.8945**	**0.9736**

Table 9: AUC, accuracy, F_1-score, precision, and recall obtained by CLAVE (base configuration, SentencePiece for Python tokenizer, and 512 input length) on the set of test pairs with and without pre-training. Best results, without overlapping 95% confidence intervals, are highlighted in bold. All 95% confidence intervals are below 0.02%.

Input length	AUC	Accuracy	F_1-score	Precision	Recall
256	0.9720	0.9169	0.9203	0.8843	0.9594
512	0.9782	0.9294	0.9324	0.8945	**0.9736**
1024	**0.9792**	**0.9340**	**0.9359**	**0.9098**	0.9634
2048	0.9770	0.9294	0.9315	0.9045	0.9602

Table 10: AUC, accuracy, F_1-score, precision, and recall obtained by CLAVE (base configuration and SentencePiece for Python tokenizer) on the set of test pairs with different input lengths. Best results, without overlapping 95% confidence intervals, are highlighted in bold. All 95% confidence intervals are below 0.02%.

supports our hypothesis that MLM facilitates learning to distinguish different coding styles, allowing us to leverage large amounts of source code for training, even when the author is unknown.

6.5. Input Length

Table 10 shows the evaluation metrics obtained by CLAVE on the test set with the different input lengths mentioned in Section 4.4. The AUC increases as the input length increases, peaking at 0.9792 for inputs with 1024 tokens. This is the highest AUC value achieved by CLAVE among all the different configurations evaluated. For an input length of 2048 tokens, the AUC obtained (0.9770) is significantly lower than that for 512 tokens (0.9782), but greater than that for 256 tokens (0.9720).

Up to 1024 tokens, the increases in context length allow the model to generate embeddings that better capture the stylometric features of the input source code. However, as observed in Figure 7, most source code files in the fine-tuning dataset are transformed into vectors of 1024 or fewer tokens after being processed by the tokenizer (SentencePiece for Python). This lack of longer instances might make it more challenging for the model to learn

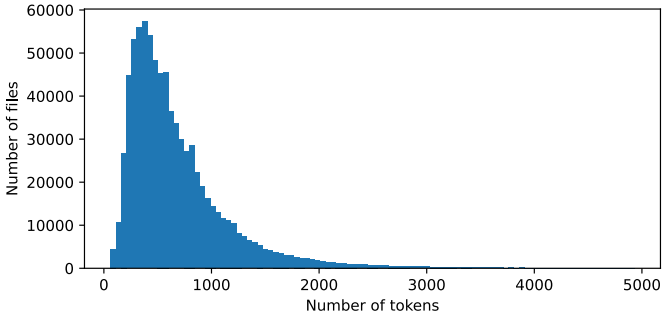


Figure 7: Distribution of the number of tokens in the fine-tuning dataset after removing outliers. The source code files in the dataset are tokenized using the SentencePiece for Python tokenizer.

effective representations from the training data when the input length is set to 2048 tokens. It is also important to note that, as detailed in Section 4.4, the input length for pre-training CLAVE is set to 512 tokens in all cases, which can further hinder the learning of representations for longer contexts.

Another approach to increasing context without extending the input length is to utilize CLAVE with different code fragments of the same input. A common example of this approach is the soft voting method [6]. This technique involves splitting the input, applying the model to each fragment separately, and then aggregating the outputs using a weighted average. As shown in Figure 7, most input files contain 1024 or fewer tokens. Therefore, we decided not to evaluate this approach since most files fit within a single input.

6.6. CLAVE as a Classifier

Although this work focuses solely on authorship verification, the stylometric representations that CLAVE generates could also be used for authorship attribution. As mentioned, authorship attribution is the task of identifying the author of an anonymous document given a set of known authors. Following the approach of White *et al.* [49], a classifier could be trained to predict the author of a source code excerpt, utilizing the embeddings generated by CLAVE as input. Alternatively, the k-nearest neighbors algorithm (kNN) could be employed without additional training. kNN would assign each anonymous source code excerpt the most common author among its k closest embeddings.

7. Conclusions

We show how state-of-the-art source code authorship verification can be achieved by leveraging contrastive learning with the NT-Xent loss function to fine-tune a pre-trained Transformer Encoder. Our system, called CLAVE, generates embeddings that represent the stylometric features of source code, bringing embeddings closer together for source code written by the same author and farther apart for different authors. When evaluated for source code authorship verification using Python submissions to the Google Code Jam and Kick Start programming competitions, CLAVE achieves an AUC of 0.9782 and an F_1-score of 0.9324, significantly outperforming the existing systems. We also analyze the impact of the most relevant components of CLAVE on its performance, specifically the tokenizer, fine-tuning loss function, pre-training, Transformer Encoder configuration, and input length.

As future research, we plan to investigate whether the tree and graph structures of source code could provide any benefit for authorship verification [38]. Syntax and semantic information could be represented as graphs to obtain detailed information about the source code using Graph Neural Networks [4].

The source code of CLAVE, the datasets described in Section 4.1, and the serialized CLAVE weights ready to be used in PyTorch are freely available for download at `https://reflection.uniovi.es/bigcode/download/2024/clave`.

Acknowledgments

This work has been partially funded by the Spanish Department of Science, Innovation and Universities: project RTI2018- 099235-B-I00. We have also received funds from the University of Oviedo through its support of official research groups (GR-2011-0040).

References

[1] A. Abbasi and H. Chen. Applying authorship analysis to extremist-group web forum messages. *IEEE Intelligent Systems*, 20(5):67–75, 2005.

[2] Mohammed Abuhamad, Ji su Rhim, Tamer AbuHmed, Sana Ullah, Sanggil Kang, and DaeHun Nyang. Code authorship identification using

convolutional neural networks. *Future Generation Computer Systems*, 95:104–115, 2019.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668. Association for Computational Linguistics, 2021.

[4] Miltiadis Allamanis. *Graph Neural Networks in Program Analysis*, pages 483–497. Springer Nature Singapore, Singapore, 2022.

[5] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[6] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[8] Luca Buratti, Saurabh Pujar, Mihaela A. Bornea, J. Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. Exploring software naturalness through neural language models. *CoRR*, abs/2006.12641, 2020.

[9] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 255–270, Washington, D.C., August 2015. USENIX Association.

[10] Pierre Carbonnelle. PYPL PopularitY of Programming Language. `https://pypl.github.io/PYPL.html`, 2024. Accessed: 2024-06-22.

[11] Carole E Chaski. Who's at the keyboard? authorship attribution in digital evidence investigations. *International journal of digital evidence*, 4(1):1–13, 2005.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[13] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey E. Hinton. A simple framework for contrastive learning of visual representations. *CoRR*, abs/2002.05709, 2020.

[14] Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005), 20-26 June 2005, San Diego, CA, USA*, pages 539–546. IEEE Computer Society, 2005.

[15] Anthony Christopher Davison and David Victor Hinkley. *Bootstrap methods and their application*. Cambridge university press, 1997.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[17] Daniel Enriquez, Gage Christensen, Hayden Donovan, Jared Lam, Noah Wong, Sergiu Dascalu, David Feil-Seifer, and Emily Hand. Authorship verification for hired plagiarism detection. In *Proceedings of the 9th International Conference on Applied Computing & Information Technology*, ACIT '22, pages 19–24, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[20] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 57–76. ACM, 2007.

[21] Imran Ghory. Using FizzBuzz to Find Developers who Grok Coding. `https://web.archive.org/web/20070207060253/http://tickletux.wordpress.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/`, 2007. Accessed: 2024-06-22.

[22] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742, 2006.

[23] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv: Learning*, 2016.

[24] Pegah Hozhabrierdi, Dunai Fuentes Hitos, and Chilukuri K. Mohan. Zero-shot source code author identification: A lexicon and layout independent approach. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2020.

[25] Mike Joy, Georgina Cosma, Jane Yin-Kim Yau, and Jane Sinclair. Source code plagiarism—a student perspective. *IEEE Transactions on Education*, 54(1):125–132, 2011.

[26] Patrick Juola. Authorship attribution. *Found. Trends Inf. Retr.*, 1(3):233–334, dec 2006.

[27] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR, 13–18 Jul 2020.

[28] Mike Kestemont, Justin Stover, Moshe Koppel, Folgert Karsdorp, and Walter Daelemans. Authenticating the writings of julius caesar. *Expert Systems with Applications*, 63:86–96, 2016.

[29] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[30] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In Eduardo Blanco and Wei Lu, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics.

[31] Robert Layton and Ahmad Azab. Authorship analysis of the zeus botnet source code. In *2014 Fifth Cybercrime and Trustworthy Computing Conference*, pages 38–43, 2014.

[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[33] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[34] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017.

[35] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[36] Weihan Ou, Steven H. H. Ding, Yuan Tian, and Leo Song. Scs-gan: Learning functionality-agnostic stylometric representations for source code authorship verification. *IEEE Transactions on Software Engineering*, 49(4):1426–1442, 2023.

[37] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 3980–3990. Association for Computational Linguistics, 2019.

[38] Oscar Rodriguez-Prieto, Alan Mycroft, and Francisco Ortin. An efficient and scalable platform for java source code analysis using overlaid graph representations. *IEEE Access*, 8:72239–72260, 2020.

[39] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, pages 20601–20611, 2020.

[40] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 815–823. IEEE Computer Society, 2015.

[41] Efstathios Stamatatos, Krzysztof Kredens, Piotr Pezik, Annina Heini, Janek Bevendorff, Benno Stein, and Martin Potthast. Overview of the authorship verification task at PAN 2023. In Mohammad Aliannejadi, Guglielmo Faggioli, Nicola Ferro, and Michalis Vlachos, editors, *Working Notes of the Conference and Labs of the Evaluation Forum (CLEF 2023), Thessaloniki, Greece, September 18th to 21st, 2023*, volume 3497 of *CEUR Workshop Proceedings*, pages 2476–2491. CEUR-WS.org, 2023.

[42] TIOBE. TIOBE Index for June 2024. `https://www.tiobe.com/tiobe-index/`, 2024. Accessed: 2024-06-22.

[43] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[44] Dmitriy Trofimov. Google Coding Competitions Archive. `https://zibada.guru/gcj/`, 2023. Accessed: 2024-06-22.

[45] Jacob Tyo, Bhuwan Dhingra, and Zachary C. Lipton. Siamese bert for authorship verification. In Guglielmo Faggioli, Nicola Ferro, Alexis Joly, Maria Maistro, and Florina Piroi, editors, *Proceedings of the Working Notes of CLEF 2021 - Conference and Labs of the Evaluation Forum, Bucharest, Romania, September 21st - to - 24th, 2021*, volume 2936 of *CEUR Workshop Proceedings*, pages 2169–2177. CEUR-WS.org, 2021.

[46] Guido van Rossum, Barry Warsaw, and Alyssa Coghlan. PEP 8 – Style Guide for Python Code. `https://peps.python.org/pep-0008/`, 2001. Accessed: 2024-06-22.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[48] Ningfei Wang, Shouling Ji, and Ting Wang. Integration of static and dynamic code stylometry analysis for programmer de-anonymization. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, AISec '18, pages 74–84, New York, NY, USA, 2018. Association for Computing Machinery.

[49] Riley White and Nathan Sprague. Deep metric learning for code authorship attribution and verification. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1089–1093, 2021.

[50] Ying Zhao and Justin Zobel. Searching with style: authorship attribution in classic literature. In *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*, ACSC '07, pages 59–68, AUS, 2007. Australian Computer Society, Inc.

Apéndice B

Plan de gestión de riesgos

A continuación, se incluye una copia del plan de gestión de riesgos del proyecto.

MIW-DGPW-GR **PLAN DE GESTIÓN DE RIESGOS**

Título del proyecto: Redes siamesas de transformers para el reconocimiento de programadores a partir de su código fuente **Fecha:** 05/11/2023

1 CAMBIOS

1.1 Cambios en la versión 1.0

- Se ha creado el documento.

2 METODOLOGÍA:

El presente Plan de Gestión de Riesgos recoge el total de los riesgos identificados para el proyecto Redes siamesas de transformers para el reconocimiento de programadores a partir de su código fuente. Estos riegos podrían hacer que el proyecto no llegara a concluirse, suponiendo una importante pérdida económica, de modo que resulta imprescindible categorizar y valorar el impacto que podrían causar cada uno de ellos.

En este documento se incluyen todos los riesgos identificados en la fase de análisis, aunque a lo largo del desarrollo del proyecto estos evolucionarán o cambiará su probabilidad de ocurrencia.

2.1 Metodología General

La metodología que se empleará durante la redacción de este Plan de Gestión de Riesgos toma como base la descrita por Boehm, la cual contiene dos fases:

1. En la primera de las fases se comienza por identificar, analizar y priorizar los riesgos que pueden producirse en el proyecto que se procederá a desarrollar.
2. La segunda fase se encarga de definir las estrategias para tratar y controlar los riesgos definidos en la fase anterior.

Partiendo de esta base, la metodología empleada incorpora nuevos conceptos obtenidos del PMBOK, tales como la matriz de riesgos.

La planificación de riesgos será realizada de acuerdo a una estrategia proactiva, es decir, se realizará una evaluación previa de todos los riegos susceptibles de producirse en el proyecto, evaluando de igual modo las consecuencias que estos podrían acarrear.

2.2 Metodología de Gestión de Riesgos

De acuerdo a la metodología general planteada en el apartado anterior se plantea una metodología de gestión de riesgos que se divide en dos etapas independientes: la evaluación de los riegos y el control de los mismos.

1. Evaluación de los riesgos.
   1.1. Identificación de los riesgos: Permite identificar todas las adversidades que se podrían originar en el contexto de ejecución del proyecto.
   1.2. Análisis de los riesgos: Determinar la probabilidad de que el riesgo se produzca y el impacto que este pudiera originar en el desarrollo del proyecto.
   1.3. Priorización de riesgos: Priorizar aquellos riegos que suponen una verdadera amenaza para el proyecto y que por tanto deberían ser tratados con mayor urgencia.
2. Control de los riesgos.
   2.1. Planificación de los riesgos: Estudio de cómo tratar de evitar o reducir el impacto causado por el riesgo en el proyecto o cómo potenciar los beneficios si el riesgo se entiende como una oportunidad para el proyecto.
   2.2. Solución a los riesgos: Definición de planes de contingencia para evitar o definir cómo proceder si el riesgo se produce verdaderamente.

2.3. Monitorización de los riesgos: Actualización y definición de planes para tener bajo control los riesgos que se identificaron en las primeras fases.

3 HERRAMIENTAS Y TÉCNOLOGÍAS

Para identificar, evaluar y monitorizar los riesgos se utilizarán las siguientes herramientas:

3.1 Tormenta de ideas

Esta técnica será utilizada en un principio para la identificación de la lista principal de los riesgos. Consiste en la discusión acerca de los principales objetivos del proyecto y todas las cosas que pueden ir mal.

3.2 Juicio de expertos

Se utilizará el juicio de expertos para obtener estimaciones que permitan evaluar cuantitativamente los riesgos del proyecto.

3.3 Las evaluaciones periódicas

Regularmente, todos los indicadores serán evaluados de acuerdo con el plan establecido en la hoja de riesgo. Todos estos resultados se discutirán en las reuniones mensuales con el director.

3.4 Reuniones

En las reuniones mensuales con el director se debe incluir un punto de discusión en el orden del día con el fin de tomar decisiones acerca de los riesgos del proyecto y sus efectos.

4 ROLES Y RESPONSIBILIDADES

4.1 Director del proyecto

Posee las funciones de dirigir y realizar el seguimiento de los riesgos. Se encarga de igual modo de la resolución de conflictos surgidos del manejo de estos y de integrar la gestión de riesgos en la gestión del proyecto.

4.2 Responsable de riesgos

Cada riesgo tendrá como asignado un responsable, y como tal se encargará de coordinar las acciones que se deberán llevar a cabo para minimizar el impacto de los riesgos en cuestión.

5 PRESUPUESTO

El presupuesto de la Gestión de riesgos se muestra a continuación:

Item	Concepto	Asignación (€)
1	Identificación de riesgos	160 €
2	Análisis y priorización de los riesgos	240 €
3	Planificación de los riesgos	120 €
4	Definición de planes de contingencia	120 €
5	Actualización y monitorización de los riesgos	240 €
	TOTAL	880 €

MIW-DGPW-GR **PLAN DE GESTIÓN DE RIESGOS**

6 CALENDARIO

Hito / Actividad	Fecha
Identificación de riesgos	**03/11/2023**
Análisis y priorización de los riesgos	**03/11/2023**
Planificación de los riesgos	**06/11/2023**
Definición de planes de contingencia	**06/11/2023**
Reunión mensual 1	**29/03/2024**
Reunión mensual 2	**29/04/2024**

7 CATEGORÍAS DE RIESGO

A fin de poder identificar los riesgos y conocer la estructura de los mismos se categorizan dentro de una de las siguientes categorías, pudiendo pertenecer cada uno de ellos a más de una categoría:

1. Técnico
   1.1. Requisitos
   1.2. Tecnología
   1.3. Prestaciones y fiabilidad
   1.4. Calidad
2. Organizacional
   2.1. Dependencias del proyecto
   2.2. Recursos
   2.3. Financiación
   2.4. Personal
3. Gestión del proyecto
   3.1. Estimación
   3.2. Planificación
   3.3. Control
   3.4. Comunicación
4. Externo:
   4.1. Proveedores
   4.2. Usuario
   4.3. Tiempo
   4.4. Fenómenos

8 DEFINICIONES DE PROBABILIDAD

Nombre	Porcentaje Equivalente	Descripción
Muy Baja	0% - 20%	La probabilidad de que el riesgo se materialice es muy poco probable
Baja	20% - 40%	La probabilidad de que el riesgo llegue a ocurrir es baja
Media	40% - 60%	Existe cierta probabilidad de que el riesgo llegue a producirse
Alta	60% - 80%	Hay altas probabilidades de que la situación que describe el riesgo se de en el contexto del proyecto
Muy Alta	80% - 100%	El riesgo afectará al proyecto con una alta probabilidad

MIW-DGPW-GR **PLAN DE GESTIÓN DE RIESGOS**

9 DEFINICIONES DEL IMPACTO POR OBJETIVOS

Impacto sobre los objetivos principales					
Objetivos de proyecto	*Escalas relativas o numéricas*				
	Muy bajo	*Bajo*	*Moderado*	*Alto*	*Muy alto*
Alcance	Reducciones de alcance poco apreciables	Reducciones que afecten al criterio de éxito de algún objetivo de alcance pero no impidan lograrlo	Reducciones que impidan lograr un objetivo de alcance	Reducciones que impidan lograr más de un objetivo de alcance	Reducciones en proyecto que provocan que deje de ser funcional
Calidad	La degradación de la calidad es poco apreciable	Reducción de la calidad en áreas poco importantes del proyecto	Reducciones de la calidad que requieren la aprobación de los patrocinadores	Reducciones de la calidad inaceptables por los patrocinadores	Reducciones de la calidad que provocan que el proyecto deje de ser funcional
Coste	Incremento poco notable en el coste	Incremento en el coste menor al 5 %	Incremento en el coste de entre 5 % y 10 %	Incremento en el coste de entre el 10 % y el 20 %	Incremento en el coste superior al 20 %
Tiempo	Incremento en el tiempo de menos de 8 horas	Incremento de menos de un 5 % de lo planificado	Incremento en tiempo de entre 5 % y el 10 %	Incremento en el tiempo de entre 10 % y 20 %	Incremento de tiempo superior al 20 %

10 MATRIZ DE PROBABILIDAD E IMPACTO

Probabilidad	**Muy Alta**	***0,90***	0,05	0,14	0,27	0,50	0,81
	Alta	***0,70***	0,04	0,11	0,21	0,39	0,63
	Media	***0,50***	0,03	0,08	0,15	0,28	0,45
	Baja	***0,30***	0,02	0,05	0,09	0,17	0,27
	Muy Baja	***0,10***	0,01	0,02	0,03	0,06	0,09
			0,05	***0,15***	***0,30***	***0,55***	***0,90***
			Muy Bajo	**Bajo**	**Medio**	**Alto**	**Crítico**
			Impacto				

11 TOLERANCIAS

El umbral de riesgo se define en una cota de 0,25. Sobrepasado ese valor se estimará que los riesgos suponen una verdadera amenaza para el proyecto. Los valores de impacto menores de dicho valor no serán tomados como relevantes en un primer término.

12 PLANES DE CONTINGENCIA GENERALES

12.1 Presupuesto

En los momentos en los que resulte imprescindible una alteración del presupuesto definido y acordado antes del inicio del proyecto, estos aspectos serán cuantificados y valorados por el jefe del proyecto y los patrocinadores, y no deberán exceder, en ningún término, un 5% del coste del proyecto.

12.2 Planificación

La única meta definida para el presente proyecto, y que deberá ser cumplida sin excepción es la entrega del proyecto final con una fecha no superior al 10 de julio de 2024.

13 FORMATOS DE LA DOCUMENTACIÓN

Para la gestión de la documentación se tomarán como referencia las siguientes normas:

- UNE-ISO 31000:2010: Gestión del riesgo
- UNE-EN 31010:2011: Gestión del riesgo. Técnicas de apreciación del riesgo

14 SEGUIMIENTO

Para el seguimiento de los riesgos se establece la siguiente política:

- Los riesgos serán analizados y recalculado su impacto cada mes de transcurso del proyecto.
- De forma mensual se realizará un análisis de nuevas amenazas que pudieran surgir en el contexto del proyecto.
- La forma en que se revisarán los riesgos es mediante la reunión mensuales con el director del proyecto.

Apéndice C

Registro de riesgos

A continuación, se incluye una copia del registro de riesgos del proyecto.

ID	Nombre	Responsable	Probabilidad	Impacto				Impacto	0,25	Response
				Presup.	Planific.	Alcance	Calidad		Priorización	
1	Cantidad de datos	Estudiante	Baja	Muy bajo	Muy bajo	Alto	Muy Bajo	0,17		Mitigar el riesgo, considerar técnicas de data augmentation en caso de que la cantidad de datos sea insuficientes
2	Calidad de los datos	Estudiante	Media	Muy Bajo	Muy Bajo	Alto	Muy Bajo	0,28		Mitigar el riesgo, planificar una sesión de selección de datos para obtener un conjunto de códigos fuente que sea representativo de los que elaboran los programadores
3	Recursos computacionales limitados	Estudiante	Muy Baja	Muy Bajo	Muy Bajo	Alto	Muy Bajo	0,06		Aceptar el riesgo, el proyecto se debe realizar con los recursos asignados
4	Compartición de los recursos computacionales	Estudiante	Muy Alta	Muy Bajo	Alto	Alto	Muy Bajo	0,50		Aceptar el riesgo, el proyecto se debe realizar con los recursos asignados
5	Falta de experiencia	Estudiante	Media	Bajo	Medio	Bajo	Medio	0,15		Aceptar el riesgo, el estudiante es quien debe realizar el trabajo
6	Desarrollo novedoso	Estudiante	Media	Bajo	Bajo	Crítico	Muy Bajo	0,45		Aceptar el riesgo, es inherente de todos los proyectos innovadores
7	Complejidad del modelo	Estudiante	Media	Medio	Medio	Medio	Alto	0,28		Aceptar el riesgo, el objetivo del proyecto implica emplear modelos complejos
8	Problemas de convergencia	Estudiante	Baja	Bajo	Bajo	Alto	Crítico	0,27		Aceptar el riesgo, es una consecuencia de trabajar con modelos complejos
9	Sobrecarga de almacenamiento	Estudiante	Media	Bajo	Muy Bajo	Muy Bajo	Muy Bajo	0,08		Aceptar el riesgo, el proyecto se debe realizar con los recursos asignados
10	Seguridad y privacidad	Estudiante	Muy Baja	Medio	Medio	Medio	Medio	0,03		Mitigar el riesgo, seguir prácticas de programación seguras y usar herramientas antivirus para reducir la probabilidad de que el código fuente usado para entrenar el modelo pueda suponer una amenaza para la seguridad
11	Mantenimiento del código	Estudiante	Baja	Bajo	Medio	Bajo	Alto	0,17		Mitigar el riesgo, se utilizarán herramientas, como sistemas de control de versiones, para facilitar el mantenimiento del código
12	Sesgo en los datos	Estudiante	Media	Muy Bajo	Muy Bajo	Medio	Muy Bajo	0,15		Mitigar el riesgo, planificar una sesión de selección de datos para obtener un conjunto de códigos con el menor sesgo posible
13	Problemas de derechos de autor	Estudiante	Baja	Bajo	Muy Bajo	Alto	Muy Bajo	0,17		Evitar el riesgo, para entrenar el modelo solo se utilizará código fuente con licencias permisivas
15	Escalabilidad	Estudiante	Muy Baja	Muy Bajo	Muy Bajo	Alto	Alto	0,06		Aceptar el riesgo, los recursos asignados son suficientes para manejar el volumen de datos previsto
16	Dependencia de software	Estudiante	Baja	Medio	Medio	Medio	Medio	0,09		Mitigar el riesgo, solo utilizar dependencias para las que exista buena documentación y soporte técnico

Apéndice D

Hojas de riesgos

Este apéndice contiene copias de las hojas de seguimiento de los riesgos más prioritarios identificados en el plan de gestión de riesgos (Apéndice B).

D.1 Hoja del riesgo "Recursos computacionales limitados"

A continuación, se incluye una copia de la hoja de seguimiento del riesgo "Recursos computacionales limitados".

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Título del Proyecto: Redes siamesas de transformers para el reconocimiento de programadores a partir de su código fuente **Fecha:** 04/11/2023

ID:	**Nombre:**					
4	Recursos computacionales limitados					
Descripción: Al tener que compartir la GPU NVIDIA A100 puede que el equipo no tenga tiempo suficiente para entrenar un modelo con el rendimiento objetivo y antes de la fecha límite acordada.						
Categoría(s) de riesgo: Recursos, Dependencias del proyecto, Planificación , Estimación						
Status: Activo	**Causas del Riesgo:** • Que el tiempo necesario para entrenar los modelos sea superior al tiempo de uso de la GPU asignado al proyecto. • Que el tiempo de uso de la GPU asignado al proyecto sea suficiente para entrenar los modelos, pero no para lograr los objetivos de rendimiento. • Que otro proyecto más prioritario necesite usar la GPU cuando esté asignada a este proyecto.					
Probabilidad	**Impacto**				**Impacto Total**	**Respuestas**
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		
Muy Alta	Muy Bajo	Alto	Alto	Muy Bajo	0,55	Aceptar el riesgo, el proyecto se debe realizar con los recursos asignados

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Probabilidad revisada *Fecha*	Impacto				Impacto Total	Respuestas
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		

Riesgos derivados de éste:

Riesgo de que el proyecto no se termine antes del 01/06/2024

Riesgo de que no se logre el objetivo de rendimiento del modelo

Riesgo residual:

Que una avería de la GPU impida que pueda ser utilizada durante el periodo de tiempo asignado al proyecto.

Evaluación Cuantitativa:

Se ha utilizado la técnica de juicio de expertos para estimar cuánto se puede retrasar el entrenamiento de los modelos:

- Optimista: 30 horas
- Más probable: 38 horas
- Pesimista: 56 horas

Retraso estimado (distribución beta) = (Optimista + 4 * Más probable + Pesimista) / 6 = 39,67 horas

Coste del entrenamiento = 0,82 € / hora

Por tanto, el coste total del retraso es de 19,67 * 0,82 = 32,53 €.

Pan de Contingencia:

1. Si hubiese retrasos en el proyecto o modificaciones en el calendario de uso de la GPU provocados por otros proyectos u otros factores, se deberá revisar el calendario para ajustarlo durante las reuniones mensuales con el director.
2. Asignar una reserva del 20 % del tiempo estimado de uso de la GPU por el

Presupuesto para contingencias:

Item	Concepto	Asignación (€)
1	Reserva de tiempo de GPU	33,46 €
	TOTAL	33,46 €

MIW - DGPW - HR **Hoja de Datos del Riesgo**

proyecto. a. 20 % de 204 horas = 40,8 horas. 40,8 * 0,82 € / hora (coste de uso de la GPU) = 33,46 €	**Planificación temporal de las contingencias:** Las reuniones mensuales se realizarán según la planificación del proyecto.

Comentarios:

N/A

Monitorización:

Se monitorizan los cambios en el calendario de uso de la GPU, el retraso de las tareas previas al entrenamiento y el consumo de la reserva.

Los indicadores medidos se valoran:

	Valor del riesgo				
INDICADORES	***Muy alto***	***Alto***	***Medio***	***Bajo***	***Muy Bajo***
Indicador 1: Número de modificaciones del calendario de uso de la GPU que afectan al proyecto	> 4	3	2	1	0
Indicador 2: Retraso de las tareas previas al entrenamiento	> 10 días	8 días	4 días	2 días	0 días
Indicador 3: Consumo de reserva	> 80 %	60 %	40 %	20 %	20 %

El riesgo se valorará como el más alto valor de cualquier indicador.

Indicadores:

Indicador 1: Número de modificaciones del calendario de uso de la GPU que afectan al proyecto. • Total de las veces que se ha realizado alguna modificación del calendario de uso de la GPU que haya afectado a las fechas que tenía asignadas este proyecto.	**Evaluación:** Se evalúa cada vez que se realice una modificación del calendario de uso de la GPU.
Indicador 2: Retraso de las tareas previas al entrenamiento. • Retraso acumulado de las tareas previas a la tarea 34.	**Evaluación:** Se evalúa cada vez que se complete una tarea.

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Indicador 3: Consumo de reserva • Tiempo (horas) consumido de la reserva de tiempo de uso de la GPU.	**Evaluación:** Se evalúa cada vez que sea necesario asignar tiempo de uso de GPU adicional al proyecto.

D.2 Hoja del riesgo “Calidad de los datos”

A continuación, se incluye una copia de la hoja de seguimiento del riesgo “Calidad de los datos”.

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Título del Proyecto: Redes siamesas de transformers para el reconocimiento de programadores a partir de su código fuente

Fecha: 04/11/2023

ID:	**Nombre:**					
2	Calidad de los datos					
Descripción: La calidad de los datos de entrenamiento tiene una gran influencia en el rendimiento del modelo. Puede que los conjuntos de datos disponibles no tengan la calidad suficiente para alcanzar los objetivos de rendimiento.						
Categoría(s) de riesgo: Requisitos, Prestaciones y fiabilidad, Proveedores						
Status: Activo	**Causas del Riesgo:** • Que no existan conjuntos de datos con la calidad necesaria. • Que no sea posible elaborar nuevos conjuntos de datos con la calidad necesaria.					
Probabilidad	**Impacto**				**Impacto Total**	**Respuestas**
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		
Media	Muy Bajo	Muy Bajo	Alto	Muy Bajo	0,28	Mitigar el riesgo, planificar una sesión de selección de datos para obtener un conjunto de códigos fuente que sea representativo de los que elaboran los programadores.

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Probabilidad revisada *Fecha*	**Impacto**				**Impacto Total**	**Respuestas**
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		

Riesgos derivados de éste:

Riesgo de que el proyecto no se termine antes del 01/06/2024.

Riesgo de que no se logre el objetivo de rendimiento del modelo.

Riesgo residual:

Que no existan muestras suficientes como para elaborar un conjunto de datos con la calidad necesaria para cumplir los objetivos del proyecto.

Evaluación Cuantitativa:

Se ha utilizado la técnica de juicio de expertos para estimar cuánto se puede retrasar el proyecto en caso de que sea necesario elaborar un nuevo conjunto de datos:

- Optimista: 12 horas
- Más probable: 16 horas
- Pesimista: 24 horas

Retraso estimado (distribución beta) = (Optimista + 4 * Más probable + Pesimista) / 6 = 16,67 horas

Coste del perfil de científico de datos = 60 € / hora

Por tanto, el coste total del retraso es de 16,67 * 60 = 1.000 €.

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Pan de Contingencia:

1. Establecer una tarea con una duración estimada de 12 horas para elaborar un conjunto de datos de entrenamiento en caso de que no existiera uno con calidad suficiente.
2. Asignar la tarea al perfil científico de datos.

Presupuesto para contingencias:

Item	Concepto	Asignación (€)
1	Reserva de tiempo para elaboración del conjunto de datos	720 €
	TOTAL	720 €

Planificación temporal de las contingencias:

Tarea 31 con una duración estimada de 12 horas.

Comentarios:

N/A

Monitorización:

Se monitorizará el tiempo consumido de la tarea 31:

	Valor del riesgo				
INDICADORES	***Muy alto***	***Alto***	***Medio***	***Bajo***	***Muy Bajo***
Indicador 1: Tiempo consumido de la tarea 31	> 12 h	12 h	8 h	4 h	0 h

Indicadores:

Indicador 1: Tiempo consumido de la tarea 31	**Evaluación:** Se evalúa al inicio de cada día desde que se comience la tarea.

D.3 Hoja del riesgo “Desarrollo novedoso”

A continuación, se incluye una copia de la hoja de seguimiento del riesgo “Desarrollo novedoso”.

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Título del Proyecto: Redes siamesas de transformers para el reconocimiento de programadores a partir de su código fuente

Fecha: 04/11/2023

ID:	**Nombre:**					
6	Desarrollo novedoso					
Descripción:						
Al tratarse de un desarrollo novedoso, es probable que se produzcan cambios en el alcance y los requisitos del proyecto.						
Categoría(s) de riesgo:						
Requisitos, Planificación						
Status:	**Causas del Riesgo:**					
Activo	• Que cambie alguno de los requisitos funcionales del proyecto. • Que se decidan cambiar las técnicas o en la metodología que se ha planificado utilizar para cumplir los objetivos de proyecto.					
Probabilidad	**Impacto**				**Impacto Total**	**Respuestas**
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		
Media	Bajo	Bajo	Crítico	Muy Bajo	0,45	Aceptar el riesgo, es inherente de todos los proyectos innovadores

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Probabilidad revisada *Fecha*	Impacto				Impacto Total	Respuestas
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		

Riesgos derivados de éste:

Riesgo de que el proyecto no se termine antes del 01/06/2024

Riesgo de que no se cumplan los requisitos funcionales del proyecto

Riesgo residual:

Que no sea posible cumplir de los requisitos del proyecto con los conocimientos científicos y técnicos actuales.

Evaluación Cuantitativa:

Al no poder conocerse cómo puede afectar este riesgo al proyecto hasta que no se haya avanzado en su desarrollo, no se puede realizar una evaluación cuantitativa más allá del impacto total estimado de 0,45.

Pan de Contingencia:

1. Si es necesario modificar los requisitos del proyecto o su alcance, será necesario consultarlo inmediatamente con el director del trabajo.
2. Del mismo modo, si se da la necesidad de cambiar algunas de las técnicas planificadas o la metodología, se deberá consultar la idoneidad del cambio con el director del trabajo.

Presupuesto para contingencias:

N/A

Planificación temporal de las contingencias:

N/A

Comentarios:

N/A

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Monitorización:

Se cambios en los requisitos del proyecto y en las técnicas y metodología.

Los indicadores medidos se valoran:

	Valor del riesgo				
INDICADORES	***Muy alto***	***Alto***	***Medio***	***Bajo***	***Muy Bajo***
Indicador 1: Número de cambios en los requisitos del proyecto	> 4	3	2	1	0
Indicador 2: Número de cambios en las técnicas y metodología	> 4	3	2	1	0

El riesgo se valorará como el más alto valor de cualquier indicador.

Indicadores:

Indicador 1: Número de cambios en los requisitos del proyecto.	**Evaluación:** Se evalúa cada vez que se produzca un cambio en los requisitos del proyecto.
Indicador 2: Número de cambios en las técnicas y metodología.	**Evaluación:** Se evalúa cada vez que se produzca un cambio en las técnicas y metodología.

D.4 Hoja del riesgo “Complejidad del modelo”

A continuación, se incluye una copia de la hoja de seguimiento del riesgo “Complejidad del modelo”.

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Título del Proyecto: Redes siamesas de transformers para el reconocimiento de programadores a partir de su código fuente

Fecha: 04/11/2023

ID:	**Nombre:**					
7	Complejidad del modelo					
Descripción: La arquitectura de modelo seleccionada para desarrollar el proyecto, Transformer, es compleja y novedosa. Por tanto, se pueden producir retrasos en su implementación.						
Categoría(s) de riesgo: Estimación, Planificación						
Status: Activo	**Causas del Riesgo:** • Complejidad de la arquitectura Transformer. • Inmadurez relativa de dicha arquitectura.					
Probabilidad	**Impacto**				**Impacto Total**	**Respuestas**
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		
Medio	Medio	Medio	Medio	Alto	0,28	Aceptar el riesgo, el objetivo del proyecto implica emplear modelos complejos

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Probabilidad revisada *05/02/2024*	Impacto				Impacto Total	Respuestas
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		
Medio	Medio	Medio	Alto	Muy Bajo	0,28	Aceptar el riesgo, el objetivo del proyecto implica emplear modelos complejos

Riesgos derivados de éste:

Riesgo de que el proyecto no se termine antes del 01/06/2024

Riesgo residual:

N/A

Evaluación Cuantitativa:

Se ha utilizado la técnica de juicio de expertos para estimar cuánto se puede retrasar la implementación del modelo:

- Optimista: 10 horas
- Más probable: 16 horas
- Pesimista: 22 horas

Retraso estimado (distribución beta) = (Optimista + 4 * Más probable + Pesimista) / 6 = 16 horas

Coste del perfil de ingeniero de datos = 50 € / hora

Por tanto, el coste total del retraso es de 16 * 50 = 800 €.

Pan de Contingencia:

1. Si se produjesen retrasos durante la implementación del modelo, se dispondrá de 10 horas de margen.
2. Si se superan esas 10 horas se llevará a cabo una revisión de la planificación.

Presupuesto para contingencias:

Item	Concepto	Asignación (€)
1	Reserva de tiempo para la implementación del modelos	500 €
	TOTAL	500 €

MIW - DGPW - HR **Hoja de Datos del Riesgo**

	Planificación temporal de las contingencias: Incremento de 5 horas en la estimación de las tareas 33 y 37, respectivamente.

Comentarios:

N/A

Monitorización:

Se monitorizará el retraso en las tareas 33 y 37:

	Valor del riesgo				
INDICADORES	***Muy alto***	***Alto***	***Medio***	***Bajo***	***Muy Bajo***
Indicador 1: Retraso de la tarea 33	> 5 horas	4 horas	2,5 horas	1,5 horas	0,5 horas
Indicador 2: Retraso de la tarea 37	> 5 horas	4 horas	2,5 horas	1,5 horas	0,5 horas

El riesgo se valorará como el más alto valor de cualquier indicador.

Indicadores:

Indicador 1: Retraso de la tarea 33.	**Evaluación:** Se evalúa al comienzo de cada día desde el inicio de la tarea 33 hasta su finalización.
Indicador 2: Retraso de la tarea 37.	**Evaluación:** Se evalúa al comienzo de cada día desde el inicio de la tarea 37 hasta su finalización.

D.5 Hoja del riesgo "Problemas de convergencia"

A continuación, se incluye una copia de la hoja de seguimiento del riesgo "Problemas de convergencia".

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Título del Proyecto: Redes siamesas de transformers para el reconocimiento de programadores a partir de su código fuente

Fecha: 04/11/2023

ID:	**Nombre:**
8	Problemas de convergencia

Descripción:

Puede que sea necesario realizar numerosas pruebas para encontrar los hiperparámetros con los que mejor rendimiento obtenga el modelo.

Categoría(s) de riesgo:

Planificación , Estimación

Status:

Activo

Causas del Riesgo:

- Que sea necesario realizar numerosas pruebas para encontrar los hiperparámetros con los que mejor rendimiento obtenga el modelo.
- Que no se disponga de los recursos necesarios para lograr que el entrenamiento del modelo converja.

Probabilidad	**Impacto**				**Impacto Total**	**Respuestas**
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		
Baja	Bajo	Bajo	Alto	Crítico	0,55	Aceptar el riesgo, es una consecuencia de trabajar con modelos complejos

MIW - DGPW - HR **Hoja de Datos del Riesgo**

Probabilidad revisada *16/02/2024*	Impacto				Impacto Total	Respuestas
	Presupuesto	**Planificación**	**Alcance**	**Calidad**		
Media	Medio	Medio	Bajo	Alto	0,28	Aceptar el riesgo, es una consecuencia de trabajar con modelos complejos

Riesgos derivados de éste:

Riesgo de que el proyecto no se termine antes del 01/06/2024

Riesgo de que no se logre el objetivo de rendimiento del modelo

Riesgo residual:

N/A

Evaluación Cuantitativa:

Se ha utilizado la técnica de juicio de expertos para estimar cuánto se puede retrasar la búsqueda de hiperparámetros:

- Optimista: 18 horas
- Más probable: 34 horas
- Pesimista: 67 horas

Retraso estimado (distribución beta) = (Optimista + 4 * Más probable + Pesimista) / 6 = 36,83 horas

Coste del entrenamiento = 0,82 € / hora

Por tanto, el coste total del retraso es de 36,83 * 0,82 = 30,20 €.

Pan de Contingencia:

1. Si se produjeran retrasos durante la selección de hiperparámetros, se dispondrá de 18 horas de margen.
2. Si se superan esas 18 horas se llevará a cabo una revisión de la planificación.

Presupuesto para contingencias:

Item	Concepto	Asignación (€)
1	Reserva de tiempo de GPU	30,20 €
	TOTAL	30,20 €

MIW - DGPW - HR **Hoja de Datos del Riesgo**

	Planificación temporal de las contingencias: Se ha incrementado en 12 y 6 horas la planificación de las tareas 34 y 38, respectivamente.

Comentarios:

N/A

Monitorización:

Se monitorizará el retraso en las tareas 34 y 38:

	Valor del riesgo				
INDICADORES	***Muy alto***	***Alto***	***Medio***	***Bajo***	***Muy Bajo***
Indicador 1: Retraso de la tarea 34	> 12 horas	8 horas	6 horas	2 horas	0 horas
Indicador 2: Retraso de la tarea 38	> 6 horas	4 horas	3 horas	1 hora	0 horas

El riesgo se valorará como el más alto valor de cualquier indicador.

Indicadores:

Indicador 1: Retraso de la tarea 34.	**Evaluación:** Se evalúa al comienzo de cada día desde el inicio de la tarea 35 hasta su finalización.
Indicador 2: Retraso de la tarea 38.	**Evaluación:** Se evalúa al comienzo de cada día desde el inicio de la tarea 39 hasta su finalización.