Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Escuela de Ingeniería Informática
Universidad de Oviedo

Alejandro Pato Tellada

Identificación del lenguaje de programación a partir del código fuente

Directores

Óscar Rodríguez Prieto
Francisco Ortín Soler

Trabajo Fin de Grado

Trabajo presentado en cumplimiento de los requisitos para el Grado de Ingeniería Informática del software en el
Grupo de investigación *Computacional Reflection*
del Departamento de Informática

Universidad de Oviedo

Resumen

Desde sus inicios, el desarrollo software se ha enfrentado a tareas cada vez más complejas y específicas, dando lugar a la constante aparición de nuevos lenguajes de programación capaces de hacerles frente. En la actualidad, las crecientes exigencias de calidad de la industria requieren de soluciones software cada vez más amplias y complejas, lo que implica el uso de múltiples lenguajes de programación. Como resultado, los desarrolladores se ven obligados a conocer y trabajar con diferentes lenguajes. En este contexto, poder identificar automáticamente el lenguaje de programación en el que está escrito el código fuente ofrece distintas ventajas. De entre ellas, destacan el resaltado de sintaxis y formato adecuado del código en IDEs, editores y otros medios como foros o chats; el refinamiento de los resultados emitidos por motores de búsqueda; la actualización de metadatos en plataformas de almacenamiento y compartición de código y la detección de código embebido.

Para cubrir esta necesidad, surgen los sistemas de identificación automática de código fuente, cuyo objetivo es determinar con precisión el lenguaje de programación al que pertenece el código. En la actualidad, existen varios trabajos que resuelven este problema con una exactitud (*accuracy*) muy alta (97-98%), pero necesitan ficheros enteros para poder realizar las inferencias (predicciones). Esto supone una limitación muy grande ya que, en los escenarios descritos, las unidades de código que interesa clasificar pueden ser mucho más pequeñas que un fichero entero de código. Por ello, también existen trabajos cuyo objetivo es la identificación de fragmentos (*snippets*) de código fuente. Al no disponer de tanta información, estos trabajos no tienen una exactitud tan alta (varía entre el 75 y el 93%) y, además, todos ellos necesitan un fragmento de código de varias líneas para poder llevar a cabo su función con la fidelidad esperada.

Con el fin de superar las limitaciones de los sistemas actuales, este trabajo estudia en qué medida es posible determinar el lenguaje de programación, con tan sólo analizar una línea de código. Para su construcción, fue necesaria la confección de un extenso corpus de ficheros de código fuente, descargados de GitHub mediante el desarrollo de un rastreador (*crawler*). Además, se implementó un verificador automático para corroborar el lenguaje de los ficheros descargados y un procesador de lenguajes capaz de detectar y limpiar determinadas líneas de estos ficheros; aquéllas que contienen lenguaje natural. Con todo ello, se generó un conjunto de datos (*dataset*) representando las líneas de código como una sucesión de caracteres, con el que entrenamos numerosos modelos de aprendizaje automático. Los modelos con mayor rendimiento fueron aquéllos basados en redes neuronales, siguiendo la arquitectura del perceptrón multicapa. Dichos modelos fueron construidos y refinados mediante la búsqueda y ajuste de sus distintos hiperparámetros.

El resultado es un sistema capaz de identificar una selección de los 21 lenguajes de programación de más utilizados según tres rankings, consiguiendo un 92.18% de exactitud y analizando una única línea de código de al menos 10 caracteres. Adicionalmente, este trabajo también aporta un corpus de 19,76 GB de tamaño, formado por 1,47 millones de ficheros de código fuente pertenecientes a estos

lenguajes. Finalmente, se comparó el rendimiento de los modelos basados en redes neuronales con otros modelos de aprendizaje automático y con una herramienta actual de identificación automática de código, seleccionada tras un análisis de los distintos trabajos relacionados. Los resultados obtenidos muestran que los sistemas actuales, entrenados con fragmentos de código de varias líneas y a nivel de palabra, no son capaces de determinar con nuestra exactitud el lenguaje de programación de una sola línea de código de fuente.

Palabras clave

Lenguaje de programación, sistema de identificación de código fuente, exactitud, fragmento de código, corpus de ficheros, rastreador, conjunto de datos, aprendizaje automático, redes neuronales, perceptrón multicapa, hiperparámetro

Abstract

Since its inception, software development has faced increasingly complex and specific tasks, leading to the constant evolution of new programming languages. Nowadays, the growing levels of quality demand extensive and complex software solutions, which involve the use of multiple programming languages. As a result, developers must learn and work with different languages. Thus, the automatic identification of the programming language from source code would offer various advantages. Among these advantages are syntax highlighting and proper formatting of code in IDEs, editors, and other mediums like forums or chats; refinement of search engine results; updating metadata in code storage and sharing platforms; and detection of embedded code.

To fulfil this need, automatic source code identification systems have emerged. They are aimed at accurately determining the programming language a code excerpt is written in. Currently, several works tackle this problem with very high accuracy (97-98%), but they require entire files to make inferences (predictions). This limitation is significant since, in the described scenarios, the code units to be classified may be smaller than a whole source file. Consequently, there are other works focused on identifying snippets of source code. Due to the use of smaller pieces of information, these works do not achieve such a high accuracy (they range from 75% to 93%). Moreover, all of them receive code snippets as input. These snippets consist of several lines of code.

To overcome the limitations of the current works, we present a machine learning system that can determine the programming language by analysing just a single line of code. To train our predictive model, we created of an extensive corpus of source code files with a crawler we implemented that downloads code from GitHub. Additionally, an automatic verifier was implemented to confirm the language of the downloaded files, along with a language processor capable of detecting and cleaning certain lines containing natural language. With these components, a dataset that represents code lines as character sequences was generated. Different machine learning models were trained. Those with the highest performance were based on neural networks. Namely, they follow the multilayer perceptron architecture. They were constructed and refined through the search and adjustment of various hyperparameters.

The result is a system capable of identifying 21 of the most commonly used programming languages according to three different programming language rankings. A 92.18% accuracy is achieved by just analysing a single line of code consisting of at least 10 characters. Additionally, this work provides a 19.76 GB corpus of 1.47 million source code files written in these 21 languages. The results obtained show that the existing systems, trained with code snippets of multiple lines at the word level, are far from being capable of determining the programming language of a single line of source code with our level of accuracy.

Keywords

Programming language, source code identification system, accuracy, code snippet, file corpus, crawler, dataset, machine learning, neural networks, multilayer perceptron, hyperparameter

Índice de contenido

Índice de Figuras

Índice de Tablas

1. Introducción

1.1. Motivación

Desde la creación del primer programa informático, estos han ido afrontando cada vez más tareas y más complejas. Tareas que, debido a su complejidad, en ocasiones han ido necesitando soluciones más específicas y novedosas. A consecuencia de esto, constantemente nacen nuevos lenguajes de programación, para solucionar una determinada problemática o superar las limitaciones de las soluciones más primitivas. Esto, unido a las crecientes exigencias de calidad del mundo actual, hace que cada vez sea más común encontrar proyectos y soluciones industriales software multilenguaje y de mayor envergadura. Por todo esto, los desarrolladores actuales se ven obligados a conocer y trabajar con varios lenguajes de programación distintos en este tipo de proyectos. En este contexto, surge la necesidad de identificar el lenguaje de programación al que pertenece cierta porción de código fuente (*snippet*), lo que podría aportar distintos beneficios para el proceso de desarrollo software.

Por un lado, a la hora de escribir código, la determinación del lenguaje de programación permitiría a los distintos IDEs y editores llevar a cabo resaltado de sintaxis, además de formato y sangrado adecuados según el caso. En muchas ocasiones, esta identificación ya se realiza de forma determinista y únicamente en base a la extensión de los distintos ficheros. Sin embargo, en muchos escenarios esto no es posible, como cuando se escribe código en un fichero que todavía no se ha guardado, o cuando se comparten porciones de código en chats, foros, aplicaciones de mensajería instantánea, servicios de correo electrónico o en sitios web de preguntas y respuestas *(Q&A*) como StackOverflow.

Por otro lado, la identificación automática de código es muy útil para los motores de búsqueda. Con herramientas de identificación automática de código, es posible determinar cuándo una página web contiene código fuente, y distinguirlo así de otro contenido textual como el lenguaje natural. Además, es posible identificar a qué lenguaje pertenecen estos pedazos de código. Con esta información, los motores de búsqueda pueden dar prioridad a páginas web con este tipo de contenido, en caso de detectar que el usuario está realizando una consulta en busca de código fuente, o relacionada con un determinado lenguaje de programación.

Otro beneficio de la identificación de código es la actualización automática de los metadatos en repositorios de código fuente como GitHub o Bitbucket. Estos metadatos se suelen calcular de forma determinista en base a la extensión de los ficheros, pero en ocasiones, los desarrolladores pueden no especificar la extensión de algunos ficheros o confundirse al especificarla. Además, algunas extensiones están asociadas con más de un lenguaje de programación, como *.h* que puede estar asociada con programas en C, C++ o Objective-C. Adicionalmente, también se pueden usar para calcular estos metadatos en sitios, como las ya mencionadas plataformas *Q&A,* en los que el código no se comparte subiendo los ficheros, sino mediante *snippets*. Finalmente, independientemente de su extensión, un fichero puede contener código embebido de otros lenguajes, como es el caso de los ficheros *.html,* que pueden contener código CSS

o JavaScript. Con los sistemas de identificación se pueden detectar los distintos lenguajes de programación que contiene un fichero, analizando los distintos *snippets* que lo componen.

1.1.1 Sistemas de identificación automática de código fuente

Para cubrir todos estos escenarios, surgen los sistemas de identificación automática de código fuente, cuyo objetivo es determinar con precisión el lenguaje de programación al que pertenece cierto *snippet.* Así, en la actualidad, existen varios trabajos que abordan este problema como [VanDam2016], [Gilda2017] y [Kiyak2020], que consiguen determinar el lenguaje de programación con una exactitud (*accuracy*) muy alta (97-98%), pero necesitan ficheros enteros para poder realizar predicciones. Esto supone una limitación muy grande para la mayoría de los escenarios descritos, donde las unidades de código que interesa clasificar son mucho más pequeñas, típicamente *snippets.* Por ello, también existen trabajos cuyo objetivo es la identificación de fragmentos de código fuente como [Alreshedy2018], [Alreshedy2020] y [Yang2021] o herramientas profesionales como [Guesslang2023]. Estos trabajos, al no disponer de tanta información para realizar predicciones, no tienen una exactitud tan alta como los anteriores (varía entre el 75 y el 93% dependiendo del sistema). Sin embargo, todos ellos necesitan un fragmento de código de varias líneas para poder llevar a cabo su función con la fidelidad esperada.

Para superar esta limitación de los sistemas actuales, presentamos este trabajo, donde nos centramos en la identificación de *snippets* lo más cortos posibles, de tan sólo una línea de código fuente. Poder identificar el lenguaje de fragmentos tan cortos, tiene una importante utilidad en los escenarios descritos. Por ejemplo, para detectar el lenguaje en el que un desarrollador está escribiendo lo antes posible; para que un motor de búsqueda identifique y clasifique cualquier porción de código en un documento de texto, por pequeña que sea; o para poder detectar fragmentos de código embebido de prácticamente cualquier tamaño. También pretendemos estudiar hasta qué punto los sistemas actuales entrenados con *snippets* son capaces de realizar predicciones correctas con tan poca cantidad de código fuente.

Cabe destacar, que el problema de identificar el lenguaje para líneas de código tan pequeñas no se puede resolver con una exactitud del 100%. Esto se debe a que existen situaciones donde una misma línea de código puede pertenecer a distintos lenguajes de programación con sintaxis cercanas.

```
int index = 0;
```

Figura 1. Línea de código fuente válida para múltiples lenguajes de programación.

Por ejemplo, el fragmento de código que mostramos en la Figura 1, constituye una línea de código muy común, que sólo consta de 5 términos y 14 caracteres. Ningún sistema ni programador experto podría determinar un único lenguaje de programación para dicha

línea, ya que es perfectamente válida y utilizada en programas de C, C++, C#, Java y Objective-C, entre otros.

Finalmente, cabe destacar que, en la última década, ha habido un crecimiento importante en el uso de repositorios de código fuente, como GitHub, SourceForge, BitBucket y CodePlex [Ortin2016]. Por ejemplo, en el caso de GitHub, en julio de 2010 albergaba 1 millón de repositorios [GitHub2023b] y hoy en día alberga más de 330 millones de repositorios, de más de 100 millones de desarrolladores y 4 millones de organizaciones [GitHub2023c]. La gran cantidad de proyectos de código abierto disponibles en dichos repositorios aportan un volumen de información sin precedentes, que puede explotarse para la construcción de modelos de aprendizaje automático, dando lugar a un nuevo campo de investigación denominado BigCode [Ortin2016]. De hecho, diferentes corpus de código abierto ya se han utilizado para mejorar escenarios de desarrollo de software, como la traducción entre lenguajes [Aggarwal2015], la solución de errores [Bhatia2016] y la documentación [Barone2017], autocompletado [Bhoopchand2016] y generación [GitHub2023a] automática de código fuente.

Para este trabajo, pretendemos aprovecharnos de esta gran cantidad de repositorios públicos de código fuente, para construir nuestro propio corpus de ficheros de distintos lenguajes. Este corpus deberá ser lo suficientemente grande y diverso, y deberá limpiarse y tratarse apropiadamente para poder resolver la exigente tarea de identificar el lenguaje de programación de una única línea de código con la máxima eficacia posible.

1.1. Contribución

La principal contribución de este trabajo es un sistema que, utilizando aprendizaje automático, es capaz de identificar a qué lenguaje de programación pertenece una línea de código fuente. Este sistema es capaz de identificar 21 de los lenguajes más usados (Assembly[1], C, C++, C#, CSS, Go, HTML, Java, JavaScript, Kotlin, Matlab, Perl, PHP, Python, R, Ruby, Scala, SQL, Swift, TypeScript, Unix Shell) con una exactitud del 92,18%.

Este sistema, basado en un modelo de red neuronal, sigue la arquitectura de un perceptrón multicapa que consiste en una capa de entrada, una capa de incrustaciones (*embeddings*), tres capas ocultas y una capa *softmax* de salida. La entrada es una línea de código limpia, sin lenguaje natural, y tokenizada a nivel de carácter. Mientras que la salida es un vector de 21 números reales, que representa las probabilidades de que la entrada pertenezca a cada uno de los lenguajes de programación mencionados.

Otra contribución de este trabajo es un corpus con un tamaño de 19,76 GB, constituido por 1,47 millones ficheros de código fuente pertenecientes a los 21 lenguajes de programación mencionados. Para su construcción, se implementó un rastreador (*crawler*) capaz de descargar ficheros procedentes de GitHub, y se verificó mediante una

[1] En este proyecto, identificamos como el lenguaje Assembly a cualquier tipo de ensamblador para microprocesadores comunes. Todos ellos tienen ciertas características similares entre sí y suficientemente distintas al resto de lenguajes de programación analizados, por los que los clasificamos todos ellos bajo la etiqueta de Assembly.

metodología semiautomática que estos ficheros se correspondieran con el lenguaje de programación esperado. A partir de este corpus, eliminamos construcciones que contienen lenguaje natural (como cadenas multilínea y comentarios), lo dividimos en líneas no vacías, y representamos cada una de ellas como una sucesión de caracteres. De esta forma, construimos un conjunto de datos (*dataset*) de propósito específico finalmente utilizado para entrenar distintos modelos de aprendizaje automático para la clasificación de líneas de código fuente.

1.2. Estructura del documento

Este documento está estructurado de la siguiente manera. El siguiente apartado enumera los objetivos llevados a cabo para la realización de este trabajo. El estudio de los trabajos relacionados está contenido en el apartado 3. En el apartado 4 se realiza una descripción detallada del sistema y sus distintos módulos. La metodología seguida se describe en el apartado 5. El apartado 6 muestra los resultados obtenidos y evalúa distintos modelos de aprendizaje automático comparando su rendimiento con una herramienta profesional actual. Las conclusiones se discuten en el apartado 7. En el apartado 8 se plantea el trabajo futuro. La planificación y el presupuesto se muestran en el apartado 9 y finalmente, el apartado 10 contiene las referencias bibliográficas.

2. Fijación de los objetivos

Este trabajo ha conseguido obtener los siguientes objetivos:

1. Diseñar e implementar un *crawler* capaz de descargar ficheros de código fuente de GitHub según su extensión.
2. Crear un corpus a partir de ficheros de código fuente pertenecientes a los 21 lenguajes de programación más utilizados según tres rankings de utilización de lenguajes de programación.
3. Diseñar e implementar un sistema de verificación semiautomática. Con el fin de validar si un conjunto de ficheros de código fuente han sido escritos, verdaderamente, en el lenguaje de programación esperado.
4. Implementar un procesador de lenguajes capaz de limpiar ficheros de código fuente. El objetivo es eliminar las distintas construcciones de cada lenguaje que contienen lenguaje natural, como cadenas multilínea y comentarios, así como las líneas que no alcanzan una determinada longitud mínima.
5. Construir un *dataset* de propósito específico formado por líneas de código fuente divididas en caracteres, partiendo del corpus del Objetivo 2.
6. Construir distintos modelos de aprendizaje automático basados en redes neuronales mediante la búsqueda y selección (*tunning*) de hiperparámetros, encontrando los modelos con mayor rendimiento.
7. Evaluar y medir del rendimiento de cada uno de los modelos desarrollados usando las métricas de exactitud, valor-F, exhaustividad y precisión, y analizando las clasificaciones emitidas mediante una matriz de confusión.
8. Estudio de los sistemas de clasificación de código fuente existentes y comparación de nuestros modelos basados en redes neuronales con uno de ellos, y con otros modelos de aprendizaje automático, sobre el problema de clasificación planteado.

2.1. Posibles ámbitos de aplicación

Basándonos en las necesidades expuestas en la motivación y en su contribución, este sistema puede ayudar a ofrecer las siguientes soluciones:

Ofrecer resaltado de sintaxis y formato en escenarios donde la especificación del lenguaje de programación no se contempla, como puede ser en chats, en plataformas de mensajería instantánea o en servicios de correo electrónico. Estos escenarios no fueron pensados, originalmente, para la codificación y la identificación automática de código es la única alternativa para averiguar a qué lenguaje pertenecen los fragmentos escritos. Además, existen otros escenarios en los que, aunque ya se contempla la identificación del lenguaje, como en editores de texto o en IDEs, la identificación automática de código podría facilitar el resaltado de sintaxis. De esta forma, no habría necesidad de especificar el lenguaje manualmente o a través de la extensión del archivo, pudiendo adelantar esta identificación del lenguaje, más teniendo en cuenta que nuestro sistema sólo necesita una línea de código para emitir inferencias.

Mejorar motores de búsqueda basados en texto. Estos motores de búsqueda funcionan seleccionando y ordenando documentos en base a la similitud entre el contenido de éstos y las palabras introducidas por el usuario en la consulta. En primer lugar, se podría detectar fácilmente cuándo el usuario especifique algún lenguaje de programación en la consulta, por medio, por ejemplo, de palabras clave asociadas a cada lenguaje. Posteriormente, el motor de búsqueda puede filtrar y priorizar aquellos documentos en los que, además de tener similitud con la consulta introducida, se identifiquen porciones de código pertenecientes al lenguaje buscado, dotándolos de mayor precisión.

Agregar y corregir metadatos acerca de los lenguajes de programación utilizados en plataformas de preguntas y respuestas orientadas a la programación, como pueden ser StackOverflow o CodeProject. En estos sitios, cada pregunta se etiqueta manualmente con los lenguajes implicados para una mejor organización y búsqueda. La identificación automática de los lenguajes de las líneas de código subidas permitiría a estas plataformas etiquetar los lenguajes utilizados cuando el usuario se olvide de hacerlo e, incluso corregir las etiquetas si el usuario se equivocara.

Crear y verificar metadatos acerca de los lenguajes de programación utilizados en un repositorio de código fuente. Generalmente, en los repositorios de código, los metadatos de los lenguajes utilizados se extraen a partir de las distintas extensiones de los ficheros. Con el sistema propuesto se podría identificar el lenguaje de programación de ficheros sin extensión o con algún error en ésta, de forma que no se corresponda con el lenguaje contenido. Adicionalmente, es posible identificar distintos lenguajes de programación contenidos en un mismo fichero de código fuente. Con el sistema propuesto se podría detectar código embebido de tan sólo una línea de longitud, como puede ser código JavaScript o CSS dentro de un fichero HTML. De esta forma, un mismo fichero podría ser etiquetado con más de un único lenguaje, independientemente de su extensión, y se podrían refinar los metadatos existentes en repositorios con este tipo de ficheros.

3. Trabajo relacionado

Esta sección describe algunos de los trabajos, ya existentes, que buscan identificar el lenguaje de programación a partir de código fuente. Primero nos centraremos en aquellos trabajos que guardan mayor relación con este documento, describiéndolos en mayor profundidad. Finalmente, mencionaremos aquellos trabajos que, si bien comparten nuestro mismo objetivo, no tienen tanta relación.

[Alreshedy2018] proponen *Source Code Classifier* (SCC), una herramienta capaz de identificar el lenguaje de programación al que pertenece un *snippet* de al menos 2 líneas de código. Para ello, emplean un modelo basado en redes bayesianas multinomiales con el cual logran un 75% de exactitud identificando 21 lenguajes de programación. Este modelo utiliza un vocabulario a nivel de palabra representado mediante el método *bag of words*, el cual no tiene en cuenta el orden de las palabras. Para el entrenamiento, SCC utiliza un *dataset* formado por 237.787 *snippets* descargados de StackOverflow basándose en su etiqueta de lenguaje, asumiendo su veracidad, sin ningún otro tipo de verificación. Este es un *dataset* no balanceado, donde todos los lenguajes cuentan con el mismo número de *snippets*, excepto *Lua* y *Markdown*.

[Yang2021] presentan un sistema, DeepSCC, capaz de identificar el lenguaje de programación al que pertenece un *snippet* de al menos 2 líneas de código. Para ello emplean RoBERTa [Liu2019], un modelo pre-entrenado basado en *transformers* [Vaswani2017] con el que mediante *fine-tuning* logran una exactitud del 87,4% identificando 19 lenguajes de programación. Para mitigar el problema de los términos fuera del vocabulario (*OOV*), utilizan un vocabulario representado mediante *byte-pair encoding*. Un método que mezcla la representación a nivel de palabra y a nivel de carácter. Para el entrenamiento, DeepSCC utiliza el *dataset* de [Alreshedy2018], aunque descartan los *snippets* correspondientes a HTML y a Markdown, los primeros por contener código embebido perteneciente a otros lenguajes y los segundos por ser insuficientes.

[Guesslang2023] es una herramienta capaz de identificar el lenguaje de programación al que pertenece un *snippet* de varias líneas de código. Para ello emplea un modelo híbrido basado en una red neuronal profunda y un clasificador lineal que, en conjunto y mediante el ajuste de hiperparámetros, logran una precisión del 93,4% identificando hasta 54 lenguajes de programación. Sin embargo, sus diseñadores identifican limitaciones como la dificultad para clasificar de códigos fuente que se encuentren en la frontera entre dos lenguajes. Como, por ejemplo, código fuente válido en C que también puede ser válido en C++, o código fuente JavaScript, que siempre será válido para TypeScript. Además, también puede tener dificultades para identificar el lenguaje correcto en fragmentos de código demasiado cortos. Para su entrenamiento, Guesslang utiliza un conjunto de datos formado por 1,9 millones de ficheros de código fuente descargados aleatoriamente de 170.000 repositorios de GitHub.

Por otro lado, existen trabajos basados en el análisis de ficheros completos de código fuente. Estos sistemas obtienen una gran precisión ya que un fichero completo, por lo general, proporciona mucha más información que un *snippet* o una línea de código fuente. Sin embargo, estos sistemas introducen el hándicap de necesitar un fichero

completo para obtener los resultados registrados. [VanDam2016] proponen un clasificador basado en cinco modelos de lenguaje natural capaz de identificar 20 lenguajes de programación con un 97,5% de exactitud. Por su parte [Gilda2017] logran clasificar hasta 60 lenguajes con una exactitud del 97% empleando un modelo basado en redes neuronales convolucionales. Mientras que [Khasnabish2014] presentan un modelo basado en un clasificador bayesiano con el que obtienen una exactitud del 93,48% identificando 10 lenguajes.

Mientras los trabajos citados anteriormente basan sus modelos exclusivamente en el análisis de código fuente, existen otros trabajos que, además de entrenar sus modelos con código fuente, también los entrenan con lenguaje natural. Estos trabajos obtienen sus datos de plataformas de preguntas y respuestas como StackOverflow. En estas plataformas, cada entrada suele contener una porción de código fuente y lenguaje natural relacionado con algún escenario o problema de programación concreto. Estos datos aportan más información, lo que, como veremos a continuación, ayuda a construir modelos más precisos. Sin embargo, estos modelos están orientados a este tipo de plataformas, necesitando tanto código fuente como lenguaje natural para su entrenamiento y para realizar inferencias, lo que imposibilita la identificación del lenguaje de programación fuera de ese contexto.

[Alrashedy2020] ofrecen una comparativa entre tres modelos basados en Random Forest y XGBoost capaces de identificar 21 lenguajes de programación. Estos modelos fueron entrenados con distintos *datasets*; uno con una combinación de *snippets* y lenguaje natural, otro sólo con lenguaje natural, y el último únicamente con *snippets*. Las exactitudes obtenidas fueron del 88,9%, 78,9% y 78,1% respectivamente. Por otro lado, [Öztürk2023] propone un modelo entrenado con *snippets* y lenguaje natural capaces de identificar 24 lenguajes. Que mediante la combinación de Random Forest y XGBoost logra una media de 85,5% de exactitud.

Alternativamente, existen trabajos que ofrecen una solución distinta a la identificación de lenguajes basada en texto, como es la identificación basada en imágenes. Esto, aunque añade la propia limitación de necesitar imágenes para que el sistema cumpla su objetivo, aporta la ventaja de no necesitar la tokenización del código fuente ni la definición de un vocabulario. Por ejemplo [Kiyak2020] plantean esta disyuntiva y ofrecen una comparación de estos modelos, ambos usando una red neuronal convolucional [LeCun1998] e identificando únicamente 8 lenguajes distintos. Por un lado, proponen un modelo tradicional basado en ficheros enteros de código fuente, con el que obtienen una exactitud del 98,81%. Mientras que, por otro lado, presentan otro modelo entrenado a partir imágenes de los mismos ficheros, el cual ofrece un 99,38% de exactitud. Por su parte, [Hong2019] proponen un clasificador basado en ResNet, una red neuronal convolucional pre-entrenada en el reconocimiento de imágenes, sobre la cual entrenan dos modelos diferentes. El primero, basado en imágenes de *snippets*, logra una exactitud del 92% identificando 10 lenguajes de programación. Mientras que el segundo, basado en imágenes de funciones, alcanza una exactitud del 99%, aunque sólo identifica 5 lenguajes diferentes. Para ambos modelos usan imágenes generadas por ellos mismos a partir de código fuente de StackOverflow y GitHub.

4. Descripción de sistema

Como hemos mencionado anteriormente, el principal propósito de este trabajo es la identificación del lenguaje de programación al que pertenece una línea de código fuente. Para ello, hemos desarrollado un sistema que recibe como entrada ficheros de código recolectados de GitHub y genera como salida un modelo de aprendizaje automático. Este modelo es capaz de determinar la probabilidad de que una única línea de código fuente pertenezca a cada uno de los 21 lenguajes de programación seleccionados.

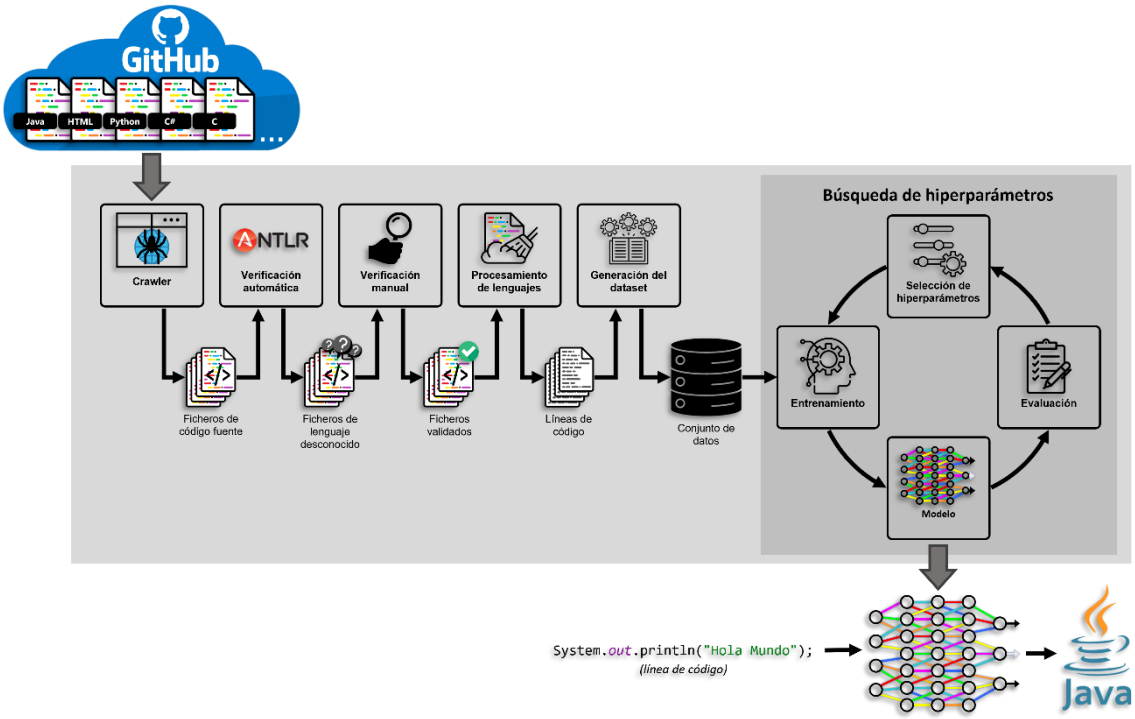

Figura 2. Arquitectura general del sistema donde se muestran los distintos componentes y cómo se comunican entre sí.

La Figura 2 muestra la arquitectura general del sistema donde se pueden apreciar los distintos módulos y cómo se comunican entre sí. Para los sistemas basados en aprendizaje automático, los datos son una parte fundamental. En nuestro caso, empleamos un *crawler* (rastreador) para, de forma automática, realizar peticiones a GitHub y descargar los ficheros de código para los distintos lenguajes contemplados. Antes de poder usarlos, estos ficheros han de ser verificados para comprobar que realmente pertenecen al lenguaje de programación esperado. Esta verificación se realiza se forma semiautomática en dos pasos. Primero, un validador automático, formado por reconocedores sintácticos de los diferentes lenguajes de programación, realizando un análisis sintáctico para determinar si los ficheros son admitidos o no por la gramática del lenguaje asociado a su extensión. Después, con aquellos ficheros que fueron rechazados en el primer paso, se selecciona una muestra representativa y se realiza una verificación manual para comprobar si realmente pertenecen al lenguaje, pero contienen algún error sintáctico o han sido escritos en una versión diferente a la soportada por su gramática.

A partir de estos ficheros ya validados, el sistema puede comenzar con el proceso de limpieza y transformación, empezando por la fase de procesamiento de lenguajes. En esta fase, primero se realiza un procesamiento de los ficheros dependiente del lenguaje donde se eliminan las construcciones que contienen lenguaje natural; como los comentarios de una línea y multilínea, y las cadenas de texto.

A continuación, se realiza otro procesamiento de los ficheros independiente del lenguaje, donde se eliminan todos los espacios en blanco y/o tabulaciones ubicadas al principio y al final de cada línea de código, así como aquellas líneas que no alcanzan una longitud mínima. Ahora que el sistema ya dispone de los datos limpiados (ficheros de líneas de código) es el momento de realizar la transformación, el proceso de generación del *dataset*. Esta fase, agrupa y adapta los datos para que sirvan como entrada del modelo de aprendizaje automático. Para ello, divide cada línea de código en caracteres y añade al final de éstas una etiqueta en referencia al lenguaje al que pertenecen.

En este punto, los datos ya forman un *dataset*, y están listos para ser usados en la búsqueda de hiperparámetros. Este subsistema es el encargado de buscar y entrenar el mejor modelo posible de aprendizaje automático, a través de la búsqueda iterativa de hiperparámetros. Este proceso comienza por la selección de unos hiperparámetros, determinados para la construcción de un modelo de red neuronal de perceptrón multicapa [Rumelhart1986] en base a ellos. A continuación, tiene lugar el entrenamiento de la red, ajustando los parámetros para clasificar mejor el conjunto de datos. Finalmente, se evalúa el modelo, midiendo qué porcentaje de instancias han sido clasificadas por el modelo de forma correcta, comparando la predicción del modelo con la etiqueta (lenguaje de programación) asociada a cada individuo. Este ciclo de búsqueda de hiperparámetros se repite, variando el valor de un hiperparámetro en cada iteración, y seleccionando el modelo con mejores resultados en la evaluación. El modelo final seleccionado (ya entrenado) está listo para la inferencia. Esto es, el modelo será capaz, a partir de una línea de código, determinar la probabilidad de su pertenencia a los distintos lenguajes seleccionados para el conjunto de datos.

4.1. Crawler

Este módulo se encarga de la construcción del corpus de ficheros que será el punto de partida de todo el sistema presentado. Para ello, se le especifican una serie de lenguajes de programación y utiliza la API de GitHub para obtener las direcciones de repositorios de código públicos asociados con estos lenguajes. Después, explora exhaustivamente todos los directorios de cada repositorio y descarga todos los ficheros cuya extensión coincida con alguna de los lenguajes solicitados. El resultado es un corpus dividido en una carpeta por lenguaje, donde se organizan todos los ficheros descargados según su extensión. Además, se almacenan metadatos acerca de los ficheros descargados, los repositorios explorados, estadísticas que representan el volumen de datos descargados para cada lenguaje e información para recuperarse de cualquier error y continuar la descarga en el punto exacto que se interrumpió o paró. Este módulo ha sido implementado usando Python 3.8 y la librería PyGithub v1.53 [PyGithub2023], que permite consultar información de forma programática a la API de GitHub desde Python.

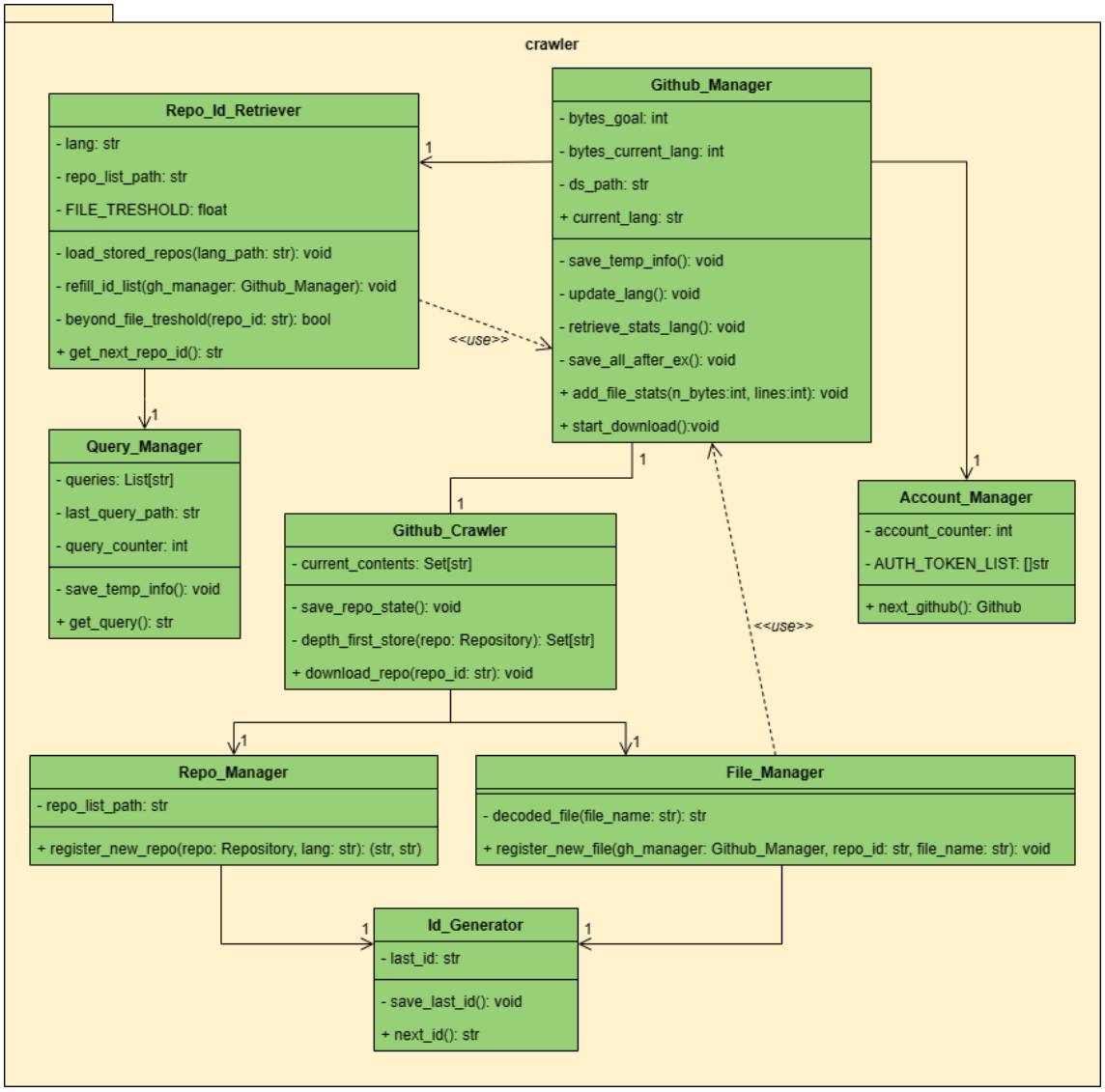

Figura 3. Diagrama de clases UML del módulo crawler.

La Figura 3 muestra la arquitectura del rastreador (*crawler*), donde se puede observar su principal componente, el `Github_Manager`, el cual orquesta el resto de los componentes. El *crawler* funciona llamando en bucle a la función `start_download` de esta clase. Esta función recupera la URL de un repositorio de código del lenguaje objetivo, explora todos sus directorios y descarga todos los ficheros que encajen con alguna de las extensiones predefinidas. Al finalizar la descarga, evalúa si es preciso cambiar de lenguaje objetivo, consultando los metadatos almacenados y comprobando si el número de bytes descargados del lenguaje actual ha superado cierto número de bytes (atributo `bytes_goal`). Cuando todos los lenguajes configurados han superado este número de bytes, pasa a ser el doble, y se vuelve a seleccionar el primer lenguaje de la lista.

La recuperación de las URLs de los repositorios es responsabilidad de la clase `Repo_Id_Retriever`. Cuando se instancia esta clase, se realizan consultas a la API de GitHub, recuperando un número predefinido de URLs, que son almacenadas en listas internas. Cada vez que se llame a la función `get_next_repo_id` se devolverá una de

las URLs y se eliminará de la lista. Cuando la lista se vacía se rellena por medio de otra consulta. La clase `Repo_Id_Retriever` guarda una lista con todos los repositorios minados para el lenguaje objetivo hasta el momento, de forma que, si el repositorio de la URL ya ha sido descargado, lo omite y se evalúa el siguiente. Adicionalmente, se consultan los metadatos del repositorio para comprobar que el porcentaje de ficheros del lenguaje objetivo supera cierto umbral (mediante la función `beyond_file_treshold`), de lo contrario también se omite la URL y se trata la siguiente.

La codificación y ordenación de las consultas es responsabilidad de la clase `Query_Manager`, que retorna la siguiente consulta a efectuar al `Repo_Id_Retriever`. Cada consulta se codifica para buscar repositorios etiquetados con el lenguaje de programación objetivo recibido por parámetro. Además, estas consultas a la API de GitHub pueden contener más restricciones, como sobre el tamaño del repositorio, el número de estrellas, de seguidores y de *forks*. El `Query_Manager` genera cientos de consultas, filtrando los repositorios por tamaño, para evitar que continuamente se retornen repositorios que ya han sido explotados previamente. Las consultas simplemente buscan programas cuyo tamaño cae dentro de unos rangos predefinidos, cubriendo todo el espectro de programas disponibles, y dando prioridad a los repositorios de mayor tamaño. En el momento que ya se han agotado todas las consultas, lanza un error hasta el `Github_Manager` para que cambie de lenguaje objetivo por falta de URLs.

La exploración de los directorios de un repositorio y la descarga de sus ficheros es responsabilidad de la clase `Github_Crawler`. Este componente utiliza también la API de GitHub para acceder a los distintos directorios y ficheros de un repositorio recibido por parámetro. Antes de comenzar la exploración del repositorio, la clase `Repo_Manager` utiliza la API de GitHub para consultar información acerca de éste y almacenarla como metadatos. De cada repositorio queda registrado el *commit* del código descargado, el nombre de la rama, la fecha, el propietario, su descripción, tamaño, número de estrellas, *forks* y suscriptores, entre otros. A continuación, el `Github_Crawler` realiza una búsqueda en profundidad por todos los directorios, y se comprueba la extensión de los ficheros que se van encontrando. Cuando esta extensión coincide con alguna de las asociadas a cualquiera de los lenguajes predefinidos, se extrae el contenido del fichero y se almacena en el directorio del lenguaje correspondiente. La extracción del contenido de los ficheros y el registro de sus metadatos los realiza la clase `File_Manager`. Para cada fichero se almacena su extensión, su ruta completa en el repositorio, tamaño, fecha, número de líneas, lenguaje de programación, y el ID de su repositorio. Por último, destacamos que la clase `Github_Crawler` almacena en unas listas internas los directorios ya explorados y ficheros ya descargados del repositorio. De esta forma, ante una parada del sistema o cualquier error (ya sea por una excepción de GitHub, un problema con la red, etc.), el estado de la exploración queda almacenado y se puede retomar en cuanto el sistema se arranque de nuevo o se recupere del error.

Como hemos ido comentado, todo este módulo está preparado para poder recuperarse de cualquier error y continuar el minado continuo de repositorios. Cuando nos encontramos ficheros de más de 1MB, la API de GitHub lanza una excepción, que es controlada por el `Github_Crawler`, que registra el problema y salta al siguiente fichero. Cuando, al intentar acceder a la información de algún repositorio, se genera alguna excepción, por ejemplo, si el repositorio no es accesible por algún problema, el `Github_Manager` la controla y solicita la URL del siguiente. Si se recibe una excepción de fallo de conexión a Internet, el sistema espera un tiempo predefinido e intenta continuar la descarga del repositorio en el punto donde se dejó. Otro error muy común es el de haber excedido el límite de consulta a la API de GitHub por unidad de tiempo. Esto se debe a que la versión utilizada de la librería PyGithub (1.53), establece unos límites de descarga por tiempo y cuenta de GitHub. En el momento que se superan, se debe esperar unas horas para volver a consultar. Debido a esto, se creó la clase `Account_Manager`, que almacena datos de varias cuentas de GitHub. Su responsabilidad es cambiar de cuenta cada vez que se supera el límite mencionado, evitando así los largos tiempos de espera. Ante cualquier otra excepción, el `Github_Manager` la controla, la registra y persiste toda la información interna para cargarla en el nuevo arranque y poder retomar la descarga del siguiente fichero o repositorio.

4.2. Verificación automática

Este módulo se ocupa de la primera fase del proceso de verificación de ficheros de código fuente. Aquí, el objetivo es verificar, de forma automática, la pertenencia al lenguaje asociado a su extensión, del mayor número de ficheros posible. Para ello, se implementa `code_validator`, una herramienta de verificación automática que, mediante el uso de reconocedores sintácticos basados en gramáticas ANTLR 4, permite resolver si un conjunto de ficheros de código satisface la sintaxis de un determinado lenguaje de programación. Este módulo ha sido implementado usando Java 18.0.1.1 y la librería ANTLR v4.10.1 [ANTLR2023a].

Como se puede observar en la Figura 4, esta herramienta se divide en dos partes, `core` y `cli` (interfaz de línea de comandos). `core` es el componente encargado de llevar a cabo todo el proceso de validación, ofreciéndolo a través de dos opciones, `Evaluate` y `Move`. Ambas opciones realizan un proceso de validación donde se comprueba si cada uno de los ficheros de código de un conjunto pertenecen a un determinado lenguaje de programación. Pero, mientras que la primera sólo devuelve el resultado de la validación, la segunda, además, organiza los ficheros analizados, moviéndolos a distintos directorios, uno para los admitidos por la gramática y otro para los no admitidos, manteniendo los ficheros que no fueron validados en el directorio de origen. El resultado de cada validación se engloba en `ValidationResult` el ofrece los datos de esta, así como un resumen compuesto por el número de ficheros validados correctamente, el número de ficheros procesados, el porcentaje de ficheros validados frente al total de ficheros procesados, y el porcentaje de ficheros procesados frente al número total de ficheros disponibles para un determinado lenguaje.

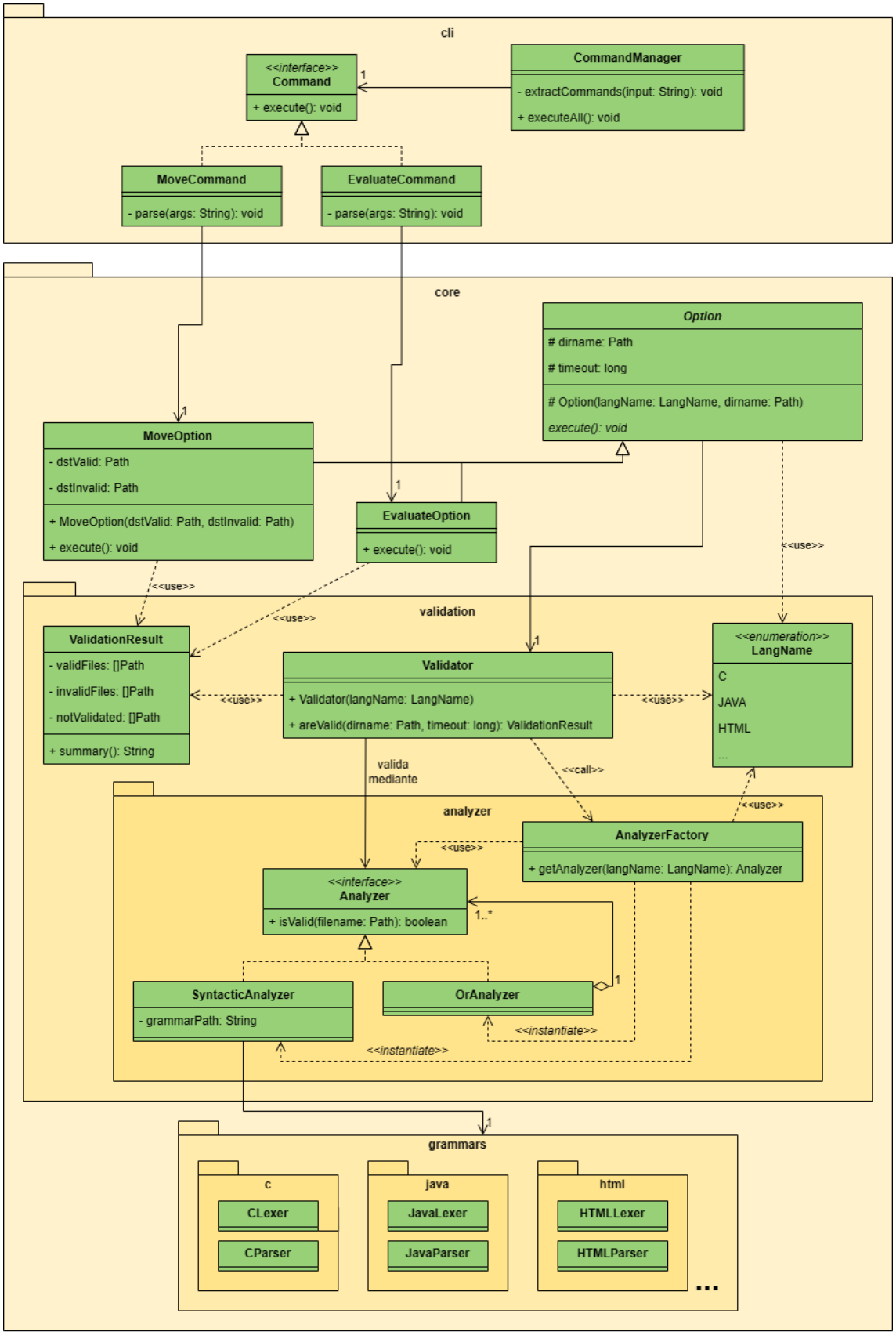

Figura 4. Diagrama de clases UML del módulo de verificación automática.

Antes de comenzar a validar los ficheros de código, el validador comienza por preparar el marco de trabajo. Primero, éste solicita un reconocedor sintáctico preconfigurado con el lenguaje de programación contra el cual se quiere validar los ficheros. Posteriormente, se accede al directorio donde se encuentra el conjunto de ficheros a validar para el lenguaje seleccionado y genera una lista con las rutas de acceso para todos y cada uno de ellos. Una vez preparado, el validador realiza de forma concurrente el reconocimiento sintáctico de cada fichero determinando si éste es admitido por la gramática asociada (y, por tanto, pertenece) al lenguaje, o no. En ocasiones, no se puede realizar la validación de algunos ficheros, ya sea por un error de lectura o; porque, debido a su tamaño y a las características de la gramática, su tiempo de análisis excede cierto *timeout* preconfigurado. Como resultado de este proceso se crean tres listas, una formada por los ficheros pertenecientes al lenguaje, otra con los que no pertenecen y una última con los ficheros que no pudieron ser validados por alguno de los motivos mencionados anteriormente.

```
Identifier
    :   IdentifierNondigit
        (   IdentifierNondigit
        |   Digit
        )*
    ;

fragment
IdentifierNondigit
    :   Nondigit
    |   UniversalCharacterName
    ;

fragment
Nondigit
    :   [a-zA-Z_]
    ;

fragment
Digit
    :   [0-9]
    ;

Constant
    :   IntegerConstant
    |   FloatingConstant
    |   CharacterConstant
    ;
```

Figura 5. Fragmento de la especificación léxica del lenguaje de programación C.

Como ya se mencionó en el párrafo anterior, el sistema dispone de varios reconocedores sintácticos preconfigurados para distintos lenguajes como C, C++, Java, Python, HTML, etc. Todos éstos están basados en gramáticas libres de contexto [Chomsky1959] (Figura 6), que dan lugar a reconocedores sintácticos, y especificaciones léxicas (Figura 5) definidas por medio de expresiones regulares [Kleene1956], que dan lugar a los distintos analizadores léxicos utilizados por los reconocedores. Además, en el caso

de algunos lenguajes como C y HTML, se requiere de preprocesadores que deben de tratar los ficheros antes de comenzar el reconocimiento. Aunque lo más común es que un analizador esté formado por una sola gramática, existen lenguajes para los que, debido a su cantidad de variantes, decidimos definir analizadores compuestos por varias gramáticas diferentes. Por ejemplo, para el analizador de SQL, usamos las gramáticas de SQLite, MySQL, PostgreSQL, PL/SQL y TSQL. Mientras que interpretar el resultado de validación de un analizador simple es trivial, un analizador compuesto genera el resultado con un razonamiento disyuntivo: para que el fichero pertenezca al lenguaje ha de satisfacer al menos uno de los reconocedores sintácticos que componen el analizador.

```
argumentExpressionList
    :   assignmentExpression (',' assignmentExpression)*
    ;

multiplicativeExpression
    :   castExpression (('*'|'/'|'%') castExpression)*
    ;

additiveExpression
    :   multiplicativeExpression (('+'|'-') multiplicativeExpression)*
    ;

shiftExpression
    :   additiveExpression (('<<'|'>>') additiveExpression)*
    ;

relationalExpression
    :   shiftExpression (('<'|'>'|'<='|'>=') shiftExpression)*
    ;

equalityExpression
    :   relationalExpression (('=='| '!=') relationalExpression)*
    ;
```

Figura 6. Fragmento de la gramática del lenguaje de programación C.

Las gramáticas, especificaciones léxicas y preprocesadores utilizados en este módulo se descargaron de [ANTLR2023b], un repositorio de GitHub propiedad de ANTLR con información acerca de 277 lenguajes distintos. Sin embargo, la aplicación directa de estas gramáticas planteó una serie de inconvenientes. En primer lugar, algunas gramáticas, como la de Unix Shell o Perl no figuran en el repositorio. En el caso de la de Unix Shell, al ser más sencilla, se pudo implementar desde cero. Sin embargo, para Perl, por cuestiones de tiempo, no se pudo completar la implementación, ni se encontraron documentos de gramáticas en el formato aceptado por ANTLR. Otro problema es que algunas gramáticas pueden estar incompletas o ser incorrectas, como fue el caso de Matlab o Ruby, entre otros. En algunos de estos casos fue posible refinar las gramáticas encontradas para mejorar su efectividad de cara al reconocimiento sintáctico de ficheros pertenecientes al lenguaje. Pero, en otras ocasiones, de nuevo por falta de tiempo, esto no fue posible. Por último, tenemos el problema de la versión de la gramática y hasta qué versión del lenguaje soporta. Cada gramática, a través de un léxico

y una sintaxis determinados, define una versión concreta de un lenguaje de programación, por lo que puede no resultar eficaz para aquellas versiones del lenguaje que empleen un léxico y/o una sintaxis distinta a la versión de la gramática. Este problema ocurre, por ejemplo, con la gramática de Java 9 la cual no soporta nuevas especificaciones de Java 14 como los *records*, véase en la Figura 7. La solución a este problema fue emplear los ya mencionados reconocedores compuestos, aglutinando gramáticas para diferentes versiones de un determinado lenguaje, como es el caso del reconocedor compuesto implementado para Java.

```
// Clase DTO en Java 9
public final class Rectangle {
    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double length() { return this.length; }
    double width()  { return this.width; }
}

// Clase DTO equivalente en Java 14 que no es soportado por Java 9
record Rectangle(double length, double width) { }
```

Figura 7. Comparativa léxica y sintáctica entre Java 9 y Java 14.

Finalmente, en cuanto al `cli`, es el encargado de realizar el papel de intérprete entre el usuario y el `core`, interpretando y ejecutando las instrucciones del usuario. Cada instrucción está formada por una etiqueta seguida de sus argumentos. Esta etiqueta tiene dos objetivos, identificar de qué tipo de instrucción se trata, y como separador para encadenar varias instrucciones en una sola. Inicialmente estas instrucciones son ilegibles para el `core` (sólo son una cadena de texto) por lo que la interfaz comienza un proceso de interpretación para poder solicitar la funcionalidad al `core`. Primero el gestor de comandos busca e identifica las etiquetas contenidas en la entrada y divide ésta en tantas partes como etiquetas haya encontrado. Una vez las instrucciones se han identificado, se les asigna sus respectivos comandos, los cuales reconocen sus argumentos. Con el proceso de interpretación completado, el gestor de comandos ejecuta las instrucciones a través de los comandos mencionados anteriormente.

4.3. Procesador de lenguajes

Este módulo aborda el procesamiento de los ficheros ya validados por el proceso de verificación semi-automática anterior, donde el objetivo es transformar la información de acuerdo con las necesidades del sistema. Para ello se desarrolla `code_processor`, una herramienta que, basándose en la sintaxis de cada lenguaje de programación, permite modificar determinadas partes del contenido de cada uno de los ficheros de

código fuente que recibe como entrada, produciendo un conjunto de ficheros de líneas procesadas. Este módulo ha sido implementado usando Go 1.18.1.

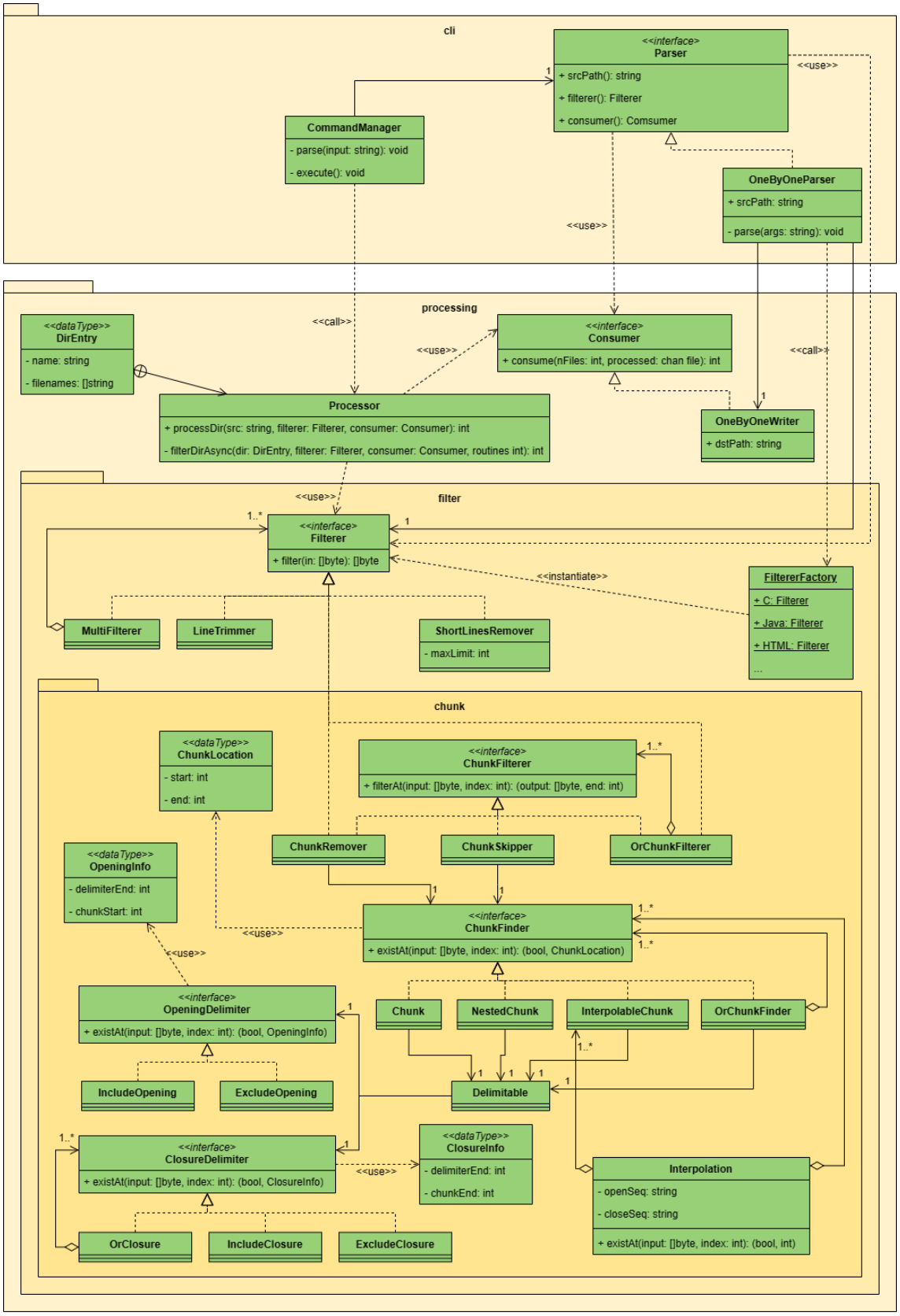

Figura 8. Diagrama de clases UML del módulo de procesamiento de lenguajes.

Como muestra la Figura 8, esta herramienta se divide en dos componentes, `processing` y `cli`. `processing` es la parte principal de este sistema. Este componente es el encargado de realizar el proceso de transformación en el cual se convierte los ficheros de código fuente validados a ficheros de líneas de código procesadas. Este proceso es gestionado por `Processor`, el cual lo divide en múltiples tareas independientes capaces de resolverse de forma asíncrona. Cada una de estas tareas se encarga de procesar un único fichero. Para ello, primero lee el fichero, luego modifica su contenido y finalmente crea otro fichero con el contenido ya procesado.

De los tres pasos, el más importante es la modificación, el cual se realiza mediante la aplicación de filtros en varias pasadas distintas, ya que unos filtros están supeditados a otros. Primero, se eliminan aquellas construcciones de cada lenguaje de programación que contienen lenguaje natural: cadenas de texto, comentarios de línea y comentarios multilínea. Para eliminarlas, se emplea un algoritmo de sustitución que, mediante el uso de pequeñas reglas léxicas predefinidas para cada lenguaje de programación, permite localizar y borrar todos los fragmentos que se encuentren delimitados por unas determinadas secuencias de caracteres de apertura y cierre. Cuando el algoritmo detecta un delimitador de apertura, omite todos los demás delimitadores posibles hasta que encuentre el correspondiente delimitador de cierre.

```go
func Area(rectangle Rectangle) float32 {
    fmt.Println("Esto es una cadena con apertura de comentario /*")

    var length float32
    var width float32

    length, width = rectangle.Size()

    /* Esto es un comentario con apertura de cadena " */
    return length * width
}
```

Figura 9. Fragmento de código fuente escrito en el lenguaje Go donde la parte sombreada indica lo que borraría una regla léxica de eliminación de comentarios multilínea.

Durante la búsqueda, como muestra la Figura 9, puede darse la situación donde varias construcciones puedan interpretarse como solapadas, por lo que realizar la búsqueda secuencial de cada construcción puede producir resultados inconsistentes dependiendo de que regla se ejecute antes. Para ello, se realiza una búsqueda simultanea mediante `OrChunk`, el cual, agrupa varias reglas léxicas asignándoles una prioridad y examina, de forma ordenada, cada carácter del fichero de código buscando posibles coincidencias con cada una de las reglas léxicas que lo componen. Dependiendo del fragmento, sus delimitadores pueden están incluidos en el mismo, o no, como, por ejemplo, las cadenas de texto, donde sus delimitadores están incluidos en el fragmento, o la sección de estilos en HTML delimitada por las etiquetas `<style>…</style>`, las cuales están excluidas del fragmento. Además, en determinadas ocasiones, un mismo fragmento puede estar delimitado por distintas etiquetas de cierre, como es el caso de los comentarios en PHP, los cuales pueden terminar con un salto de línea, o con la etiqueta de cierre `?>`.

```
/* El comentario comienza aquí
/* contiene un comentario anidado */
y termina aquí. */
```

Figura 10. Comentario multilínea anidado escrito en el lenguaje Kotlin.

Debido a la estructura de alguna de las construcciones, su búsqueda requiere de procesos diferentes. Dejando a un lado las construcciones sencillas cuya estructura simplemente es lenguaje natural entre dos delimitadores, existe otro tipo de construcciones donde el contenido entre los delimitadores puede albergar, a su vez, otras construcciones. Este es el caso, por ejemplo, de los comentarios multilínea de Kotlin, los cuales, como se puede observar en la Figura 10, pueden contener más comentarios multilínea anidados. O el caso de las cadenas interpoladas, las cuales puede contener más cadenas anidadas, como muestra la Figura 11 para el lenguaje C#. Para estos casos, cada vez que `NestedChunk` o `InterpolableChunk` encuentran un delimitador de apertura, van apilando sucesivamente, mediante recursividad, una nueva ejecución en la pila de llamadas hasta encontrar su delimitador de cierre, momento en el que termina la ejecución y continúa buscando otros fragmentos hasta encontrar el delimitador de cierre del fragmento principal.

```
Console.WriteLine($"¡Esto {$"es {$"una {$"cadena"}"}"} interpolada!");
// Muestra: ¡Esto es una cadena interpolada!
```

Figura 11. Cadena formada por varias interpolaciones anidadas escrita en el lenguaje C#.

Una vez se ha borrado el lenguaje natural, `LineTrimmer` se encarga de aplicar otro filtro para quitar los espacios en blanco y/o las tabulaciones situadas al principio y al final de cada línea de código. Finalmente, `ShortLinesRemover` elimina aquellas líneas que, sin tener en cuenta los espacios en blanco y tabulaciones entre cada palabra, una determinada longitud mínima. El resultado de este proceso es un fichero de líneas de código procesadas por cada fichero de entrada.

Por otro lado, el componente `cli` o interfaz en línea de comandos, sirve como intermediario entre el usuario y el componente `processing`, traduciendo y ejecutando las instrucciones que el usuario introduce por consola. Inicialmente, cada una de estas instrucciones es simplemente una cadena de texto, la cual comienza por el nombre del lenguaje de programación a procesar, seguido de la ubicación de los ficheros originales y la dirección de salida de los ficheros procesados. La traducción se centra exclusivamente en convertir el nombre del lenguaje, el cual permite seleccionar la configuración léxica adecuada para modificar los ficheros, a través del `Parser OneByOneParser`. Una vez la instrucción la instrucción ha sido traducida, el `CommandManager` se encarga de comunicarse con `processing` para que comience con el procesamiento de los ficheros.

4.4. Generación del conjunto de datos

Este módulo se ocupa de la generación del conjunto de datos (*dataset*), el proceso por el cual, mediante la aplicación de varias operaciones, se convierte el corpus de ficheros

de líneas procesadas en un conjunto de instancias preparado para entrenar los distintos modelos de aprendizaje automático. Este módulo se desarrolló en Python 3.8, usando las librerías de tensorflow 2.5.0 [TensorFlow2023], keras 2.9.0 [Keras2023] y numpy 1.19.5 [NumPy2023], y provee las funcionalidades presentadas a continuación.

La primera funcionalidad necesaria para la generación del *dataset* es la de unificar todas las líneas procesadas en un único conjunto de datos. Para ello se realiza una lectura de los ficheros de líneas procesadas de todos los lenguajes de programación contemplados. De esta lectura se extrae cada línea de los ficheros y se le añade una etiqueta numérica al final que identifica el lenguaje al que pertenece.

Una vez las líneas se encuentran agrupadas, este módulo se encarga de su tokenización, fase en la cual pasan de ser líneas textuales a vectores numéricos del mismo tamaño con los que entrenar y evaluar los modelos de aprendizaje automático. Para ello, primero se truncan las líneas cuya longitud excede el límite máximo definido. Después, las líneas se dividen en caracteres y estos, a su vez, se sustituyen por sus respectivos códigos ASCII, dando lugar a los mencionados vectores de números enteros de igual longitud. Posteriormente, se modifican estos vectores para representarse sobre un vocabulario cerrado predefinido. Con vocabulario cerrado nos referimos a un conjunto de códigos ASCII, definido a priori, que limita los distintos caracteres que pueden ser representados por nuestro conjunto de datos. Si un carácter está dentro de este vocabulario, se representará mediante su código ASCII y, de lo contrario, se representará por un token especial denominado OOV (Out Of Vocabulary), que será el mismo para cualquier carácter que no forme parte del vocabulario.

Adicionalmente, es necesario transformar los distintos códigos ASCII pertenecientes al vocabulario para que formen un rango cerrado que empiece en 0. Éste es el formato de entrada requerido para la primera capa de nuestros modelos de redes neuronales, así como para la transformación de estos códigos ASCII a formato *one-hot*, necesario para el entrenamiento de los demás modelos de aprendizaje automático utilizados en el proyecto. El formato *one-hot* es una manera de representar números enteros como vectores numéricos llenos de ceros (dispersos), con un único uno en el índice correspondiente al número representado. Este formato es muy útil a la hora de representar variables categóricas nominales, en las que las distintas categorías no obedecen a una clasificación intrínseca, como es el caso de los distintos caracteres para nuestro problema. De esta forma, los distintos modelos de aprendizaje utilizados no interpretarán estos códigos ASCII como variables numéricas continuas, sino como categorías nominales igualmente distintas y equidistantes entre sí en el espacio vectorial. Para finalizar el proceso de tokenización, la última funcionalidad necesaria es la de relleno (*padding*). Como todos los individuos del conjunto de datos deben tener exactamente la misma longitud, aquellas líneas que no alcanzan la longitud máxima definida se completan repitiendo un token especial hasta que alcance la longitud esperada. De esta forma, se consigue que el conjunto de líneas procesadas pase a ser un conjunto de vectores numéricos del mismo tamaño.

Otra funcionalidad necesaria para construir un conjunto de datos de calidad es la de barajar todas las instancias que lo conforman. Para ello, en primer lugar, este módulo

realiza la mencionada lectura de ficheros extrayendo sus líneas de forma intercalada. Esto es, se leen simultáneamente varios ficheros y sus líneas se van entremezclando para dar lugar al conjunto de líneas de código sobre el que se aplicará la tokenización mencionada en el párrafo anterior. Adicionalmente, en un paso posterior, también se mezclan aleatoriamente las distintas líneas extraídas de los ficheros. Como el proceso de barajar todos los individuos del conjunto de datos puede ser muy costoso, especialmente para conjuntos muy grandes, como los que empleamos para este proyecto, este proceso se desarrolla en dos partes. Primero, se dividen las instancias en subconjuntos del mismo tamaño (denominados *batches* o lotes), y estos subconjuntos se barajan aleatoriamente. Después, se mezclan aleatoriamente todas las instancias de cada subconjunto.

La última funcionalidad necesaria para la generación del conjunto de datos final consiste en separar, los vectores numéricos que representan las distintas líneas de tokens, de las etiquetas que identifican el lenguaje de programación al que pertenecen dichas líneas. Adicionalmente, también se dividen, a su vez, en dos subconjuntos, el de entrenamiento y el de validación. Este proceso, además permite seleccionar el número de individuos que tendrá el *dataset*, amén de permitir exportarlo en formato CSV o el formato de serialización de Python, el *pickle*, que coincide con el formato de entrada del proceso de búsqueda de hiperparámetros.

4.5. Búsqueda de hiperparámetros

La búsqueda de hiperparámetros es un subsistema formado por varios módulos, cuyo objetivo es, partiendo de un conjunto de datos, crear, entrenar y evaluar la eficiencia de distintos modelos de aprendizaje automático construidos en función de diferentes hiperparámetros, para finalmente seleccionar el que presente mejor rendimiento sobre el conjunto de validación. Este subsistema lleva a cabo su objetivo mediante un proceso iterativo en el que intervienen cuatro módulos: selección de hiperparámetros, entrenamiento, construcción del modelo y evaluación. Cada iteración, mediante la variación de un hiperparámetro, busca crear un nuevo modelo, entrenarlo y evaluar su rendimiento. Para ello, se comienza por establecer una nueva configuración de hiperparámetros modificando tan sólo uno de ellos. Posteriormente, se crea un nuevo modelo en base a esa configuración y se entrena usando los datos del conjunto de entrenamiento. Finalmente se evalúa el modelo entrenado sobre el conjunto de validación. Al finalizar todas las variaciones de hiperparámetros contempladas, se selecciona el modelo de mayor rendimiento según la citada evaluación realizada al final de cada iteración. Este módulo se desarrolló en Python 3.8, usando las mismas librerías que se citan al principio de la sección del módulo anterior, además de scikit-learn 1.2.2 [SciKitLearn2023].

Todos los modelos de redes neuronales implementados para este proyecto se basan en la arquitectura del perceptrón multicapa [Rumelhart1986]. La arquitectura empleada se compone principalmente de la capa de entrada, una capa de incrustaciones (*embeddings)* [Mikolov2013], tres capas ocultas y la capa de salida. La capa de entrada es la encargada de recibir los datos en forma de vectores numéricos de misma longitud.

Como ya hemos mencionado, estos datos, al ser códigos ASCII relativos a caracteres, constituyen categorías nominales, por lo que es necesario representarlos de una manera distinta a la mera utilización de un número entero. La solución más simple a este problema consiste en que la capa de entrada de nuestra arquitectura soporta vectores en formato *one-hot,* comentados en la sección anterior. Sin embargo, el problema de esta codificación es que añade demasiada dimensionalidad al modelo, ya que cada elemento del vocabulario pasa de ser representado por un único número entero a ser representado por un vector de la misma dimensión que el vocabulario. Por ello, adicionalmente, nuestra arquitectura también provee una capa de *embeddings* opcional, que sólo se crea si la entrada al modelo no se ha transformado previamente a la codificación *one-hot.* Esta capa permite representar cada elemento del vocabulario en un espacio multidimensional como vectores densos de una longitud configurable. Los *embeddings*, realizan esta transformación aprendiendo, a medida que se va entrenando la red, cómo traducir cada elemento del vocabulario en función del contexto en el que se usa. Es decir, aprendiendo a representar cada carácter del vocabulario en función de los demás caracteres que le rodean en sus distintas ocurrencias dentro del conjunto de entrenamiento. Con esto conseguimos reducir la dimensionalidad del modelo gracias a la agrupación de características, lo que se traduce en una disminución de los tiempos y del espacio de computación con respecto a la codificación *one-hot*.

Cada una de las capas densas están compuestas por el mismo número de neuronas, las cuales, mediante la aplicación de una función de activación a la suma ponderada de sus entradas, calculan su salida y se interconectan totalmente con la siguiente capa. Nuestro sistema utiliza varias funciones de activación como hiperparámetros: SELU [Klambauer2017], ELU [Clevert2015], Leaky ReLU [Maas2013] y ReLU [Nair2010]. Para que una red neuronal aprenda de manera efectiva, necesita que los pesos de sus conexiones tengan, inicialmente un valor aleatorio. A su vez, cada función de activación se comporta mejor con una determinada inicialización de pesos, por lo que nuestro modelo asocia cada función de activación con una estrategia de inicialización de pesos.

Además de las capas densas, nuestro modelo permite añadir una capa de *dropout* [Srivastava2014] entre cada dos capas densas para que, a medida que avance el entrenamiento, el modelo apague aleatoriamente determinado porcentaje de neuronas y así poder retrasar el sobreajuste (*overfiting*). *Dropout* es un mecanismo muy efectivo de regularización de redes neuronales, que permite aumentar su capacidad de generalización [Srivastava2014].

Inicialmente, los modelos basados en redes neuronales no fueron diseñados para la clasificación de múltiples categorías. Pero, configurando la capa de salida con una neurona por cada categoría a predecir, junto con la función de activación *softmax*, se puede conseguir una salida en forma de vector de números reales que representa la probabilidad de pertenencia de una entrada a cada una de las distintas categorías.

Para su aprendizaje, el modelo necesita pasar por la fase de entrenamiento, donde, mediante el uso de técnicas de aprendizaje automático, adquiere la capacidad de predecir. Esta fase utiliza los datos del conjunto de entrenamiento, el cual, para agilizar el entrenamiento, se divide en pequeños subconjuntos llamados lotes (*batches*). El

proceso de aprendizaje se lleva a cabo de manera iterativa de forma que, en cada iteración, el modelo recibe un *batch*, genera una salida, calcula su error con respecto a la etiqueta esperada y actualiza los pesos del modelo para reducir ese error mediante el uso de un algoritmo de gradiente descendiente. Cuando todos los *batches* han sido procesados por el modelo, se constituye lo que se conoce como una época (*epoch*).

Otro de los aspectos a tener en cuenta es cuando parar un el entrenamiento de un modelo. Inicialmente esto viene determinado por del número de *epochs*, que se considera un hiperparámetro del entrenamiento. Este hiperparámetro va variando cada vez que la red entrena con todas las instancias del conjunto de entrenamiento, lo que permite mantener el resto de hiperparámetros de la configuración exactamente igual y aprovechar el modelo anterior para reentrenarlo. Pero el resultado de este reentrenamiento iterativo no tiene por qué producir un modelo con mejor rendimiento que el anterior, por lo que se aplicamos un nuevo criterio de parada: la parada tempana o *early stopping* [Haykin1999]. Ésta determina que un modelo tiene que dejar de ser reentrenado si en su último entrenamiento obtuvo un rendimiento peor que el anterior. Aunque este criterio ejerce de mecanismo de regularización para evitar que un modelo haga sobreajuste (*overfiting*), en determinadas ocasiones puede ser conveniente seguir reentrenando un modelo a pesar de que los últimos entrenamientos tengan un menor rendimiento. Existen situaciones donde un modelo reentrenado obtiene peores resultados que su antecesor, pero en la siguiente iteración mejora a todos los anteriores. Debido a esto, se usa otro hiperparámetro de entrenamiento llamado paciencia. Este hiperparámetro determina el número de *epochs* que el *early stopping* debe permitir a pesar de obtener peores resultados que el mejor modelo encontrado hasta el momento.

5. Metodología

5.1. Obtención del corpus de ficheros

Antes de comenzar este trabajo decidimos realizar una búsqueda para identificar los lenguajes de programación más populares a fecha del 21/07/2020. Para ello, acudimos a los siguientes rankings de sitios de utilización de lenguajes de programación: Tiobe [TIOBE2023], PYPL [PYPL2023] y RedMonk [RedMonk2023]. Seleccionamos los 15 lenguajes más utilizados según cada ranking.

Con el fin de complementar este proceso de selección, utilizamos BigQuery [BigQuery2023] para extraer una lista de los 15 lenguajes de GitHub con mayor número de repositorios de código. Hay que mencionar que BigQuery sólo nos permitió la consulta de información sobre repositorios públicos y con licencia. A pesar de esto, se consultó información de más de 7 millones de repositorios. Los resultados de esta búsqueda se agruparon en la Tabla 1, la cual utilizamos para realizar la unión de todos sus lenguajes y obtener así, los lenguajes de programación sobre los que desarrollaríamos nuestro proyecto:

Posición	GitHub	TIOBE	PYPL	RedMonk
1	JavaScript	C	Python	JavaScript
2	CSS	Java	Java	Python
3	HTML	Python	JavaScript	Java
4	Shell	C++	C#	PHP
5	Python	C#	C/C++	C#
6	Ruby	Visual Basic .NET	PHP	C++
7	Java	JavaScript	R	Ruby
8	PHP	R	Objective-C	CSS
9	C	PHP	Swift	TypeScript
10	C++	Swift	TypeScript	C
11	Makefile	SQL	Matlab	Swift
12	Objective-C	GO	Kotlin	Objective-C
13	C#	Assembly language	GO	Scala
14	Perl	Perl	VBA	R
15	Go	MATLAB	Ruby	GO

Tabla 1. Rankings con los lenguajes de programación más populares en GitHub, TIOBE, PYPL y TedMonk.

De esta forma, los lenguajes seleccionados en un principio fueron 25: Assembly, C, C#, C++, CSS, GO, HTML, Java, JavaScript, Kotlin, Makefile, MATLAB, Objective-C, Perl, PHP, Python, R, Ruby, Scala, Shell, SQL, Swift, TypeScript, VBA y Visual Basic .NET.

Para la elaboración de un corpus de ficheros de estos lenguajes se utilizó el mencionado módulo del *crawler*, que utiliza la API de GitHub para descargar ficheros de repositorios de código públicos. Estos repositorios proveen metadatos sobre el porcentaje de ficheros que contienen de cada lenguaje. Sólo se minaron aquellos repositorios que tenían más de un 10% de ficheros de alguno de los 25 lenguajes especificados. Una de las limitaciones de esta API, es que no permitía la descarga de ficheros mayores de 1 MB, por lo que estos no están incluidos en el corpus de ficheros. Como ya se mencionó en la descripción del sistema, nuestro *crawler* se diseñó para nunca descargar dos veces el mismo repositorio, ni el mismo fichero dentro de un mismo repositorio.

Lenguaje	Extensiones
Assembly	asm, s, S
C	c
C#	cs
C++	cpp, cc, cx
CSS	css
Go	go
HTML	html
Java	java
Javascript	js
Kotlin	kt, kts, ktm
Makefile	cmake
Matlab	m
Objective-C	mm, M
Perl	pl
PHP	php
Python	py
R	r
Ruby	rb
Scala	scala, sc
SQL	sql
Swift	swift
Typescript	ts
Unix Shell	sh
VBA	vba
Visual Basic .Net	vb

Tabla 2. Configuración del crawler con las extensiones asociadas para descargar cada lenguaje de programación.

Además, la descarga y organización por lenguaje de los ficheros se realiza en base a su extensión. En la Tabla 2 se muestran las distintas extensiones configuradas en el *crawler* asociadas a los 25 lenguajes seleccionados. En esta tabla, se puede observar cómo la extensión .h no está contemplada para ningún lenguaje. Esta decisión se tomó porque la extensión .h la utilizan 3 lenguajes de la lista: C, C++ y Objective-C; por lo que sería

problemática a la hora de distinguir a qué lenguaje pertenece un fichero descargado con esta extensión.

El cr*awler* estuvo funcionando intermitentemente medio año (desde septiembre de 2020 hasta febrero de 2021), confeccionando el corpus cuyas dimensiones se muestran a continuación, en la Tabla 3:

Lenguaje	Ficheros	Tamaño (MB)
Assembly	44.627	684
C	34.687	1.327
C++	23.714	645
C#	56.494	640
CSS	14.840	642
Go	76.733	1.454
HTML	22.143	642
Java	110.169	1.282
JavaScript	34.400	1.383
Kotlin	92.580	641
MATLAB	75.825	645
Perl	91.522	1.280
PHP	55.910	651
Python	62.305	1.280
R	52.556	645
Ruby	183.486	1.297
Scala	87.155	753
SQL	57.615	1.283
Swift	61.315	640
TypeScript	143.397	1.307
Unix Shell	93.111	640
Total	1.474.584	19.759

Tabla 3. Estadísticas con el número de ficheros y el tamaño del corpus construido.

En la Tabla 3 sólo figuran 21 lenguajes de los 25 seleccionados en primera instancia. Esto se debe a que para los lenguajes Makefile, Objective-C, VBA y Visual Basic.NET, el *crawler* no fue capaz de descargar tantos datos como para el resto de lenguajes. Más concretamente, para ninguno de estos cuatro lenguajes fue posible descargar más de 50 MB de ficheros. Esto supone un 3% del volumen descargado con respecto al lenguaje con más datos (Go, con 1.454 MB descargados). Por esta razón, optamos por excluir estos cuatro lenguajes para el proyecto actual, dando prioridad al balanceado de datos, a costa de disminuir el número de lenguajes que el sistema presentado es capaz de clasificar.

Finalmente, la mayor diferencia en cuanto al tamaño de los ficheros del corpus la tenemos entre el lenguaje Go, cuyos datos tienen 2,27 veces más tamaño que los de C# (que cuenta con 640 MB descargados). El *crawler* siguió funcionando para conseguir más

datos de los cuatro lenguajes excluidos. Sin embargo, el corpus para este proyecto se dejó cerrado en marzo de 2021, por cuestiones de tiempo, para poder comenzar con los procesos de verificación, procesado y limpieza, y generación del conjunto de datos final. Así, el corpus final para este proyecto contiene ficheros de 21 lenguajes de programación distintos, 1.474.584 ficheros y un volumen de 19.759 MB.

5.2. Verificación del corpus de ficheros

Para el proceso de verificación automática previamente descrito, procedimos a ejecutar el módulo *code_validator* sobre los conjuntos de ficheros pertenecientes a aquellos lenguajes de programación de los que disponemos de gramática. Para ello ejecutamos la herramienta con un *timeout* de 10 segundos por fichero, ya que la complejidad y tamaño del código fuente contenido en algún fichero hace que la herramienta se ralentice o incluso se bloqueé esperando por el veredicto de algunas gramáticas. El resultado de esta fase se muestra en la Tabla 4.

Lenguaje	Ficheros	Ficheros verificados	Ficheros sin verificar	% ficheros verificados	Ficheros admitidos	Ficheros no admitidos	% admitido
Assembly	44.626	44.626	0	100,0%	23.457	21.169	52,5%
C	34.686	34.686	0	100,0%	11.473	23.213	33,0%
C++	23.714	23.674	40	99,8%	18.584	5.090	78,3%
C#	56.494	56.492	2	99,9%	54.010	2.482	95,6%
CSS	14.840	14.840	0	100,0%	13.187	1.653	88,8%
Go	76.733	76.712	21	99,9%	76.563	149	99,8%
HTML	22.143	21.946	197	99,1%	21.003	943	95,7%
Java	110.168	110.168	0	100,0%	110.027	139	99,8%
JavaScript	34.399	34.399	0	100,0%	31.188	3.211	90,6%
Kotlin	92.580	92.571	9	99,9%	89.620	2.951	96,8%
MATLAB	75.825	-	75.825	0,0%	-	-	
Perl	91.522	-	91.522	0,0%	-	-	
PHP	55.910	54.910	1.000	98,2%	54.908	2	99,9%
Python	62.304	62.304	0	100,0%	59.133	3.171	94,9%
R	52.555	52.555	0	100,0%	45.845	6.710	87,2%
Ruby	183.486	-	183.486	0,0%	-	-	
Scala	87.155	87.148	7	99,9%	68.031	19.117	78,0%
SQL	57.615	57.151	135	99,7%	43.031	14.120	75,2%
Swift	61.315	61.309	6	99,9%	57.542	3.767	93,8%
TypeScript	143.397	143.371	26	99,9%	87.187	56.184	60,8%
Unix Shell	93.111	92.769	342	99,6%	66.915	25.854	72,1%

Tabla 4. Estadísticas acerca del proceso de verificación automática, como el número de ficheros a verificar, o el porcentaje de ficheros verificados y el de ficheros admitidos para cada lenguaje de programación.

La imposibilidad de implementar una versión aceptable de hasta tres gramáticas (Ruby, Perl y Matlab), unida a la baja eficacia de otras como la C y Assembly puso de manifiesto

la necesidad de complementar la verificación automática añadiendo una segunda fase de análisis manual al proceso.

5.2.1. Verificación manual

Una vez disponemos de los resultados de la verificación automática de cada lenguaje, llevamos a cabo la fase de verificación manual. Si los resultados de un lenguaje muestran un porcentaje de ficheros admitidos con respecto al total, superior al 60%, se selecciona una muestra de aleatoria de 50 de sus ficheros no admitidos para analizarlos manualmente. Por el contrario, si los resultados muestran un porcentaje inferior al 60% o la gramática es inexistente, se selecciona una muestra aleatoria de 100 ficheros de entre sus no admitidos o de sus no evaluados en caso de que no exista gramática. El resultado de esta fase se puede observar en la Tabla 5.

	Ficheros analizados a mano	Ficheros ajenos al lenguaje	Ficheros pertenecientes al lenguaje	% Ficheros pertenecientes
Assembly	100	0	100	100%
C	100	0	100	100%
C++	50	0	50	100%
C#	50	0	50	100%
CSS	50	0	50	100%
Go	50	1	49	98%
HTML	50	0	50	100%
Java	50	1	49	98%
JavaScript	50	0	50	100%
Kotlin	50	0	50	100%
MATLAB	100	0	100	100%
Perl	100	2	98	98%
PHP	2	0	2	100%
Python	50	0	50	100%
R	50	3	47	94%
Ruby	100	0	100	100%
Scala	50	0	50	100%
SQL	50	0	50	98%
Swift	50	0	50	100%
TypeScript	50	0	50	100%
Unix Shell	50	1	49	98%

Tabla 5. Estadísticas acerca del proceso de verificación manual, como el número de ficheros a verificar o el porcentaje de ficheros pertenecientes para cada lenguaje de programacion.

En la Tabla 5, observamos que hay 5 lenguajes para los que durante el análisis manual se encontró algún fichero ajeno al lenguaje. Para 4 de ellos (Java, Go, R y Unix Shell) la verificación automática se llevó a cabo previamente, al disponer de gramática, obteniendo en el peor de los casos, Unix Shell, un 72,1% de ficheros verificados por el reconocedor sintáctico. Para este caso peor, el 27,9% de los ficheros fueron rechazados por el reconocedor, de los cuales (extrapolando los datos del análisis de la muestra),

únicamente el 2% de los ficheros serían ajenos al lenguaje. Esto hace un total de un 0,5% de ruido con respecto a la totalidad de los ficheros, para este caso peor, que consideramos suficientemente bajo como para ser considerado un mecanismo de regularización [Igl2019]. Realizando los mismos cálculos para los otros 3 lenguajes avalados por la verificación automática obtenemos un 0,7% de ruido para R, que también consideramos como adecuado, y un ruido insignificante para Java y Go del 0,004%.

Por otro lado, tenemos el caso de Perl, para el que no pudimos efectuar la fase de verificación automática, y durante el análisis manual encontramos 2 ficheros (un 2%) ajenos al lenguaje. Este porcentaje de ruido ya puede considerarse más significativo, pero hay que precisar que los dos ficheros ajenos encontrados durante el análisis manual tienen de media 4,5 líneas de código, mientras que la totalidad de ficheros del corpus para el lenguaje Perl tiene una media de 234,13 líneas. Si partimos de la muestra manual y asumimos que para el 98% de los ficheros pertenecientes al lenguaje la media de líneas será aproximadamente 234, y la media de líneas de código para los ficheros ajenos es de 4,5, obtenemos un porcentaje de ruido (medido sobre líneas de código) del 0,04%. Por lo que, finalmente decidimos que también era adecuado y no tuvimos que excluir a Perl de nuestro conjunto de 21 lenguajes, ni refinar o extender el proceso de verificación descrito.

5.3. Generación del conjunto de datos

Para el procesamiento de los ficheros verificados decidimos eliminar las construcciones más propensas a contener lenguaje natural: los comentarios de una línea y varias líneas, y las cadenas de texto multilínea. Sin embargo, decidimos conservar las cadenas de una sola línea por aquellos lenguajes donde las cadenas pueden contener código como puede ser Python o HTML. Además, conservamos las cadenas interpoladas en aquellos lenguajes que las admiten, ya que, además de lenguaje natural, contienen código fuente. Este procesamiento es totalmente dependiente del lenguaje, al tener los distintos lenguajes construcciones distintas, usadas de forma diferente por los programadores. Procedemos a detallar los distintos procesamientos realizados para cada lenguaje en la tabla 6.

Adicionalmente, eliminamos todos los caracteres en blanco o tabulaciones situadas al principio y al final de cada línea de código.

Nuestro proyecto tiene como objetivo la clasificación de una única línea de código, para estudiar qué eficiencia se puede alcanzar al realizar la clasificación de porciones de código muy pequeñas. Además, fijamos un tamaño mínimo de línea de 10 caracteres, considerando dicho valor como la cantidad mínima de información que los modelos con características a nivel de carácter pueden utilizar de forma efectiva para diferenciar su lenguaje la de entre las 21 posibilidades (cabría realizar un análisis de diferencias en los distintos valores de esta variable). Por lo que, después de realizar el procesamiento descrito anteriormente, eliminamos del conjunto de datos las líneas que no superan esta longitud.

Lenguaje de programación	Construcciones eliminadas
Assembly	Comentarios de línea
C	Comentarios de línea Comentarios multilínea
C++	Comentarios de línea Comentarios multilínea Cadenas multilínea
C#	Comentarios de línea Comentarios multilínea Comentarios de documentación XML Cadenas multilínea
CSS	Comentarios de línea Comentarios multilínea
Go	Comentarios de línea Comentarios multilínea Cadenas multilínea
HTML	Comentarios de línea Comentarios multilínea
Java	Comentarios de línea Comentarios multilínea
JavaScript	Comentarios de línea Comentarios multilínea
Kotlin	Comentarios de línea Comentarios multilínea Comentarios multilínea anidados Cadenas multilínea
MATLAB	Comentarios de línea Comentarios multilínea
Perl	Comentarios de línea
PHP	Comentarios de línea Comentarios multilínea
Python	Comentarios de línea Cadenas multilínea
R	Comentarios de línea
Ruby	Comentarios de línea Comentarios multilínea
Scala	Comentarios de línea Comentarios multilínea
SQL	Comentarios de línea Comentarios multilínea
Swift	Comentarios de línea Comentarios multilínea Comentarios anidados Cadenas multilínea
TypeScript	Comentarios de línea Comentarios multilínea
Unix Shell	Comentarios de línea

Tabla 6. Construcciones eliminadas para cada lenguaje de programación durante el procesamiento de lenguajes.

Para la generación del conjunto de datos final:

1. Leemos todos los ficheros del corpus junto con su etiqueta numérica relativa a su lenguaje de programación.
2. Barajamos todos los ficheros entre sí.
3. Extraemos el contenido de los ficheros, decodificándolo de UTF-16 Little Endian (formato utilizado por el *crawler*), y codificándolo a UTF-8.
4. Dividimos el contenido de cada fichero en líneas y barajamos las líneas de cada fichero.
5. Intercalamos las distintas líneas barajadas de los ficheros.
6. Llegados a este punto lo ideal sería barajar todas las líneas del conjunto de datos, pero por cuestiones de eficiencia, dividimos el conjunto en *batches* de tamaño 512 y barajamos estos *batches* entre sí.
7. Barajamos las líneas de cada *batch* y deshacemos los *batches* para volver a disponer de un conjunto de datos de líneas de código.
8. Separamos el 95% de los individuos del conjunto para el entrenamiento y el 5% para validación. Quedando 732,5M y 38,5M de instancias respectivamente.
9. Transformamos cada línea en un vector numérico con el código Unicode de sus distintos caracteres.
10. Transformamos cada etiqueta relativa al lenguaje de programación a formato *one-hot.*
11. Truncamos las líneas a 40 caracteres como máximo. A pesar de que una mayor longitud de línea aporta más información al modelo, también requiere de una mayor cantidad de parámetros y, por tanto, recursos, siendo éstos limitados. De esta forma, conseguimos reducir tiempos de entrenamiento y validación, y pudimos realizar el entrenamiento con *batches* de mayor tamaño. Además, esto nos permitió cargar una parte mayor del conjunto de datos directamente en memoria, reduciendo más los tiempos de entrenamiento.
12. Definimos un vocabulario cerrado de caracteres para los modelos. Dentro de los distintos caracteres Unicode, seleccionamos un rango cerrado que contiene los operadores usados en los distintos lenguajes, los espacios, los dígitos y las letras del alfabeto occidental en mayúsculas y minúsculas. En resumen, los caracteres más utilizados en nuestro *dataset*. Este rango abarca del código 32 al 126, ambos inclusive, como se puede observar en la Figura 12. Además, elegimos un rango cerrado de códigos, para facilitar la implementación de la generación del conjunto de datos. Así, sustituimos todos los números fuera de este rango por un token especial *OOV (Out Of Vocabulary)*, el número 31.
13. Transformamos los valores numéricos para que estén dentro en un rango continuo que empiece en el 0, requisito para la transformación a formato *one-hot*, y para la entrada de los *embeddings* utilizados en la arquitectura de las redes neuronales. Para ello, restamos 30 a cada número, de forma que el 0 lo reservamos como carácter especial y el token OOV pasa de ser el 31 al 1.
14. Rellenamos (*padding*) de las líneas que no alcanzan los 40 caracteres, rellenando todo el espacio sobrante con ceros. De esta forma, obtenemos un conjunto de datos de vectores de tamaño 40, con todos los números en el rango [0-96], constituyendo un vocabulario cerrado de códigos numéricos de tamaño 97.

15. Seleccionamos 700M de individuos para el conjunto de entrenamiento y 1M para el de validación, para agilizar el entrenamiento y la evaluación de los modelos.
16. Finalmente, lo convertimos en matrices de la librería numpy y lo serializamos para poder cargarlo de forma eficiente y usarlo para el proceso de entrenamiento.

0	NUL	32	ESPACIO	64	@	96	`	128	€	160	¡	192	À	224	à
1	SOH	33	!	65	A	97	a	129	•	161	¢	193	Á	225	á
2	STX	34	"	66	B	98	b	130	‚	162	£	194	Â	226	â
3	ETX	35	#	67	C	99	c	131	ƒ	163	¤	195	Ã	227	ã
4	EOT	36	$	68	D	100	d	132	„	164	¥	196	Ä	228	ä
5	ENQ	37	%	69	E	101	e	133	…	165	¦	197	Å	229	å
6	ACK	38	&	70	F	102	f	134	†	166	§	198	Æ	230	æ
7	BEL	39	'	71	G	103	g	135	‡	167	¨	199	Ç	231	ç
8	BS	40	(	72	H	104	h	136	ˆ	168	©	200	È	232	è
9	HT	41	)	73	I	105	i	137	‰	169	ª	201	É	233	é
10	LF	42	*	74	J	106	j	138	Š	170	«	202	Ê	234	ê
11	VT	43	+	75	K	107	k	139	‹	171	¬	203	Ë	235	ë
12	FF	44	,	76	L	108	l	140	Œ	172	-	204	Ì	236	ì
13	CR	45	-	77	M	109	m	141	•	173	®	205	Í	237	í
14	SO	46	.	78	N	110	n	142	Ž	174	¯	206	Î	238	î
15	SI	47	/	79	O	111	o	143	•	175	°	207	Ï	239	ï
16	DLE	48	0	80	P	112	p	144	•	176	±	208	Ð	240	ð
17	DC1	49	1	81	Q	113	q	145	‘	177	²	209	Ñ	241	ñ
18	DC2	50	2	82	R	114	r	146	’	178	³	210	Ò	242	ò
19	DC3	51	3	83	S	115	s	147	“	179	´	211	Ó	243	ó
20	DC4	52	4	84	T	116	t	148	”	180	µ	212	Ô	244	ô
21	NAK	53	5	85	U	117	u	149	•	181	¶	213	Õ	245	õ
22	SYN	54	6	86	V	118	v	150	–	182	·	214	Ö	246	ö
23	ETB	55	7	87	W	119	w	151	—	183	¸	215	×	247	÷
24	CAN	56	8	88	X	120	x	152	˜	184	¹	216	Ø	248	ø
25	EM	57	9	89	Y	121	y	153	™	185	º	217	Ù	249	ù
26	SUB	58	:	90	Z	122	z	154	š	186	»	218	Ú	250	ú
27	ESC	59		91	[	123	{	155	›	187	¼	219	Û	251	û
28	FS	60	<	92	\	124	|	156	œ	188	½	220	Ü	252	ü
29	GS	61	=	93	]	125	}	157	•	189	¾	221	Ý	253	ý
30	RS	62	>	94	^	126	~	158	ž	190	¿	222	Þ	254	þ
31	US	63	?	95	_	127	DEL	159	Ÿ	191	À	223	ß	255	ÿ

Figura 12. Conjunto de caracteres para UTF-8 donde la parte sombreada indica el subconjunto seleccionado para conformar el vocabulario de los modelos de aprendizaje automático de este trabajo.

5.4. Arquitectura de la red neuronal

Como se puede observar en la Figura 13, la arquitectura de nuestro modelo está formada por una capa de entrada, una capa de *embeddings*, 3 capas ocultas y una capa *softmax* de salida. El tamaño de la capa de entrada está determinado por el tamaño de las instancias. Cada neurona en la capa de entrada corresponde a una característica

específica de la instancia, lo cual implica que, como nuestros individuos están formados por 40 códigos numéricos, nuestra capa de entrada deberá contener 40 neuronas.

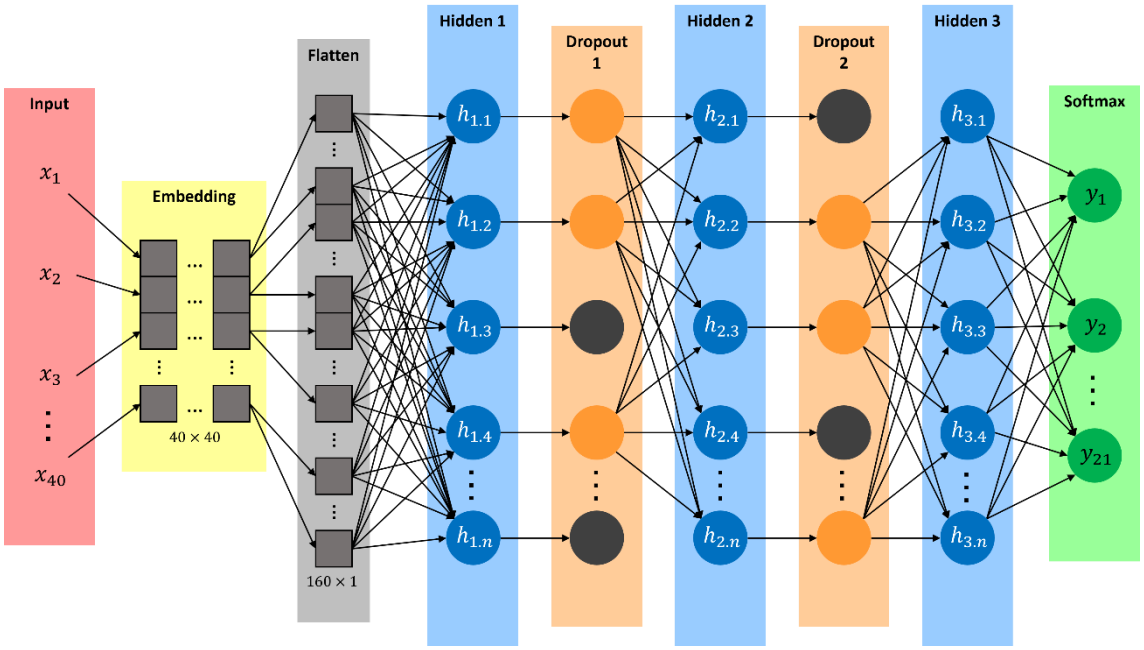

Figura 13. Arquitectura base de nuestros modelos de red neuronal.

Para evitar la codificación de las instancias del conjunto de datos a formato *one-hot*, incluimos *embeddings* como primera capa de la arquitectura de nuestra red. Esta decisión se basa principalmente en la dimensionalidad de los datos, ya que los *embeddings* nos ayudan a reducirla y, por consiguiente, nos permiten entrenar el modelo con *batches* más grandes y cargar muchos más datos directamente en memoria. Utilizamos *embeddings* de 40 dimensiones, obteniendo como salida una matriz de 40 dimensiones por 40 características.

Como ya hemos mencionado, la salida de la capa de *embeddings* es una matriz de dimensión 40x40. Sin embargo, las capas ocultas de la arquitectura están diseñadas para trabajar con vectores unidimensionales, por lo que necesitamos añadir una capa *flatten* para transformar estas matrices a vectores de dimensión 160, concatenando los vectores de tamaño 40 de cada elemento de la entrada.

En cuanto al número de capas densas, establecimos que tres era el número mínimo de capas para crear una red neuronal profunda capaz de aprender características y patrones de alto nivel. No elegimos un número de capas superior, por falta de recursos temporales y de hardware. Para reducir el número de hiperparámetros, decidimos que todas las capas ocultas estuvieran compuestas por el mismo número de neuronas, configurable según el modelo que se quiera utilizar. Nuestra arquitectura también incluye dos capas de *dropout*, una entre cada par de capas densas, con el fin de poder estudiar el impacto de este método de regularización en la eficiencia de clasificación de nuestros modelos. Ambas capas de *dropout* comparten la misma ratio (configurable) de *dropout*, con el fin de reducir, de nuevo, el número de hiperparámetros del modelo

Finalmente, para utilizar modelos de redes neuronales en un problema de clasificación multiclase como el nuestro, la capa de salida de la arquitectura tiene que ser de tipo *softmax*, y debe de tener un número de neuronas igual al número de categorías a clasificar. En nuestro caso buscamos clasificar 21 lenguajes de programación, por lo que nuestra capa de salida deberá contener 21 neuronas. Así, la salida de esta capa *softmax* será un vector de 21 números reales, con el que no sólo podemos determinar la clase a la que pertenece una entrada, sino que representa las probabilidades de que la entrada pertenezca a cada una de las clases.

5.5. Entrenamiento de la red

El optimizador elegido para los modelos fue el *Nesterov Accelerated Gradient (NAG)* [Nesterov1983], una variante del *Stochastic Gradient Descent (SGD),* con una tasa de aprendizaje del 0,001 y un *momentum* de 0,9. Según la bibliografía estudiada [Géron2022], este optimizador provee una velocidad de convergencia media, superior a otros optimizadores como *SGD;* y una calidad de convergencia muy buena, superando incluso a otros optimizadores más sofisticados como *AdaGrad*, *RMSprop*, *Adam* o *Nadam*. Para la tasa de aprendizaje y el *momentum*, elegimos los valores mencionados porque, según la bibliografía [Géron2022] es una configuración ampliamente utilizada y funciona muy bien en combinación con el optimizador *NAG* elegido.

En cuanto a la función de pérdida empleada, utilizamos *categorical cross entropy*, diseñada para calcular la pérdida en problemas de clasificación multiclase y con las etiquetas en formato *one-hot.* Además, aprovechamos el hecho de tener un conjunto de datos muy grande (700M de individuos) para simplificar el proceso de generación y no tener que balancear estos datos. Sabemos que la clase más representada del conjunto de datos cuenta con 68.815.659 instancias (un 9,8% del total), mientras que la clase menos representada tiene 20.581.790 (un 2,9% del total). Según la bibliografía analizada [Géron2022], cuando se disponen de millones de instancias para cada clase, como es el caso, tener un conjunto de datos totalmente balanceado no es necesario para obtener una mejor eficiencia de clasificación, siempre que se pondere la función de pérdida según la representación de las distintas clases. Para calcular los pesos de cada clase, utilizamos la función `compute_class_weight`, de la librería scikit-learn, que nos permite ponderar en mayor grado las instancias de las clases menos representadas y viceversa.

Para determinar el número de *epochs* del entrenamiento de los modelos usamos el *early stopping*. De esta forma, prevenimos el sobreajuste de los modelos entrenados, establecemos un criterio homogéneo de parada para todos ellos y, a la vez, facilitamos la obtención de modelos que generalizan mejor sobre el conjunto de validación. Además, con el fin de priorizar los modelos que sean capaces de clasificar de forma correcta mayor número de instancias, establecimos como variable a monitorizar la exactitud (*accuracy*) del modelo sobre el conjunto de validación. Por último, utilizamos una paciencia de 2, lo que implica que el entrenamiento finalizará cuando, a lo largo de dos *epochs* consecutivos, no se mejore la exactitud del mejor modelo entrenado hasta el momento. Elegimos este valor porque observamos que, a partir de 2 iteraciones sin

mejora, el modelo empezaba a oscilar, pudiendo perder el carácter regulador del *early stopping*. Además, también observamos que, a partir de este punto, la exactitud de los modelos ya empezaba a converger, por lo que este valor de paciencia nos ayuda a minimizar los tiempos de entrenamiento, sin prácticamente perder eficiencia de clasificación.

Finalmente, seleccionamos un tamaño de *batch* de 16.384. A través de un experimento con distintos tamaños observamos que, hasta llegar a este número, el tiempo de entrenamiento es inversamente proporcional al tamaño de *batch.* Esto implica que usar un *batch* el doble de grande agilizaba el tiempo de entrenamiento para un *epoch* a la mitad. También observamos que la exactitud de los modelos decrecía ligeramente al emplear tamaños más grandes, pero nunca de manera proporcional. Por lo que, de nuevo para minimizar los tiempos del entrenamiento, nos decantamos por este valor. Nuestra GPU llegaba a admitir un tamaño de *batch* incluso mayor, de 32.768, sin embargo, el tiempo de entrenamiento de cada *epoch* ya no se reducía a la mitad, por lo que lo descartamos.

5.6. Búsqueda de hiperparámetros

Para esta fase, únicamente pudimos experimentar con la variación de tres hiperparámetros, de nuevo, por cuestiones de tiempo: el número de neuronas ocultas de cada capa, la función de activación de las capas ocultas y la ratio de *dropout*.

Los valores utilizados para el número de neuronas de las capas ocultas fueron 194, 1.000, 2.000 y 3.000. Esto se debe a que los primeros experimentos los realizamos con 194 neuronas en cada capa. Este número, relativamente pequeño, nos permitió experimentar de forma ágil para constatar la validez de los distintos hiperparámetros elegidos para la arquitectura y el entrenamiento de los modelos, presentados en las dos secciones anteriores. Posteriormente, fuimos aumentando la capacidad de los modelos y estudiando los resultados, hasta que, de nuevo por cuestiones de tiempo no pudimos probar con más valores.

En cuanto a las funciones de activación, seleccionamos las más modernas y sofisticadas: SELU, ELU y Leaky Relu. Según la bibliografía estudiada [Géron2022], estas funciones de activación tienen mayor rendimiento en general que otras funciones más primitivas como Relu, Tanh o Logistic. A pesar de que cada capa puede tener una función de activación distinta, decidimos que todas las capas ocultas del mismo experimento deberían utilizar esa misma función de activación para reducir el número de hiperparámetros. Además, basándonos también en [Géron2022], elegimos el esquema de inicialización de pesos que mejor funcionaba para cada función de activación. Así, empleamos la inicialización de He [He2015] para la función de activación SELU, y la inicialización de LeCun [LeCun2002] para ELU y LeakyRelu. También hay que mencionar que, excepto SELU, la cual incorpora una forma de autorregulación que ayuda a mantener una salida de activación equilibrada y una propagación estable de gradientes, el resto de las funciones de activación se utilizaron junto con capas de normalización (*batch normalization layer*). De esta forma, evitamos el problema del desvanecimiento

del gradiente *(gradient vanishing problem),* ayudando a los modelos profundos a converger.

Finalmente, se quiso estudiar el impacto de otro mecanismo de regularización además del *early stopping*, por lo que realizamos experimentos sin *dropout*, y con un *dropout* de 0,2 y 0,1. No probamos con ratios mayores de *dropout* porque, según la bibliografía [Géron2022], para estos problemas relacionados con el procesamiento de lenguajes, un valor mayor suele ser improductivo y tiene el riesgo de tener un impacto negativo en los modelos.

5.7. Otros modelos de aprendizaje automático entrenados

Además de medir el rendimiento de nuestros propios modelos de redes neuronales, llevamos a cabo la ejecución del *benchmark* Lazy Predict [LazyPredict2023], en su versión 0.2.12. De esta forma, evaluamos la eficiencia de distintos modelos de aprendizaje automático a la hora de realizar la tarea de clasificación con nuestros mismos datos, constituyendo un marco de comparación con los modelos de redes neuronales implementados. Lazy Predict es una herramienta capaz de construir una gran cantidad de modelos para ayudar a comprender cuales de ellos funcionan mejor sin hacer búsqueda de hiperparámetros. Para el entrenamiento de los modelos, Lazy Predict carga en memoria todo el conjunto de datos, que debe estar compuesto por vectores numéricos de la misma longitud para que los algoritmos de aprendizaje puedan funcionar. Además, teniendo en cuenta que nuestros datos están compuestos por los valores numéricos de los distintos caracteres (variables categóricas nominales), y que ninguno de los modelos del *benchmark* provee un mecanismo análogo a los *embeddings*, cada elemento de estos vectores numéricos debe ser convertido a formato *one-hot.* A continuación, se describe de forma somera los 25 modelos distintos que Lazy Predict es capaz de construir, entrenar y evaluar:

AdaBoostClassifier: Este algoritmo está basado en el concepto de ensamble. Combina múltiples clasificadores débiles en un modelo fuerte mediante un proceso iterativo de ajuste de pesos en las instancias de entrenamiento. En cada iteración, se da más peso a las instancias clasificadas incorrectamente para enfocarse en los casos más difíciles. Los clasificadores débiles se combinan mediante votación ponderada, donde los clasificadores con mejor rendimiento tienen más influencia en la decisión final. AdaBoostClassifier es efectivo en problemas de clasificación binaria y multiclase, pero puede ser sensible a valores atípicos y ruido en los datos de entrenamiento.

BaggingClassifier: Este es un algoritmo que utiliza la técnica de *bagging* para combinar múltiples modelos base. Mediante el muestreo aleatorio con reemplazo, se generan diferentes subconjuntos de datos para entrenar cada modelo. Luego, las predicciones de cada modelo se combinan mediante votación para determinar la clase final. Esta técnica ayuda a reducir la varianza y mejorar la generalización, al promediar las predicciones de varios modelos. BaggingClassifier es especialmente útil para mitigar el impacto de datos ruidosos o atípicos en la clasificación. Una variante conocida es el Random Forest, que utiliza árboles de decisión como modelos base.

BernoulliNB es un algoritmo de aprendizaje automático utilizado para la clasificación de datos binarios, donde las características son variables binarias (0 o 1). Se basa en el teorema de Bayes y asume independencia condicional entre las características. Estima las probabilidades a priori y condicionales de cada clase basándose en la presencia o ausencia de características binarias en los datos de entrenamiento. Luego, utiliza estas estimaciones para clasificar nuevas instancias calculando la probabilidad posterior de cada clase y seleccionando la clase con la mayor probabilidad. BernoulliNB es rápido y eficiente, especialmente para datos dispersos con muchas características, pero es importante tener en cuenta que solo considera la presencia o ausencia de características sin tener en cuenta su frecuencia.

CalibratedClassifierCV es una clase en scikit-learn que se utiliza para calibrar las probabilidades estimadas de los clasificadores. Su objetivo principal es corregir la calibración deficiente de las probabilidades y mejorar la confiabilidad de las predicciones. Utiliza una estrategia de validación cruzada interna para ajustar una curva de calibración a las salidas de probabilidad del clasificador base. Esto permite obtener estimaciones más precisas de las probabilidades y predecir las probabilidades calibradas para nuevas instancias. CalibratedClassifierCV es útil cuando se requiere una estimación precisa de las probabilidades y se desea utilizar un clasificador que no proporciona estimaciones de probabilidad bien calibradas de forma nativa.

DecisionTreeClassifier: Este algoritmo construye un árbol de decisiones para tomar decisiones basadas en preguntas sobre características del conjunto de datos. Cada nodo del árbol representa una pregunta y cada rama representa una posible respuesta. Durante el entrenamiento, se busca dividir los datos de manera que se maximice la pureza de las clases en cada nodo. El árbol resultante puede ser utilizado para clasificar nuevas instancias siguiendo el camino desde la raíz hasta un nodo hoja. DecisionTreeClassifier es interpretable y capaz de manejar diferentes tipos de datos, pero puede sufrir de *overfiting*. Se pueden aplicar técnicas como la poda o la combinación con métodos de ensamble para mejorar su rendimiento en datos no vistos.

ExtraTreeClassifier: Este algoritmo es muy semejante a ExtraTreesClassifier aunque su principal diferencia radica en el mecanismo de división de los árboles. ExtraTreeClassifier utiliza divisiones aleatorias en cada nodo, lo que lo hace más rápido pero potencialmente más sensible al ruido. Otra diferencia es que ExtraTreeClassifier construye un solo árbol, mientras que su semejante construye un conjunto de árboles formando un bosque. Además, ExtraTreeClassifier devuelve la predicción del árbol individual, mientras que ExtraTreesClassifier realiza una votación o promedio de las predicciones de todos los árboles para obtener la predicción final.

ExtraTreesClassifier: Este es otro algoritmo basado en Random Forest (bosques aleatorios) pero que utiliza una estrategia adicional de aleatorización. Construye múltiples árboles de decisión utilizando subconjuntos aleatorios de características y puntos de corte seleccionados de forma adicionalmente aleatoria. Esto ayuda a reducir el *overfiting* y mejorar la generalización del modelo. El ExtraTreesClassifier puede manejar características categóricas y numéricas, pero puede ser computacionalmente más costoso y menos interpretable que los árboles de decisión estándar.

GaussianNB: Este algoritmo de clasificación está basado en el teorema de Bayes, el cual asume una distribución gaussiana para las características. Estima los parámetros de la

distribución gaussiana para cada clase durante el entrenamiento y utiliza estos parámetros para calcular la probabilidad de pertenencia a cada clase durante la predicción. Es eficiente, adecuado para conjuntos de datos grandes y puede manejar características continuas. Sin embargo, puede no ser adecuado para características categóricas o problemas con dependencias entre características. GaussianNB es apreciado por su simplicidad y capacidad de manejar características continuas.

KNeighborsClassifier: Este algoritmo basado en vecinos cercanos utiliza las etiquetas de las instancias vecinas más cercanas para asignar una etiqueta de clase a una instancia desconocida. Para ello, calcula la distancia entre la instancia desconocida y todas las instancias de entrenamiento, selecciona los k vecinos más cercanos y determina la clase más frecuente entre ellos. KNeighborsClassifier es flexible y puede manejar datos numéricos y categóricos, aunque su rendimiento depende de la elección adecuada de k y la métrica de distancia. Este algoritmo se basa en el almacenamiento de los datos de entrenamiento en memoria para realizar clasificaciones posteriores.

LabelPropagation: Este algoritmo, al igual que el anterior realiza tareas de clasificación semi-supervisada pero propaga probabilidades de clase a través de conexiones en un grafo. Utiliza la similitud entre instancias y pesos de conexiones para propagar etiquetas desde instancias etiquetadas a instancias no etiquetadas. El algoritmo busca un equilibrio entre la consistencia local y la coherencia global. LabelPropagation también es útil cuando parte de los datos no disponen de etiquetas, pero se quieren aprovechar para mejorar el modelo. Es importante tener en cuenta la calidad de la matriz de afinidad y los parámetros seleccionados para obtener resultados precisos en la propagación de etiquetas.

LabelSpreading: Este algoritmo de clasificación semi-supervisada propaga las etiquetas desde instancias etiquetadas a instancias no etiquetadas en función de la similitud entre ellas. Utiliza un modelo de grafos y una matriz de afinidad para medir la similitud y asignar pesos a las conexiones. El algoritmo permite la propagación flexible de etiquetas y puede manejar problemas de clasificación multiclase. Sin embargo, se deben considerar posibles errores de propagación y seleccionar cuidadosamente los parámetros para obtener resultados óptimos. LabelSpreading es útil cuando se dispone de datos con un número limitado de etiquetas y se desea aprovechar la información de los datos no etiquetados para mejorar las predicciones.

LGBMClassifier: Este algoritmo de aprendizaje automático se basa en *gradient boosting* para realizar la tarea de clasificación. Se destaca por su implementación optimizada y eficiente utilizando el algoritmo de *gradient boosting* basado en histogramas. Utiliza la técnica de *binning* para agrupar características y construir histogramas, lo que permite una búsqueda rápida de las mejores divisiones en cada nodo del árbol. LGBMClassifier ofrece una alta velocidad de entrenamiento y predicción, puede manejar conjuntos de datos grandes y trabajar con características categóricas sin necesidad de codificación previa. También cuenta con parámetros ajustables para controlar la complejidad del modelo y prevenir el *overfiting*.

LinearDiscriminantAnalysis (LDA): LDA es un algoritmo de aprendizaje automático utilizado para clasificación y reducción de dimensiones. Busca encontrar una combinación lineal de características que maximice la separación entre clases y minimice la varianza dentro de cada clase. LDA se basa en supuestos de clases

linealmente separables y distribución gaussiana de características. Puede proporcionar una interpretación de las características más discriminativas y se puede utilizar para reducir la dimensionalidad de los datos. Sin embargo, puede tener dificultades en escenarios con clases superpuestas o características irrelevantes o altamente correlacionadas.

LinearSVC: Este algoritmo pertenece a la familia de métodos de Support Vector Machines (SVM), pero se enfoca en problemas de clasificación lineal basándose en el principio de margen máximo. LinearSVC es especialmente eficiente en términos computacionales en comparación con otros algoritmos SVM cuando se trabaja con conjuntos de datos grandes. Sin embargo, no puede resolver problemas con un conjunto de datos no linealmente separables sin utilizar transformaciones. Además, no proporciona directamente estimaciones de probabilidad.

LogisticRegression: Este algoritmo, aunque el nombre sugiere "regresión", en realidad es un clasificador utilizado especialmente en aquellos problemas en los que las variables objetivo son binarias, es decir, solo tienen dos clases. Modela la relación entre las características y la probabilidad de pertenecer a una clase utilizando la función logística. Es eficiente, proporciona probabilidades estimadas y se puede generalizar a problemas de clasificación multiclase. Sin embargo, asume una relación lineal entre las características y las probabilidades, por lo que puede requerir transformaciones adicionales en algunos casos.

NearestCentroid: Este es un algoritmo de clasificación simple y eficiente que se basa en la idea de los centroides más cercanos. Calcula los centroides para cada clase y asigna la clase con el centroide más cercano como la etiqueta de predicción. Es útil cuando las clases tienen centroides bien definidos y separados, pero puede ser sensible a los valores atípicos. No puede aprender interacciones complejas entre características y solo utiliza información de los centroides para realizar predicciones, por lo que puede no ser adecuado para problemas con relaciones no lineales o dependencias complejas entre características.

PassiveAggressiveClassifier: Este algoritmo es utilizado en problemas de clasificación en línea donde los datos llegan de forma secuencial y se procesan uno por uno. Cuando se comete un error, actualiza los parámetros del modelo de manera "agresiva" y cuando clasifica correctamente, los actualiza de manera "pasiva". Es eficiente en términos computacionales y se adapta rápidamente a los cambios en los datos, lo que lo hace adecuado para flujos de datos en tiempo real. Sin embargo, puede ser sensible a la escala de las características y requiere la elección adecuada de parámetros para un buen rendimiento.

Perceptron: Este algoritmo de aprendizaje automático se utiliza en clasificación binaria. Ajusta los pesos de las características para realizar la clasificación basándose en un modelo inspirado en las neuronas del cerebro humano. Es un algoritmo simple y eficiente, pero solo puede clasificar conjuntos de datos linealmente separables. A pesar de sus limitaciones, el Perceptron ha sido fundamental en el desarrollo de algoritmos más avanzados y se puede utilizar en problemas de clasificación en tiempo real debido a su velocidad.

QuadraticDiscriminantAnalysis (QDA): Este algoritmo de clasificación supervisada, a diferencia del algoritmo LinearDiscriminantAnalysis (LDA), no asume igualdad de covarianzas entre las clases. En cambio, QDA permite que las covarianzas varíen entre las clases. Utiliza la función de densidad multivariable para estimar la probabilidad de pertenecer a cada clase. Al clasificar una nueva instancia, QDA calcula las probabilidades posteriores y asigna la etiqueta de clase con la mayor probabilidad. QDA es útil cuando las covarianzas de las clases son diferentes, lo que proporciona un modelo más flexible. Sin embargo, puede requerir más parámetros y ser más propenso al *overfiting* en conjuntos de datos pequeños en comparación con LDA.

RandomForestClassifier: Este es un algoritmo basado en la idea de Random Forest (bosques aleatorios), el cual es un método de ensamblaje de árboles de decisión. Construye un conjunto de árboles de decisión independientes utilizando subconjuntos aleatorios de características y puntos de corte. Combina las predicciones de los árboles mediante votación mayoritaria para determinar la clase final de una muestra. Es eficiente, puede manejar características categóricas y numéricas, y proporciona una estimación de la importancia de las características. Sin embargo, puede ser menos interpretable que un solo árbol de decisión y la importancia de las características puede verse afectada por la correlación.

RidgeClassifier: Este algoritmo se basa en el modelo de regresión Ridge y aplica regularización para controlar el *overfiting*. El parámetro de regularización, alfa, debe ser especificado manualmente por el usuario. El RidgeClassifier ajusta un modelo de regresión logística utilizando un enfoque iterativo para encontrar los coeficientes que maximizan la verosimilitud de los datos de entrenamiento y minimizan la penalización de Ridge. Es eficiente y adecuado para conjuntos de datos grandes. Sin embargo, asume una relación lineal entre las características y las probabilidades logarítmicas de las clases.

RidgeClassifierCV: Este algoritmo está basado en el anterior, RidgeClassifier pero su principal diferencia radica en la selección del parámetro de regularización. Mientras que el RidgeClassifier requiere que el usuario especifique manualmente el valor de alfa, el RidgeClassifierCV utiliza validación cruzada para seleccionar automáticamente el mejor valor de alfa. Esto hace que el RidgeClassifierCV sea más conveniente, ya que automatiza la selección de hiperparámetros y proporciona métricas de rendimiento evaluadas a través de la validación cruzada.

SGDClassifier: Este es un algoritmo de clasificación eficiente que utiliza el descenso de gradiente estocástico para entrenar modelos en conjuntos de datos grandes. Es versátil y puede manejar características numéricas y categóricas, ofreciendo flexibilidad al permitir diferentes funciones de pérdida y regularizaciones. Sin embargo, debido a su enfoque estocástico, es importante ajustar correctamente los hiperparámetros y seguir el proceso de entrenamiento.

Support Vector Classifier (SVC): SVC es un algoritmo basado en máquinas de vectores de soporte que busca encontrar el hiperplano óptimo que mejor separa las instancias de diferentes clases en un espacio de alta dimensión. SVC utiliza funciones de *kernel* y técnicas de regularización para mejorar la capacidad de generalización del modelo. Es un enfoque poderoso en clasificación y puede manejar tanto problemas de clasificación

binaria como multiclase. Sin embargo, es importante ajustar adecuadamente los parámetros del *kernel* y la regularización para obtener un buen rendimiento del modelo.

XGBClassifier: Este algoritmo es un es un potente clasificador basado en Gradient Boosting llamado XGBoost (Extreme Gradient Boosting). Es eficiente, escalable y se utiliza en problemas de clasificación complejos. Utiliza árboles de decisión para construir un modelo más fuerte y corregir errores secuencialmente. XGBoost ofrece una mayor capacidad de control y ajuste de hiperparámetros, proporciona una selección automática de características y puede manejar conjuntos de datos de gran escala. Sin embargo, es importante ajustar adecuadamente los hiperparámetros para obtener un mejor rendimiento.

5.8. Evaluación

Para medir la eficiencia de clasificación de los distintos modelos de aprendizaje automático utilizamos las siguientes métricas:

1. Exactitud (*Accuracy*): Es la proporción de instancias correctamente clasificadas sobre el total de instancias. Se calcula dividiendo el número de instancias correctamente clasificadas (verdaderos positivos y verdaderos negativos) entre el número total de instancias. La exactitud es una métrica comúnmente utilizada, pero puede ser engañosa si las clases están desbalanceadas.

$$Exactitud = \frac{(verdaderos\ positivos\ +\ verdaderos\ negativos)}{Total\ de\ instancias}$$

Figura 14. Métrica Exactitud (Accuracy).

2. Precisión (*Precision*): Mide la proporción de verdaderos positivos sobre el total de predicciones positivas. La precisión es útil cuando es importante evitar falsos positivos, es decir, minimizar los casos en los que se clasifican erróneamente instancias negativas como positivas.

$$Precisión = \frac{verdaderos\ positivos}{(verdaderos\ positivos\ +\ falsos\ positivos)}$$

Figura 15. Métrica Precisión (Precision).

3. Exhaustividad o sensibilidad (*Recall*): Mide la proporción de verdaderos positivos sobre el total de instancias positivas. La exhaustividad es útil cuando es importante evitar falsos negativos, es decir, minimizar los casos en los que se clasifican erróneamente instancias positivas como negativas.

$$Exhaustividad = \frac{verdaderos\ positivos}{(verdaderos\ positivos\ +\ falsos\ negativos)}$$

Figura 16. Métrica Exhaustividad o Sensibilidad (Recall).

4. Valor-F (*F1-Score*): Es una métrica que combina la precisión y el recall en un solo valor. El F1-Score proporciona una medida equilibrada entre la precisión y el recall, y es especialmente útil cuando las clases están desbalanceadas.

$$Valor - F = \frac{2 * (precisión * exhaustividad)}{(precisión + exhaustividad)}$$

Figura 17. Métrica Valor-F (F1-Score).

Además de estas 4 métricas, también utilizamos la matriz de confusión (*Confusion Matrix*), que muestra visualmente la concordancia o discrepancia entre las clasificaciones realizadas por un modelo y las clases reales a las que pertenecen los datos. Consiste en una tabla que muestra el recuento de las clasificaciones correctas e incorrectas realizadas por un modelo. De esta forma, ayudan a visualizar, analizar y discutir el desempeño de un modelo de clasificación, permitiendo una evaluación más precisa de su exactitud y de errores cometidos en la clasificación de datos.

5.8.1. Evaluación de los modelos de Lazy Predict

Para la evaluación del *benchmark* Lazy Predict, establecimos la siguiente metodología, con el objetivo de evitar el entrenamiento de los 25 modelos incluidos para todo el conjunto de datos, ya que esto sería muy costoso en términos de tiempo. Comenzamos por entrenar los 25 modelos sobre un pequeño subconjunto de datos de únicamente 700 instancias, para observar su rendimiento y, de esa forma, tener un punto de partida a la hora de incrementar el número de instancias. Tras analizar la exactitud, el valor-F y el tiempo de ejecución obtenidos por cada modelo, decidimos ejecutar otra prueba con todos los modelos incrementando el número de instancias a 7.000. En esta prueba se apreció un aumento en los tiempos de ejecución, especialmente para algunos modelos cuyo entrenamiento tenía complejidad temporal cuadrática. Debido a esto, para la siguiente prueba, efectuada sobre un subconjunto de 70.000 instancias, sólo se seleccionaron los 5 modelos con mayor exactitud de la prueba anterior. Finalmente, se decidió realizar una última prueba, evaluando esos mismos 5 modelos, pero, esta vez con el tamaño máximo posible de instancias: 3,5 millones. Este tamaño viene determinado por la memoria RAM, ya que los modelos del *becnhmark* no admiten en entrenamiento online, es decir, cargan de forma ansiosa todo el conjunto de datos en memoria. Esto, sumado al uso de la codificación *one-hot* que requieren estos algoritmos, hace que la memoria RAM disponible no permita entrenar los modelos con más de 3,5 millones de instancias.

5.8.2. Evaluación del trabajo relacionado

Para la comparación de nuestros modelos con los sistemas actuales, partiendo de los trabajos relacionados estudiados, seleccionamos aquellos entrenados para clasificar a nivel de *snippet* y basados únicamente en código fuente. Como se puede observar en la siguiente tabla, identificamos cuatro sistemas que cumplen estos requisitos: [Alreshedy2018], [Alrashedy2020] #3, [Yang2021] y [Guesslang2023], por lo que se pueden considerar cómo los más cercanos a nuestro trabajo. Finalmente, por cuestiones

de tiempo, únicamente seleccionamos el de mayor rendimiento para incluirlo en la evaluación y compararlo con nuestros modelos. Así, seleccionamos Guesslang 2.2.1, que tiene un 93,4% de exactitud y, además, es capaz de clasificar los 21 lenguajes seleccionados para este proyecto, es el más moderno de los sistemas estudiados, es una herramienta profesional, gratis, y de fácil instalación y uso:

Publicación	N.º de lenguajes	Tamaño de entrada	Exactitud obtenida	Tipo de entrada
[Khasnabish2014]	10	Fichero completo	93,48%	Código fuente
[VanDam2016]	20	Fichero completo	97,50%	Código fuente
[Gilda2017]	60	Fichero completo	97,00%	Código fuente
[Alreshedy2018]	21	Snippet	75,00%	Código fuente
[Hong2019] #1	10	Snippet	92,00%	Imagen
[Hong2019] #2	5	Función	99,00%	Imagen
[Alrashedy2020] #1	21	Snippet	88,90%	Código fuente y lenguaje natural
[Alrashedy2020] #2	21	-	78,90%	Lenguaje natural
[Alrashedy2020] #3	21	Snippet	78,10%	Código fuente
[Kiyak2020] #1	8	Fichero completo	98,81%	Código fuente
[Kiyak2020] #2	8	Fichero completo	99,38%	Imagen
[Yang2021]	19	Snippet	87,40%	Código fuente
[Öztürk2023]	24	Snippet	85,50%	Código fuente y lenguaje natural
[Guesslang2023]	54	Snippet	93,40%	Código fuente

Tabla 7. Estadísticas acerca de los trabajos relacionados, como el tamaño y el tipo de entrada o la exactitud obtenida por cada uno de ellos.

La evaluación de Guesslang se llevó a cabo sobre nuestro conjunto de validación de 1 millón de instancias. Sin embargo, Guesslang está entrenado para distinguir entre 54 lenguajes de programación distintos, cuando nuestros datos sólo contienen líneas de código de 21 de estos lenguajes. Por ello, se tuvieron que adaptar las predicciones de Guesslang a nuestro problema de clasificación, para llevar a cabo una comparación legítima. Aprovechando que cada predicción emitida por Guesslang es un vector de números reales con las distintas probabilidades de que determinado *snippet* pertenezca a cada una de las 54 clases, simplemente seleccionamos el lenguaje con mayor probabilidad que perteneciera al conjunto de 21 lenguajes contemplados para nuestro proyecto.

6. Resultados

6.1. Resultados de las redes neuronales

6.1.1. Análisis del impacto del número de neuronas

La Figura 18 presenta una comparativa de la exactitud obtenida por varios modelos de red neuronal en función del número de *epochs* durante los que han sido entrenados, donde cada modelo tiene distinto número de neuronas por capa. Para hacer una comparación justa, estos modelos comparten el resto de los hiperparámetros variables en nuestra configuración: la función de activación SELU y la ausencia de *dropout*.

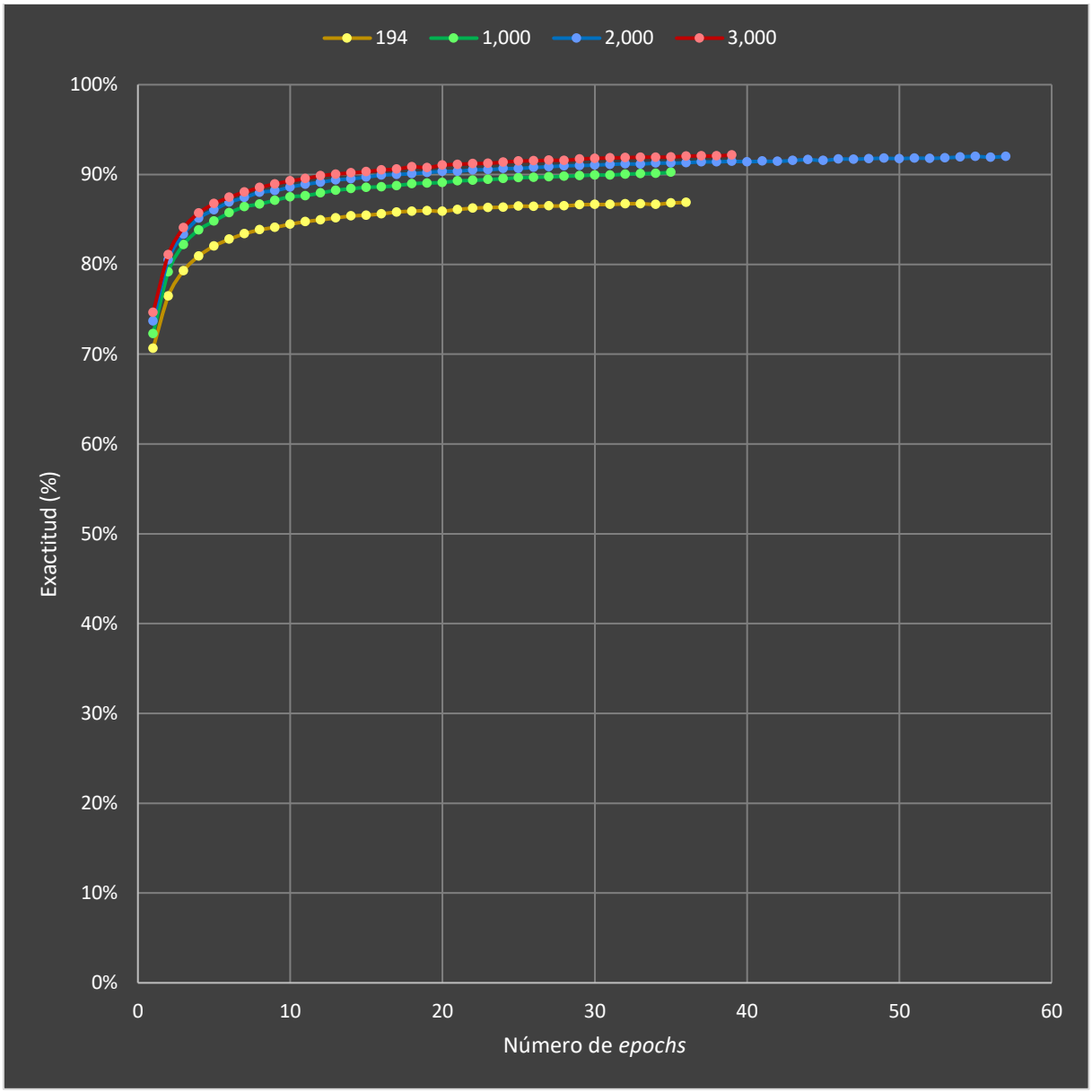

Figura 18. Comparativa de la exactitud obtenida por distintos modelos de red neuronal donde cada uno de ellos contiene diferente número de neuronas por capa: 194, 1.000, 2.000 y 3.000.

El eje X de la gráfica representa el número de *epochs*, mientras que el eje Y representa el rendimiento del modelo, medido en términos de exactitud. Cada curva representa un modelo con un número de neuronas por capa específico: 194, 1.000, 2.000 y 3.000. Con este experimento buscamos analizar el impacto directo del número de neuronas en el rendimiento de la red neuronal y determinar si más neuronas resultan en una mejora de la eficiencia de clasificación.

Observando la Figura 18, se puede apreciar una tendencia de mejora en la exactitud a medida que se aumenta el número de neuronas por capa en los modelos y por tanto su capacidad. El modelo con 3.000 neuronas es el que alcanza la mayor exactitud con un valor del 92,18%. Le sigue el modelo con 2.000 neuronas, que obtiene una exactitud ligeramente inferior del 92,02%. En tercer lugar, se encuentra el modelo con 1.000 neuronas, el cual presenta una exactitud del 90,4%. Por último, el modelo con 194 neuronas muestra la menor exactitud con un valor del 86,88%. Esta tendencia se debe a la capacidad de los modelos con más neuronas para capturar patrones más complejos, lo que les permite realizar predicciones más precisas.

Sin embargo, al analizar la diferencia de exactitud entre los modelos, podemos observar una tendencia interesante. A medida que aumentamos el número de neuronas, la diferencia de exactitud entre ellos se reduce. El modelo con 1.000 neuronas muestra una diferencia de exactitud de 1,79% con respecto al modelo con 2.000 neuronas, mientras que la diferencia de exactitud entre el modelo con 2.000 neuronas y el modelo con 3.000 neuronas es de sólo el 0,16%. Esta tendencia puede indicar que se está alcanzando el límite de capacidad útil del modelo y añadir más neuronas por capa no produce mejoras notables. Cabe recordar que, como ya mencionamos en la motivación, debido al uso de líneas de código de al menos 10 caracteres como instancias del modelo, este problema no puede obtener el 100% de exactitud.

La tendencia de todos estos modelos es muy similar. Inicialmente, los modelos experimentan un crecimiento más acelerado, pero finalmente comienzan a converger hasta que se cumple el criterio de parada y el entrenamiento finaliza. Este comportamiento queda reflejado en la tasa media de crecimiento por *epoch* obtenida de los cuatro modelos en distintos intervalos del eje X. Entre el *epoch* 1 y 4 se sitúa la zona de mayor crecimiento con una mejora del 3,7% de media por *epoch*. A continuación, del *epoch* 4 al 12, se puede apreciar el codo de la curva, donde los modelos comienzan a reducir su crecimiento, pasando del 3,7% del intervalo anterior, a un 0,51% de media por *epoch*. Finalmente, del *epoch* 12 en adelante, la convergencia se hace más evidente, donde la media de crecimiento cae hasta el 0,08% por *epoch*.

Otro aspecto a destacar es el comportamiento del modelo con 2.000 neuronas por capa, el cual alcanza un mayor número de *epochs* con respecto al resto de modelos, obteniendo un rendimiento cercano al modelo de 3.000 neuronas. Esto se debe a que a medida que avanza su entrenamiento, este modelo experimenta un crecimiento más lento y oscilatorio, pero también más consistente, por lo que el criterio de parada tarda más en cumplirse, aumentando ligeramente la exactitud final alcanzada.

6.1.2. Análisis del impacto del tipo de función de activación

La Figura 19 presenta una comparativa de la exactitud obtenida por varios modelos de red neuronal, donde cada modelo utiliza una función de activación distinta. Para hacer una comparación justa, estos modelos comparten la misma configuración para el resto de los hiperparámetros: 1.000 neuronas por capa oculta y la ausencia de *dropout*. El eje X de la gráfica representa el número de *epochs*, mientras que el eje Y representa el rendimiento del modelo, medido en términos de exactitud. Cada curva representa un modelo con una función de activación específica: SELU, ELU y Leaky ReLU. Con esta configuración buscamos analizar el impacto directo de la función de activación en el rendimiento de la red neuronal y determinar cuál de ellas obtiene una mejor eficiencia de clasificación.

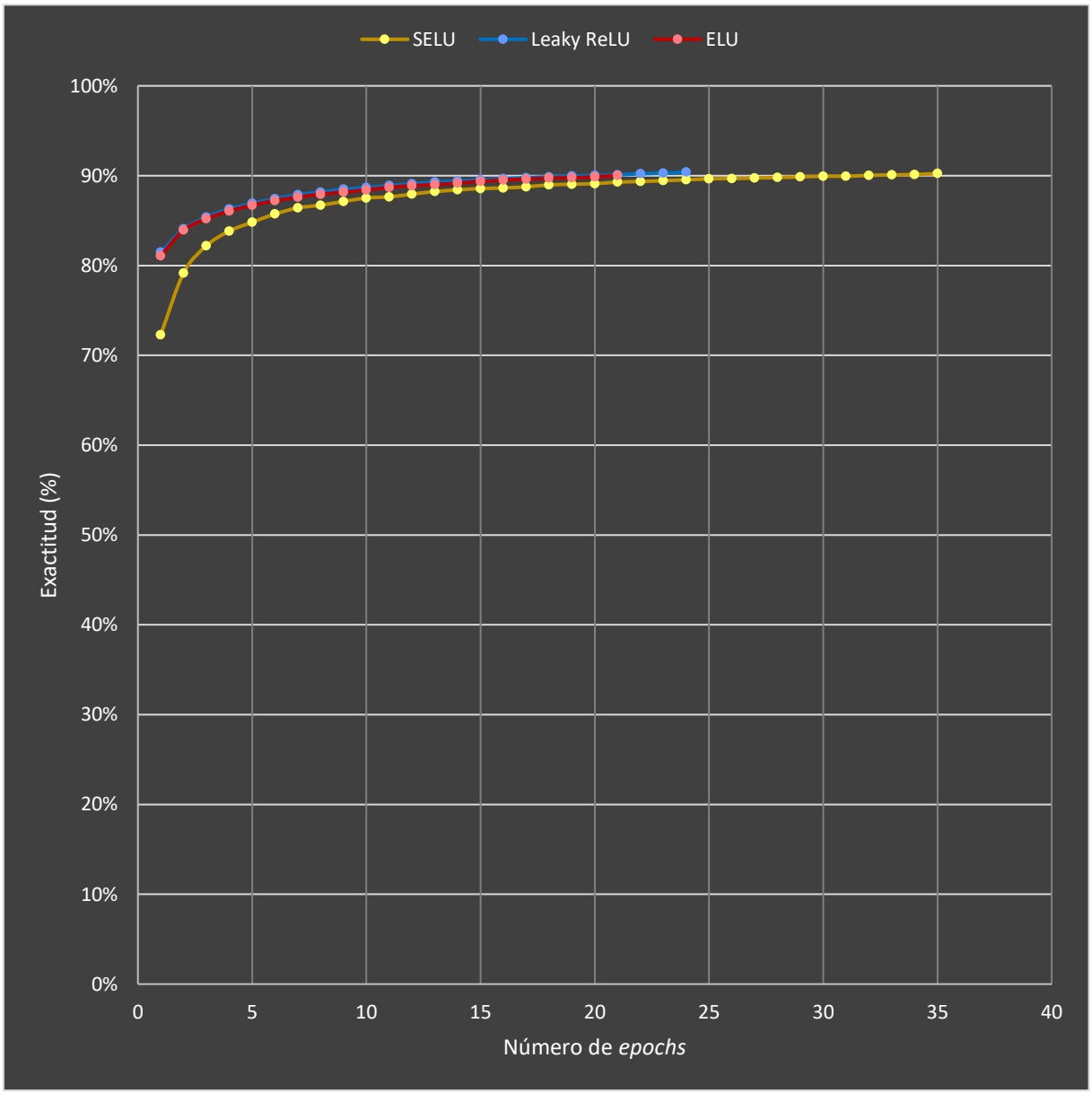

Figura 19. Comparativa de la exactitud obtenida por distintos modelos de red neuronal donde cada uno de ellos utiliza distinta función de activación: SELU, ELU y Leaky ReLU.

Como muestra la Figura 19, los modelos alcanzan una exactitud muy similar. El modelo con la función de activación Leaky ReLU obtiene la mayor exactitud, con un valor de 90,4%. Por otro lado, los modelos con las funciones de activación SELU y ELU tienen una exactitud ligeramente menor, con valores de 90,24% y 90,02% respectivamente. Sin embargo, por cuestiones de tiempo no fue posible entrenar los modelos que usan las funciones ELU y Leaky ReLU hasta cumplir al criterio de parada definido por el *early stopping*. Teniendo esto en cuenta, si nos fijamos en el último *epoch* para el que tenemos datos de los tres modelos (*epoch 21*), tenemos que el mejor rendimiento lo ofrece la función de activación Leaky ReLU, con un 90,10% de exactitud, seguida muy de cerca por ELU, con un 90,02%, y por último SELU con un 89,28%. Sería muy interesante terminar este experimento porque parece que, a medida que los modelos convergen, el que usa SELU se va acercando al rendimiento de los otros dos modelos. En el *epoch* 3, por ejemplo, la diferencia entre SELU y Leaky ReLU es del 3,18%, mientras que en el *epoch* 21, esta diferencia se reduce hasta un 0,82%.

En general, la tendencia de los modelos es similar a la descrita en los experimentos anteriores, donde los modelos comienzan con un crecimiento acelerado para, finalmente, tender a converger. Sin embargo, en este caso se puede apreciar cómo los modelos que utilizan ELU y Leaky ReLU, experimentan un crecimiento inicial menos acelerado que el de SELU. Esto se ve reflejado en la tasa media de crecimiento por *epoch* obtenida hasta el *epoch* 13. Por un lado, con un comportamiento prácticamente idéntico, tenemos los modelos con ELU y Leaky ReLU, los cuales logran un crecimiento medio por *epoch* del 0,66% y 0,65%. Por el otro lado, tenemos el modelo con SELU, el cual obtiene un crecimiento medio de 1,33% por *epoch*.

6.1.3. Análisis del impacto del *dropout*

La Figura 20 presenta una comparativa de la exactitud obtenida por varios modelos de red neuronal, donde cada modelo utiliza una tasa de *dropout* distinta. Para hacer una comparación adecuada, estos modelos comparten la misma configuración para el resto de los hiperparámetros: 3.000 neuronas por capa oculta y la función de activación SELU. El eje X de la gráfica representa el número de *epochs*, mientras que el eje Y representa el rendimiento del modelo, medido en términos de exactitud. Cada curva representa un modelo con una tasa de *dropout* específica: 0, 0,1 y 0,2. Con esta configuración buscamos analizar el impacto directo de la tasa de *dropout* en el rendimiento de la red neuronal.

Como podemos observar en la Figura 20, el modelo con mayor rendimiento es el que no usa *dropout*, alcanzando un máximo de 92,18% de exactitud, seguido del modelo con una tasa de dropout del 0,2 (89,76% de exactitud) y del modelo con 0,1 (85,70% de exactitud). Sin embargo, por cuestiones de tiempo no fue posible entrenar los dos modelos que usan *dropout* hasta cumplir al criterio de parada definido por el *early stopping*. Teniendo esto en cuenta, si nos fijamos en el último *epoch* para el que tenemos datos de los tres modelos (*epoch* 13), tenemos que el mejor modelo sigue siendo el que no usa *dropout* con un 90,04% de exactitud. Pero, el modelo con tasa de *dropout* de 0,1 exhibe un 0,92% más de exactitud que el de *dropout* 0,2. Con esta

evidencia limitada, observamos que una mayor tasa de *dropout* implica un menor rendimiento de los modelos. Aunque, sería muy interesante terminar este experimento porque parece que, a medida que los modelos convergen, el modelo con tasa 0,1 de *dropout* se acerca cada vez más a la eficiencia del modelo con 0,2 de *dropout.* En el *epoch* 5, por ejemplo, esta diferencia llega a ser del 2,47%, y para el *epoch* 13 se reduce a tan sólo un 0,92%. A su vez, el modelo con tasa de 0,2 de *dropout* se va acercando al rendimiento del modelo sin *dropout*. En el *epoch* 5, esta diferencia es del 7,63%, mientras que, en el *epoch* 39, se reduce a un 2,62%.

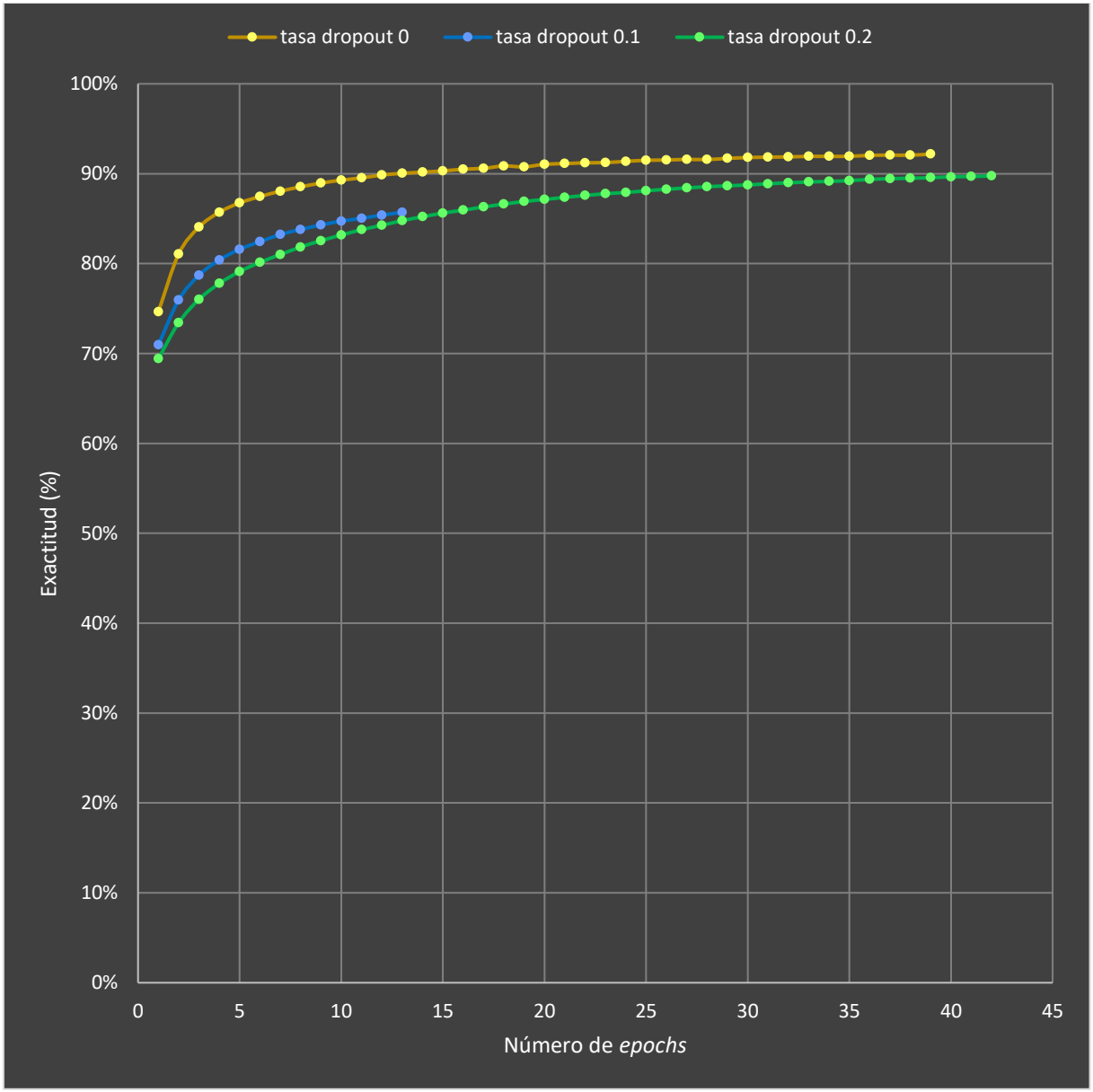

Figura 20. Comparativa de la exactitud obtenida por distintos modelos de red neuronal donde cada uno de ellos utiliza distinta tasa de dropout: 0, 0,1 y 0,2.

La tendencia de los modelos es similar al de los anteriores experimentos, exhibiendo un mayor crecimiento en los primeros *epochs* para finalmente comenzar a converger. Sin embargo, en este caso podemos apreciar como los modelos con *dropout* crecen de manera más constante, especialmente al comienzo del experimento. Si nos fijamos en

el intervalo del *epoch* 1 al 4, el modelo con una tasa del 0,2 crece un 2,79% de media por *epoch*, algo por debajo del 3,15% del modelo con 0,1, y ambos por debajo del 3,69% del modelo sin *dropout*. Superado el codo de la curva, se puede ver como comienza a caer el crecimiento, aunque los modelos con *dropout* son los que menos acusan esta caída, demostrando una convergencia más lenta, debido a la regularización. De nuevo, sería interesante finalizar los experimentos para cuantificar el número de *epochs* necesarios para que los modelos con *dropout* finalicen su convergencia, activando la condición de parada definida por el *early stopping*.

6.1.4. Resumen de los resultados obtenidos

La siguiente tabla muestra los valores obtenidos según diferentes métricas para cada uno de nuestros modelos de red neuronal, ordenados de mayor a menor exactitud. Las columnas de "Número de neuronas por capa", "Función de activación" y "*Dropout"*, hacen referencia a la configuración de los hiperparámetros que fuimos variando para cada modelo.

Número de neuronas por capa	Función de activación	*Dropout*	Precisión	Exhaustividad	Valor-F	Exactitud
3.000	SELU	-	0,9068	0,9157	0,9109	0,9218
2.000	SELU	-	0,9047	0,9140	0,9089	0,9202
1.000	Leaky ReLU	-	0,8857	0,8969	0,8907	0,9040
1.000	SELU	-	0,8847	0,8957	0,8896	0,9024
1.000	ELU	-	0,8817	0,8937	0,8870	0,9002
3.000	SELU	0,2	0,8780	0,8909	0,8836	0,8976
194	SELU	-	0,8469	0,8615	0,8530	0,8688
3.000	SELU	0,1	0,8334	0,8495	0,8399	0,8570
Promedio			**0,8777**	**0,8897**	**0,8829**	**0,8965**

Tabla 8. Rendimiento de los modelos de red neuronal representado con distintas métricas.

Basándonos en la Tabla 8 podemos observar que las métricas de precisión, exhaustividad y valor-F son siempre más bajas que la Exactitud de su correspondiente modelo, siendo de media un 1,88%, 0,68%, 1,36% más bajas respectivamente. Concretamente, nuestro mejor modelo obtiene una Exactitud del 92,18%, lo que supone un 1,5% más que su precisión, un 0,61% más que su exhaustividad y un 1,09% más alta que su valor-F. Esta diferencia por parte de la exactitud es la consecuencia de realizar la validación y el criterio de parada en base a dicha métrica, por lo que el entrenamiento estaba orientado a maximizarla. También puede haber influido el que los datos de *dataset* no estén balanceados, tendiendo a elegir las clases mayoritarias en caso de dificultad en la clasificación.

6.1.5. Análisis de las predicciones emitidas

La Figura 21 muestra una matriz de confusión con el porcentaje (redondeado a dos decimales) de las clasificaciones correctas e incorrectas realizadas por nuestro mejor

modelo (con 3.000 neuronas por capa oculta, función de activación SELU y sin *dropout*). Gracias a ella, podemos averiguar las predicciones donde el modelo tiene más dificultad en interpretar qué lenguajes son más difíciles de diferenciar y en qué medida.

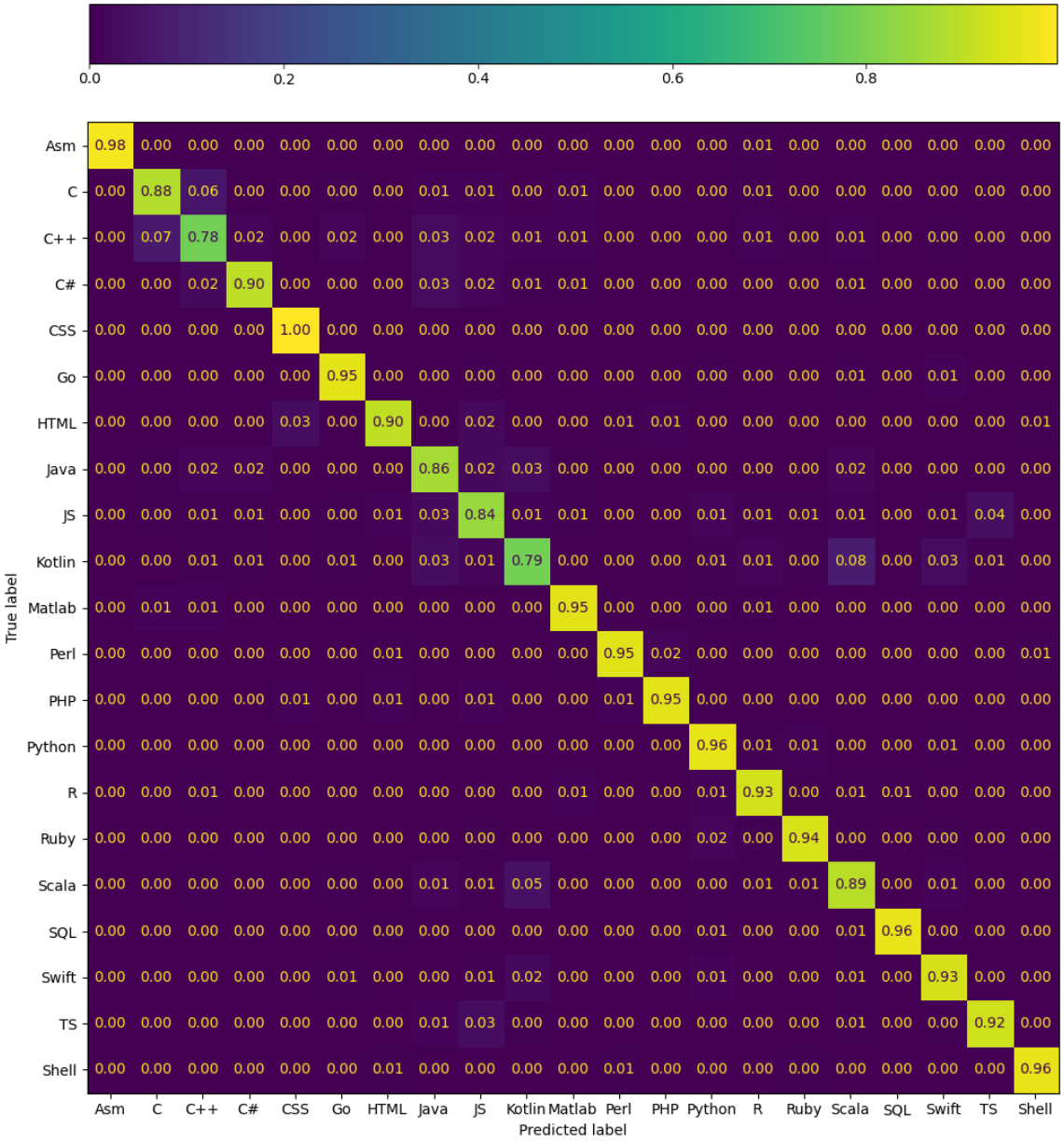

Figura 21. Matriz de confusión donde se representan las clasificaciones correctas e incorrectas obtenidas por el mejor modelo de red neuronal para cada lenguaje de programación.

Por un lado, destaca la clasificación de ficheros CSS, donde la probabilidad de confusión es prácticamente nula. En este caso, la matriz muestra un porcentaje perfecto por cuestiones de redondeo, pero en realidad es del 99,7%. También se puede observar que la identificación de ficheros en Assembly (Asm), con un 98% de acierto, es realmente alta, confundiéndolos mínimamente con el lenguaje R. Otros lenguajes como Go, Matlab, PHP, R, SQL y Shell, también exhiben un porcentaje de acierto alto de entre el 93% y el 96%, y que no exceden el 1% de confusión con ningún otro lenguaje.

Fijándonos en Perl, podemos apreciar como a pesar de tener un porcentaje de acierto del 95%, se confunde un 2% con PHP. Esto puede deberse a que ambos lenguajes utilizan el símbolo $ para denotar variables. Al presentar todos estos lenguajes exactitudes por encima de la media, podemos interpretar que son lenguajes bastante característicos, y que en nuestro conjunto de lenguajes no existe ningún otro lo suficientemente parecido como para causar confusiones notables en su clasificación.

Por otro lado, lenguajes como C y C++ generan más de confusión, lo cual es normal debido a que ambos comparten muchas de sus características. Por ejemplo, C, se confunde un 6% con C++, mientras que éste se confunde un 7% con el primero. Esto hace que la probabilidad de acierto de C se resienta, siendo inferior que la media, con un 88%. En el caso de C++, la exactitud obtenida es mucho menor, del 78%, debido a que, al ser más moderno que C, se confunde entre un 2% y un 3% con otros lenguajes posteriores, que se inspiraron en él, como C#, Java y JavaScript. C#, por su parte, tiene un porcentaje de acierto de 90%, donde su mayor confusión es con Java y C++ en un 3% y 2% respectivamente, ya que C# fue diseñado con la intención de ser amigable para los desarrolladores de Java y C++. Además, debido a su sintaxis estilo C y su orientación a objetos, el modelo también confunde a C# con JavaScript en un 2%.

HTML es un lenguaje orientado a desarrollo web que comúnmente suele utilizarse junto con los lenguajes CSS o JavaScript, los cuales, en numerosas situaciones suelen estar embebidos en cualquier parte del documento HTML. Esta circunstancia, probablemente sea el motivo por el que su identificación se confunda con CSS y JavaScript, en un 3% y un 2% respectivamente. Además, es curioso ver cómo, en sentido contrario, esta relación no existe o no es tan marcada, ya que los ficheros CSS no incluyen HTML y en JavaScript tampoco es habitual.

En el caso de Java, se puede apreciar que tiene una probabilidad de acierto inferior a la media, del 86%. Esto se debe a que Java es un lenguaje orientado a objetos que tiene una sintaxis estilo C. Por lo que no es de extrañar que el modelo lo confunda en un 2% con lenguajes como C++, C# y JavaScript. Además, Scala y Kotlin, son lenguajes que fueron diseñados partiendo de Java y que contienen varias características similares, como bien refleja el 2% y 3% de confusión respectivamente. JavaScript, por su parte, se clasifica de forma correcta sólo un 84% de las veces debido a que se confunde principalmente con TypeScript (4%), el cual es un transpilador del propio JavaScript, por lo que comparten muchas características. Como también se puede observar, el modelo confunde JavaScript con Java en un 3%, lenguaje del que adopta convenciones y con el que comparte algunos aspectos de su sintaxis. Por otro lado, TypeScript, al ser un superconjunto de JavaScript, exhibe una confusión del 3% con este lenguaje, cuando se intenta identificar un código fuente de TypeScript.

Analizando Kotlin, se puede ver cómo el modelo lo confunde con Scala en un 8%, lo que tiene sentido, si tenemos en cuenta aspectos como que, ambos son lenguajes que vienen a resolver limitaciones de Java, que ambos se ejecutan sobre la JVM, que ambos soportan el paradigma funcional. A consecuencia de esta relación con Java, también se puede apreciar como el modelo confunde el 2% de las veces, a Kotlin con Java. Por último, la confusión del 3% con Swift, otro lenguaje enfocado a la programación de

aplicaciones para dispositivos móviles, que además permite la programación orientada a objetos y el paradigma funcional. Del mismo modo que el modelo tiende a confundir Kotlin con Scala, también se puede apreciar una confusión a la inversa, en la cual se confunde Scala con Kotlin en un 5% debido a los mismos motivos. También se puede observar que, por los mismos motivos que nuestro modelo confunde Kotlin con Swift, también confunde Swift con Kotlin en un 2%.

Finalmente, al momento de identificar Ruby, nuestro modelo tiende a confundirlo con Python en un 2%, probablemente debido a que ambos se utilizan para *scripting*, son de tipado dinámico y soportan la programación orientada a objetos.

6.1.6. Análisis de tiempos de entrenamiento

Numero de neuronas por capa	Función de activación	*Dropout*	Numero de parámetros del modelo	Tiempo medio de entrenamiento por *epoch* (seg.)
194	SELU	-	674.365	9.222
1.000	SELU	-	3.627.901	15.202
2.000	SELU	-	11.251.901	32.400
3.000	SELU	-	22.875.901	60.697
1.000	ELU	-	3.627.901	20.278
1.000	Leaky ReLU	-	3.627.901	21.229
3.000	SELU	0,1	22.875.901	65.096
3.000	SELU	0,2	22.875.901	65.124

Tabla 9. Tiempos consumidos por epoch durante el entrenamiento de los modelos de red neuronal.

Basándonos en la Tabla 9, podemos apreciar que los modelos con mayor número de parámetros necesitan un mayor tiempo de ejecución por *epoch*. El número de parámetros de un modelo viene determinado principalmente por el número de neuronas de sus capas. Si nos fijamos en los cuatro primeros modelos, los cuales solo se diferencian en el número de neuronas de sus capas ocultas, podemos observar como un mayor número de neuronas por capa incide directamente en el tiempo de entrenamiento. El modelo de 1.000 neuronas es 1,6 horas (5.980 segundos) más costoso de entrenar por *epoch* que el modelo de 194 neuronas, siendo su entrenamiento 1,65 veces más lento. Con 2.000 neuronas se necesitan 4,7 horas (17.198 segundos) más de tiempo en cada *epoch* que con 1.000 neuronas, siendo este modelo 2,13 veces más lento de entrenar. Finalmente, el modelo con 3.000 neuronas tarda 7,8 horas (28.297 segundos) más que el de 2.000, siendo 1,87 veces más costoso. Este aumento de tiempos se debe al incremento del número de neuronas, el cual incide directamente sobre el número de parámetros del modelo. Estos parámetros intervienen en el cálculo de la función de pérdida, su derivada y son modificados en cada *epoch* del entrenamiento. Por lo que el aumento del número de parámetros incide directamente en el tiempo computacional que requiere el entrenamiento del modelo.

Además del número de neuronas, hay varios aspectos que también influyen en el tiempo de entrenamiento por *epoch*. Uno de estos aspectos es el *dropout*. Si observamos los dos últimos modelos de la tabla, los que varían el hiperparámetro *dropout*, nos damos cuenta de que su tiempo de entrenamiento por *epoch* es superior a su modelo

equivalente sin *dropout*, el cuarto de la tabla. Así, el modelo con 0,1 de *dropout* es 1,08 veces más lento de entrenar que el mismo modelo sin *dropout*.

Por otra parte, la función de activación también afecta al tiempo de entrenamiento. Si comparamos el segundo modelo de la tabla, el cual utiliza la función SELU, con los modelos quinto y sexto, los cuales utilizan ELU y Leaky ReLU respectivamente, observamos que Leaky ReLU es la función de activación más lenta (1,4 veces más que SELU), seguida de ELU (1,33 más lenta que SELU) y siendo SELU la más rápida. Esto se debe a que la función SELU tiende a autorregularse. Por el contrario, ELU y Leaky ReLU necesitan de una capa extra de normalización para regular su salida, lo que añade más complejidad al modelo.

Como ya hemos comentado, cabe destacar que el tiempo de entrenamiento fue una de las principales limitaciones para nuestra experimentación, reduciendo las combinaciones de hiperparámetros que pudimos llevar a cabo e impidiendo el entrenamiento completo de algunos modelos. Por ejemplo, para entrenar de forma completa nuestro mejor modelo (con 3.000 neuronas por capa oculta, 4º fila de la Tabla 9), necesitamos 39 *epochs*, lo que se traduce en un total de aproximadamente 2.367.183 segundos, es decir, 27,4 días.

6.2. Resultados de otros modelos de aprendizaje automático

A continuación, se presentan los resultados obtenidos por los 25 modelos de aprendizaje automático del benchmark Lazy Predict en conjuntos de datos de diferente tamaño. Las tablas 10 y 11 muestran la exactitud, valor-F y tiempo de entrenamiento en subconjuntos de 700 y 7.000 instancias respectivamente. Luego, la tabla 12 destaca los resultados de los 5 mejores modelos de la tabla anterior, pero esta vez entrenados con 70.000 instancias. Por último, la tabla 13 muestra la información de los mismos 5 modelos sobre el mayor número de instancias posible, que es de 3.500.000. Estas tablas proporcionan una comparativa del rendimiento de los modelos a medida que aumenta el tamaño del conjunto de datos de entrenamiento, brindando información valiosa para evaluar su eficiencia.

La siguiente tabla muestra la exactitud, el valor-F y el tiempo de entrenamiento de los distintos modelos de aprendizaje sobre un subconjunto de datos de 700 instancias.

Modelo	Exactitud	Valor-F	Tiempo (seg.)
LinearSVC	0,31	0,33	88,69
PassiveAggressiveClassifier	0,31	0,33	3,43
GaussianNB	0,20	0,21	0,23
ExtraTreesClassifier	0,20	0,19	1,44
Perceptron	0,19	0,20	0,90
RidgeClassifier	0,19	0,19	0,31
RidgeClassifierCV	0,19	0,19	1,63
SGDClassifier	0,19	0,18	0,98
LogisticRegression	0,19	0,17	3,23
DecisionTreeClassifier	0,17	0,19	0,56
LinearDiscriminantAnalysis	0,16	0,14	7,38
XGBClassifier	0,16	0,13	14,70
RandomForestClassifier	0,16	0,11	1,02
LGBMClassifier	0,14	0,17	2,26
NearestCentroid	0,14	0,11	0,17
BaggingClassifier	0,13	0,11	1,98
AdaBoostClassifier	0,11	0,19	1,85
ExtraTreeClassifier	0,10	0,11	0,17
KNeighborsClassifier	0,06	0,04	0,26
BernoulliNB	0,06	0,02	0,17
CalibratedClassifierCV	0,04	0,02	279,91
SVC	0,04	0,01	1,60
LabelSpreading	0,03	0,00	0,24
LabelPropagation	0,03	0,00	0,23
QuadraticDiscriminantAnalysis	0,01	0,01	2,85

Tabla 10. Estadísticas acerca del rendimiento de los modelos del benchmark Lazy Predict entrenados con 700 instancias.

A continuación, se observa la tabla donde se muestra la exactitud, el valor-F y el tiempo de los distintos modelos del *benchmark* sobre un subconjunto de datos de 7.000 instancias.

Modelo	Exactitud	Valor-F	Tiempo (seg.)
XGBClassifier	0,52	0,56	456,68
LGBMClassifier	0,51	0,56	272,79
ExtraTreesClassifier	0,49	0,51	17,59
BaggingClassifier	0,47	0,51	55,73
RandomForestClassifier	0,45	0,48	12,95
LogisticRegression	0,44	0,48	20,28
SGDClassifier	0,43	0,46	175,24
DecisionTreeClassifier	0,42	0,48	10,70
CalibratedClassifierCV	0,41	0,41	3,038,65
PassiveAggressiveClassifier	0,40	0,46	45,26
RidgeClassifierCV	0,40	0,45	71,64
RidgeClassifier	0,40	0,45	2,96
LinearDiscriminantAnalysis	0,40	0,45	67,01
LinearSVC	0,37	0,43	664,27
NearestCentroid	0,36	0,41	1,30
Perceptron	0,36	0,40	24,71
SVC	0,35	0,35	290,95
ExtraTreeClassifier	0,31	0,36	1,61
BernoulliNB	0,31	0,35	1,51
AdaBoostClassifier	0,28	0,27	23,50
GaussianNB	0,26	0,30	2,08
KNeighborsClassifier	0,18	0,20	2,13
LabelSpreading	0,10	0,05	4,02
LabelPropagation	0,10	0,05	4,11
QuadraticDiscriminantAnalysis	0,04	0,05	14,11

Tabla 11. Estadísticas acerca del rendimiento de los modelos del benchmark Lazy Predict entrenados con 7.000 instancias.

En este punto podemos observar que, entre los cinco algoritmos con mayor exactitud, cuatro de ellos están basados en árboles, mientras que BaggingClassifier se basa en la combinación de varios modelos base. Esto puede deberse a que los algoritmos basados en árboles se benefician más de las variables nominales en formato *one-hot,* características de nuestro problema de clasificación, frente a otros algoritmos, como por ejemplo los basados en regresión o distancias.

En la siguiente tabla, se muestra el rendimiento obtenido por los 5 mejores modelos de la tabla anterior, entrenados con 70.000 instancias. En ella se puede apreciar que, con datos suficientes, los cuatro mejores modelos son los basados en árboles, dejando en quinta posición a BaggingClassifier.

Modelo	Exactitud	Valor-F	Tiempo (seg.)
LGBMClassifier	0,70	0,73	423,69
XGBClassifier	0,68	0,71	1.284,22
ExtraTreesClassifier	0,68	0,71	443,50
RandomForestClassifier	0,67	0,70	271,39
BaggingClassifier	0,66	0,71	1.913,20

Tabla 12. Estadísticas acerca del rendimiento de los modelos del benchmark Lazy Predict entrenados con 70.000 instancias.

Seguidamente se muestra una tabla con el rendimiento obtenido por los cinco mismos modelos de la tabla anterior sobre el mayor número de instancias posibles: 3.500.000.

Modelo	Exactitud	Valor-F	Tiempo (seg.)
ExtraTreesClassifier	0,83	0,82	113.002,36
RandomForestClassifier	0,82	0,81	64.993,63
XGBClassifier	0,71	0,70	9.160,61
LGBMClassifier	*Out of memory*		
BaggingClassifier	*Out of memory*		

Tabla 13. Estadísticas acerca del rendimiento de los modelos del benchmark Lazy Predict entrenados con 3.500.000 instancias.

Finalmente, fijándonos en las cuatro tablas presentadas, podemos afirmar que el volumen de datos de entrenamiento es una variable crucial para obtener buen rendimiento para este problema de clasificación. Conforme aumentamos el número de instancias, el rendimiento de la clasificación aumenta, para todos los algoritmos probados. Tomando como referencia el de mayor rendimiento, ExtraTreesClassifier, la exactitud aumenta de un 49% para 7.000 instancias a un 70% para 70.000, llegando a un 83% cuando se entrena con 3,5 millones de individuos.

Vemos, pues, cómo el modelo basado en redes neuronales propuesto es el que mejor rendimiento consigue, permitiendo aunar la potencia de un aprendizaje profundo con un número elevado de parámetros con el rendimiento de solo cargar en memoria el número de elementos de cada *batch*.

6.3. Comparación con otros sistemas

La Figura 22 muestra una comparación entre la exactitud obtenida por nuestro mejor modelo de redes neuronales (con 3.000 neuronas por cada capa, función de activación SELU, y sin *dropout*), la de los 3 mejores modelos de aprendizaje del *benchmark* utilizado y el modelo del trabajo relacionado seleccionado, [Guesslang2023]. Como se puede observar, nuestro modelo de red neuronal obtiene la mejor exactitud con un 92,18%. Seguidamente se sitúan los modelos del *benchmark* ExtraTreesClassifier, RandomForestClassifier y XGBClassifier, los cuales logran una exactitud que varía entre el 71% y el 83%. Por último, Guesslang muestra la menor exactitud de todos los modelos, con solo un 18,3%.

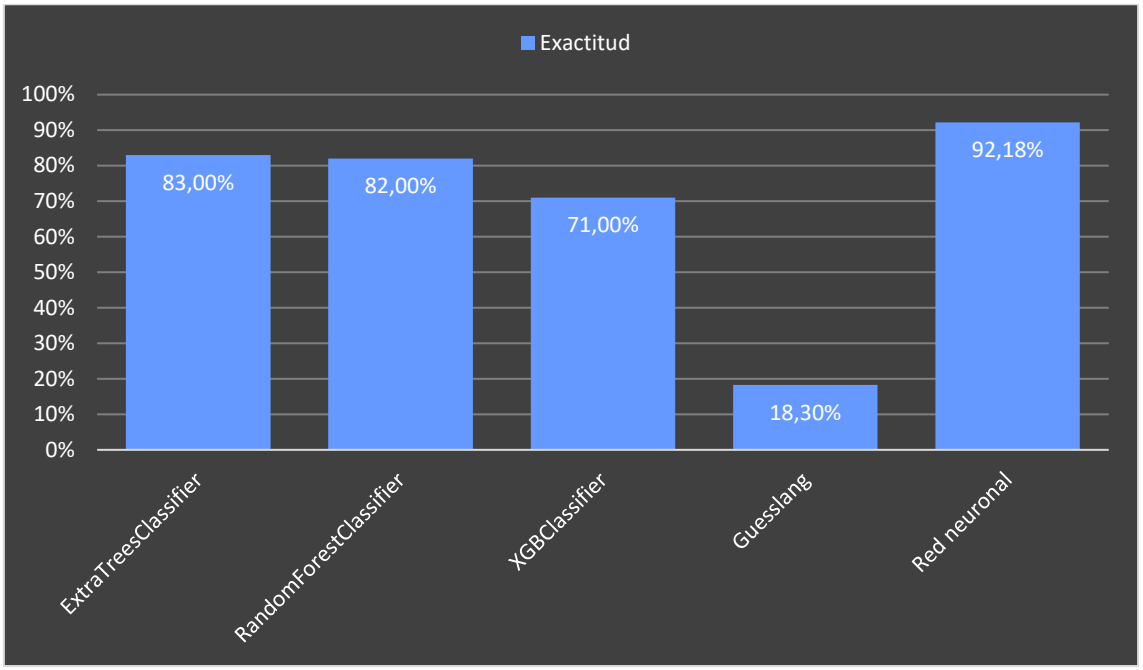

Figura 22. Comparativa de la exactitud obtenida por los tres mejores modelos de Lazy Predict, el mejor modelo del trabajo relacionado y el mejor modelo de red neuronal.

Esta diferencia de exactitud entre Guesslang y el resto de los modelos se debe principalmente al conjunto de entrenamiento. Mientras que los modelos del *benchmark* y nuestra red neuronal fueron entrenados utilizando líneas de código fuente de al menos 10 caracteres como instancias y un vocabulario a nivel de carácter, Guesslang fue entrenado con *snippets* como instancias y un vocabulario a nivel de palabra. Como resultado, Guesslang no está preparado para generalizar en base a una sola línea de código, y mucho menos, líneas de 10 caracteres, lo que afecta a su capacidad para realizar predicciones precisas en tales casos.

Con respecto a la exactitud obtenida por los tres modelos del *benchmark*, que se sitúa entre el 71% y el 83%, es importante recordar que estos modelos fueron entrenados con el mismo conjunto de datos que nuestra red neuronal. Sin embargo, debido a las limitaciones de memoria RAM y al diseño de estos modelos, que no permiten un entrenamiento *online*, y requieren codificación *one-hot* para variables nominales, sólo pudieron ser entrenados con 3,5 millones de instancias, lo que representa solo el 0,5% del tamaño del conjunto de datos utilizado para el entrenamiento de nuestra red neuronal. Esto subraya la importancia crucial de contar con un conjunto de datos de volumen y calidad en el entrenamiento de modelos de aprendizaje automático. El conjunto de datos proporciona la información necesaria para que el modelo pueda aprender y generalizar de manera efectiva. Por otro lado, la diferencia de más de un 10% de rendimiento entre XGBClassifier y los otros dos mejores modelos del *benchmark*, probablemente se deba a que XGBClassifier requiere ajustar adecuadamente los hiperparámetros para obtener un mejor rendimiento.

Es importante mencionar que, aunque en la gráfica presentamos nuestro mejor modelo de red neuronal, los otros 7 modelos que tenemos disponibles también superan en exactitud a todos los modelos con los que nos comparamos. Incluso nuestro peor

modelo (que terminó de ser entrenado), a pesar de contar con sólo 194 neuronas por capa oculta, logra una exactitud del 86,88%, superando el 83% de ExtraTreesClassifier. Esto resalta el buen rendimiento general de nuestros modelos de red neuronal en comparación con los modelos de la comparación.

7. Conclusiones

En este proyecto abordamos el problema de la identificación del lenguaje de programación de líneas de código, con al menos 10 caracteres y longitud máxima de 40. Hemos visto cómo una red neuronal profunda con 22,8 millones de parámetros es el modelo más apropiado para este trabajo. Dicho modelo ha sido capaz de clasificar las líneas de código fuente con un 92,18% de exactitud de entre 21 lenguajes, valor significativamente superior al resto de sistemas evaluados.

Para ello, mediante el desarrollo de un *crawler*, se ha confeccionado un corpus seleccionando ficheros de hasta 21 lenguajes de programación distintos y ampliamente utilizados según rankings de confianza. Este corpus se ha verificado satisfactoriamente con una metodología semi-automática, determinando que para 16 de los 21 lenguajes no se ha detectado ningún fichero ajeno, y para los 5 lenguajes restantes el porcentaje de ruido detectado era asumible. El corpus final contiene 1.474.584 ficheros y tiene un volumen de 19,8 GB.

A partir de este corpus, se ha generado un conjunto de datos de propósito específico para resolver el problema de identificación de código fuente para una única línea de código. Para ello, se ha construido un procesador de lenguajes capaz de limpiar las partes de los ficheros, correspondientes a distintas construcciones dependientes de cada lenguaje, que contienen lenguaje natural. Posteriormente, se han transformado estos ficheros, generando un conjunto de datos final compuesto de vectores numéricos de la misma longitud, que representan cada una de las líneas de código, y que sirve como entrada para los algoritmos de aprendizaje automático utilizados. El conjunto de datos final consta de 700M de instancias para el entrenamiento y 1M para la validación.

Con este conjunto de datos se han creado hasta 8 modelos de redes neuronales mediante la búsqueda y ajuste de hiperparámetros, alcanzando un máximo de exactitud del 92,18%. Esto demuestra hasta qué punto se puede identificar el lenguaje de programación de entre 21 candidatos, con tan sólo una línea de código fuente de 10 o más caracteres. Tras analizar los resultados de los experimentos realizados, determinamos que los modelos con mayor número de neuronas por capa, y por lo tanto mayor capacidad, exhiben siempre mayor eficiencia de clasificación que los modelos con menor capacidad. De hecho, el modelo con mayor número de neuronas por capa (3.000 neuronas) obtuvo la mayor exactitud de este trabajo con un 92,18%, un 0,16%, un 1,94% y un 5,29% más, que los siguientes modelos con más neuronas por capa (2.000, 1.000 y 194 respectivamente). Con respecto a las distintas funciones de activación analizadas, podemos afirmar que, Leaky ReLU, a pesar de no cumplir el criterio de parada, fue la mejor función de activación logrando un 90,4% de exactitud, un 0,38% por encima de SELU y un 0,16% más que ELU. En cuanto al *dropout,* las evidencias de los experimentos realizados sugieren que, a mayor tasa, más se resiente la eficiencia de clasificación de las redes neuronales, para nuestro problema. Esto es debido a que el elevado número de individuos utilizados ya se comporta con un buen mecanismo de regularización.

Se han analizado los errores cometidos por el mejor modelo, observando cómo los lenguajes más difíciles de diferenciar son lenguajes que extienden unos de otros como es el caso de C y C++, JavaScript y TypeScript o lenguajes que permiten la programación orientada a objetos y además comparten una sintaxis estilo C, como C++, C#, Java, JavaScript, Kotlin o Scala. Por otro lado, los lenguajes que mejor se han discriminado son lenguajes con una sintaxis concreta o un propósito especifico, como Assembly, CSS, Matlab, Python, SQL o Unix Shell, para los que consigue un porcentaje de acierto de entre 96% y 99%.

Además de evaluar nuestros propios modelos de redes neuronales, empleamos la herramienta Lazy Predict 0.2.12 para evaluar diferentes modelos de aprendizaje automático utilizando nuestro conjunto de datos y sin necesidad de ajustar manualmente los hiperparámetros. Esta evaluación nos permitió determinar que los modelos más efectivos para nuestro problema de clasificación son los basados en árboles (ExtraTreesClassifier, RandomForestClassifier y XGBClassifier), que consiguen una exactitud de entre el 71% y el 83%. La limitación que presentan estos modelos es la de no poder entrenar con tantos datos como las redes neuronales, al no poder cagar todo el conjunto de datos en memoria.

Finalmente, se llevó a cabo un análisis de los trabajos relacionados para seleccionar aquellos con mayor relación con el nuestro y llevar a cabo una comparación. A causa de la dificultad para encontrar trabajos relacionados centrados en la clasificación de líneas de código fuente, se seleccionaron varios sistemas centrados en la clasificación de *snippets*. De entre ellos, por limitación de tiempo, finalmente se seleccionó el de mayor rendimiento, Guesslang 2.2.1 el cual reporta una exactitud del 93,4% en su documentación. Sin embargo, tras evaluar este trabajo con nuestro conjunto de datos compuesto por líneas de código de 10 o más caracteres, la exactitud de este sistema cayó a un 18,3%. Esto pone de manifiesto la incapacidad de los sistemas actuales a la hora de clasificar *snippets* realmente cortos.

Para esta comparación también incluimos los tres mejores modelos de aprendizaje automático del *benchmark* (ExtraTreesClassifier, RandomForestClassifier y XGBClassifier), así como nuestro mejor modelo basado en redes neuronales. Estos resultados indican que nuestro modelo de red neuronal es el de mayor eficiencia, con un 92,18% de exactitud. Nuestro modelo obtiene una mejora de entre 9,18% y 21,18% de exactitud con respecto a los modelos del *benchmark,* y un aplastante 73,88% de diferencia comparado con Guesslang. Esto pone de manifiesto la importancia de los datos para poder resolver el problema de clasificación propuesto en este trabajo con mayor eficiencia. El modelo entrenado a nivel de *snippet* y basado en palabras no está preparado para la clasificación de líneas de código, presentando una diferencia de más del 50% de exactitud con respecto al peor modelo entrenado con nuestro conjunto de datos. Mientras que los modelos de aprendizaje entrenados con sólo 3,5 millones de instancias y sin ajuste de hiperparámetros ofrecen un rendimiento inferior al de las redes neuronales, entrenadas con 700 millones de ejemplos.

8. Trabajo futuro

A continuación, se muestra una breve descripción de los distintos trabajos futuros que planeamos afrontar, tras la realización del presente proyecto:

- Implementar modelos de redes neuronales más eficientes en la utilización de parámetros para secuencias de entrada, como modelos de lenguaje basados en redes recurrentes LTSM [Hochreiter1997] o modelos neuronales basados en *transformers* [Vaswani2017]. De esta forma, buscamos superar el rendimiento de nuestro mejor modelo conseguido, que sigue una arquitectura clásica basada en el perceptrón multicapa.

- Completar el trabajo añadiendo la clasificación de los 4 lenguajes de programación que nos vimos obligados a excluir por escasez de datos, CMake, Visual Basic .Net, VBA y Objective-C. Lenguajes para los que ya tenemos preparada la infraestructura y de los que actualmente disponemos de suficientes datos, ya que durante la elaboración de este trabajo el *crawler* descargó nuevos datos.

- Ampliar la selección de lenguajes de programación de los distintos rankings, para poder clasificar otros lenguajes popularmente utilizados.

- Mejorar el proceso de verificación automática del corpus de ficheros, utilizando sistemas de clasificación de código fuente a nivel de fichero existentes y descritos en la sección de trabajo relacionado como [Gilda2017] y [VanDam2016], que proporcionan precisiones cercanas al 100%. El objetivo de esto es poder omitir el proceso de verificación manual, que resulta muy costoso.

- Eliminar construcciones no contempladas más complejas que contienen lenguaje natural como los HEREDOCs de PHP, Perl o Ruby, los NOWDOCs de PHP, o los *percent strings* de Ruby o Perl.

- Terminar de entrenar los modelos inacabados por limitaciones de tiempo, como los de las funciones de activación y los de *dropout* 0,1 y 0,2.

- Experimentar con las funciones de activación ELU y Leaky ReLU en modelos con 3.000 neuronas por capa, para comprobar si podemos superar el rendimiento de nuestro mejor modelo conseguido.

- Experimentar con la variación de más hiperparámetros como la tasa de aprendizaje, la dimensión de los *embeddings,* el número de capas ocultas de la red o el tipo de optimizador.

- Establecer el vocabulario cerrado del modelo en función de la frecuencia de aparición de los caracteres en una muestra del corpus de ficheros, en lugar de usar

un rango fijo de caracteres seleccionado por un experto. Esta mejora ya está implementada, aunque por limitaciones de tiempo no pudo ser evaluada.

- Los modelos de clasificación construidos se podrían integrar en aplicaciones reales ya existentes como editores de texto, motores de búsqueda, IDEs, plataformas de pregunta y respuesta como StackOverflow, y repositorios de código como Github o BitBucket. De esta forma, dotaríamos a estas aplicaciones de un mecanismo de identificación automática de código fuente, capaz de obtener muy buen rendimiento para *snnipets* muy pequeños, de tan sólo una línea de código fuente.

- También se pueden desarrollar nuevas herramientas software particulares para la clasificación de código fuente, basadas en los modelos construidos.

- Analizar la precisión de los modelos con números cambiantes de caracteres mínimos y máximos, para poder discutir los datos en base a cuál es el número necesario de caracteres para identificar el lenguaje de programación a partir de una línea de éste.

- Publicar los resultados de nuestra investigación, así como el corpus, *dataset* y modelos construidos.

9. Planificación y presupuesto

9.1. Planificación

La siguiente tabla pone de manifiesto las horas de trabajo junto con sus fechas de inicio y final. Durante la fase de investigación, el número de horas diarias fue de 2. Para la fase de desarrollo se invirtió una media de unas 4 horas por día de trabajo. Mientras que, para la fase de evaluación, el número de horas diarias dedicadas al proyecto subió a 8. Finalmente, para la documentación, la media de horas invertidas por día de trabajo fue de 4.

Tarea	Duración (horas)	Comienzo	Final
Investigación	**34**	**20/07/2020**	**11/08/2020**
Selección y análisis de lenguajes de programación	24	20/07/2020	04/08/2020
Estudio del trabajo relacionado	10	5/08/2020	11/08/2020
Desarrollo	**250**	**14/09/2020**	**14/09/2022**
Diseño e implementación del *crawler*	45	14/09/2020	21/09/2020
Diseño e implementación del verificador automático	53	23/09/2020	12/10/2020
Implementación del procesador de lenguajes	79	17/10/2020	9/12/2020
Diseño e implementación del generador de *dataset*	42	01/08/2022	18/08/2022
Diseño e implementación del selector de hiperparámetros	31	05/09/2022	14/09/2022
Evaluación	**50**	**16/09/2022**	**31/05/2023**
Análisis de modelos de red neuronal	26	16/09/2022	10/05/2023
Análisis del *benchmark*	14	29/05/2023	30/05/2023
Análisis de trabajo relacionado	10	30/05/2023	31/05/2023
Documentación	**220**	**20/01/2023**	**05/07/2023**
Total	554	20/07/2020	05/07/2023

Tabla 14. Planificación del proyecto.

9.2. Presupuesto

9.2.1. Coste laboral unitario

Todas las tareas que se muestran en la tabla 14 se han llevado a cabo siguiendo un flujo de trabajo secuencial, en el cual, cada tarea ha sido realizada por una persona utilizando su propio ordenador personal. Esta persona ha desempeñado diferentes roles, como investigador y desarrollador de software, según las tareas que se abordaban.

En las Tablas 15, 16 y 17 se muestran los salarios para las tareas en las que participaron un investigador, un desarrollador de software y un analista de datos, respectivamente.

Estos precios incluyen impuestos, aunque no incluyen recursos adicionales, ya que otros costes, como el consumo de electricidad o el uso de Internet, se consideran costes indirectos, y los trabajadores proporcionaron sus propios ordenadores para llevar a cabo el trabajo. Para obtener más detalles sobre los costes indirectos, consulta la sección Presupuesto total.

Unidad	Rol	Precio/hora (€)	Horas	Subtotal (€)
Hora	Investigador	30,00 €	1	30,00 €
		Total precio/hora		30,00 €

Tabla 15. Coste laboral unitario para el investigador.

Unidad	Rol	Precio/hora (€)	Horas	Subtotal (€)
Hora	Desarrollador de software	22,00 €	1	22,00 €
		Total precio/hora		22,00 €

Tabla 16. Coste laboral unitario para el desarrollador de software.

Unidad	Rol	Precio/hora (€)	Horas	Subtotal (€)
Hora	Analista de datos	35,00 €	1	35,00 €
		Total precio/hora		35,00 €

Tabla 17. Coste laboral unitario para el analista de datos.

9.2.2. Salario por unidad de obra

En las siguientes tablas se presenta el precio de cada tarea realizada en el proyecto, teniendo en cuenta el número de trabajadores asignados y las horas de trabajo correspondientes.

Se puede observar que el rol de desarrollador de software se encarga de las tareas que implican la construcción e implementación de soluciones de software, como el desarrollo de las herramientas de verificación automática o el procesador de lenguajes.

El rol de investigador se encarga de seleccionar y analizar los lenguajes de programación sobre los que se realiza la clasificación, estudiar los trabajos ya existentes y comunicarse con el desarrollador de software para analizar las necesidades de los sistemas. Además, esta persona es la encargada de elaborar el estado del arte.

Por otro lado, el rol del analista de datos se encarga de desarrollar los sistemas necesarios para convertir el corpus en un *dataset* y para crear y entrenar los modelos de aprendizaje automático. Además, esta persona es la responsable de evaluar y redactar los resultados obtenidos por los Alquiler de equipo

La siguiente tabla muestra el salario del investigador por realizar la tarea del Selección y análisis de lenguajes de programación.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
24	Investigador	30,00 €	720,00 €
		Precio total	720,00 €

Tabla 18. Coste de la selección y el análisis de lenguajes de programación.

La siguiente tabla muestra el salario del investigador por el estudio de los trabajos relacionados.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
10	Investigador	30,00 €	300,00 €
		Precio total	300,00 €

Tabla 19. Coste del estudio del trabajo relacionado.

La siguiente tabla muestra el salario del desarrollador de software por diseñar e implementar el rastreador (*crawler*).

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
45	Desarrollador de software	22,00 €	990,00 €
		Precio total	990,00 €

Tabla 20. Coste del diseño e implementación del crawler.

La siguiente tabla muestra el salario del desarrollador de software por diseñar e implementar el verificador automático.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
53	Desarrollador de software	22,00 €	1.166,00 €
		Precio total	1.166,00 €

Tabla 21. Coste del diseño e implementación del verificador automático.

La siguiente tabla muestra el salario del desarrollador de software por implementar el procesador de lenguajes.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
79	Desarrollador de software	22,00 €	1.738,00 €
		Precio total	1.738,00 €

Tabla 22. Coste de la implementación del procesador de lenguajes.

La siguiente tabla muestra el salario del analista de datos por diseñar e implementar el generador del *dataset*.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
42	Analista de datos	35,00 €	1.470,00 €
		Precio total	1.470,00 €

Tabla 23. Coste del diseño e implementación del generador de dataset.

La siguiente tabla muestra el salario del analista de datos por diseñar e implementar el subsistema de selección de hiperparámetros.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
31	Analista de datos	35,00 €	1.085,00 €
		Precio total	1.085,00 €

Tabla 24. Coste del diseño e implementación del subsistema de selección de hiperparámetros.

La siguiente tabla muestra el salario del analista de datos por evaluar los resultados de los modelos de red neuronal.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
26	Analista de datos	35,00 €	910,00 €
		Precio total	910,00 €

Tabla 25. Coste de la evaluación de los resultados de los modelos de red neuronal.

La siguiente tabla muestra el salario del analista de datos por evaluar los resultados obtenidos del *benchmark* Lazy Predict.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
14	Analista de datos	35,00 €	490,00 €
		Precio total	490,00 €

Tabla 26. Coste de la evaluación de los resultados del benchmark.

La siguiente tabla muestra el salario del analista de datos por evaluar los resultados obtenidos del mejor trabajo relacionado, Guesslang.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
10	Analista de datos	35,00 €	350,00 €
		Precio total	350,00 €

Tabla 27. Coste de la evaluación de los resultados de Guesslang.

La siguiente tabla muestra el salario del investigador por redactar la introducción, la fijación de objetivos y el trabajo relacionado de este documento. Por su parte, el desarrollador de software fue el encargado de redactar la descripción del crawler, de la verificación automática y del procesador de lenguajes. Finalmente, el analista de datos redacto la metodología llevada a cabo, los resultados obtenidos, las conclusiones, el trabajo futuro y el resumen del documento.

Medida (horas)	Rol	Precio/hora (€)	Subtotal (€)
72	Investigador	30,00 €	2.160,00 €
40	Desarrollador de software	22,00 €	880,00 €
108	Analista de datos	35,00 €	3.780,00 €
		Precio total	6.820,00 €

Tabla 28. Coste de la redacción del documento.

9.2.3. Presupuesto total

La Tabla 29 exhibe el presupuesto definitivo del proyecto, detallando los precios de todas las tareas mencionadas previamente. Es notable que se ha aplicado un incremento del 10 % en concepto de costos indirectos con el propósito de cubrir otros

gastos relacionados con el proyecto. La Tabla 30 contiene el listado de los ítems adicionales contemplados en el costo.

Unidad	Tarea	Coste (€)
1	Selección y análisis de lenguajes de programación	720,00 €
1	Estudio del trabajo relacionado	300,00 €
1	Diseño e implementación del *crawler*	990,00 €
1	Diseño e implementación del verificador automático	1.166,00 €
1	Implementación del procesador de lenguajes	1.738,00 €
1	Diseño e implementación del generador de *dataset*	1.470,00 €
1	Diseño e implementación del selector de hiperparámetros	1.085,00 €
1	Análisis de modelos de red neuronal	910,00 €
1	Análisis del *benchmark*	490,00 €
1	Análisis de trabajo relacionado	350,00 €
1	Documentación	6.820,00 €
	Coste total	16.039,00 €
	Coste indirecto (10%)	1.603,90 €
	Presupuesto final (Coste total + coste indirecto)	17.642,90 €

Tabla 29. Presupuesto final del trabajo.

Partida de gastos
Electricidad
Internet
Alquiler de máquina virtual
Comunicación (voz + datos)
Dietas
Transporte
Limpieza y mantenimiento
Suministros de oficina
Primas de seguros

Tabla 30. Costes indirectos.

10. Referencias

[Aggarwal2015] Aggarwal, K., Salameh, M., & Hindle, A. (2015). Using machine translation for converting python 2 to python 3 code (No. e1817). PeerJ PrePrints.

[Alreshedy2018] Alreshedy, K., Dharmaretnam, D., German, D. M., Srinivasan, V., & Gulliver, T. A. (2018). SCC: automatic classification of code snippets. arXiv preprint arXiv:1809.07945.

[Alrashedy2020] Alrashedy, K., Dharmaretnam, D., German, D. M., Srinivasan, V., & Gulliver, T. A. (2020). Scc++: Predicting the programming language of questions and snippets of stack overflow. Journal of Systems and Software, 162, 110505.

[ANTLR2023a] Download ANTLR, v4.10.1. https://www.antlr.org/download.html

[ANTLR2023b] Grammars written for ANTLR v4; expectation that the grammars are free of actions. https://github.com/antlr/grammars-v4

[Barone2017] Barone, A. V. M., & Sennrich, R. (2017). A parallel corpus of python functions and documentation strings for automated code documentation and code generation. arXiv preprint arXiv:1707.02275.

[Bhatia2016] Bhatia, S., & Singh, R. (2016). Automated correction for syntax errors in programming assignments using recurrent neural networks. arXiv preprint arXiv:1603.06129.

[Bhoopchand2016] Bhoopchand, A., Rocktäschel, T., Barr, E., & Riedel, S. (2016). Learning python code suggestion with a sparse pointer network. arXiv preprint arXiv:1611.08307.

[BigQuery2023] BigQuery: un almacén de datos empresariales | Google Cloud. https://cloud.google.com/bigquery

[Chomsky1959] Chomsky, N. (1959). On certain formal properties of grammars. Information and control, 2(2), 137-167.

[Clevert2015] Clevert, D. A., Unterthiner, T., & Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv:1511.07289.

[Géron2022] Géron, A. (2022). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems (3rd ed.). O'Reilly Media.

[Gilda2017] Gilda, S. (2017, July). Source code classification using Neural Networks. In 2017 14th international joint conference on computer science and software engineering (JCSSE) (pp. 1-6). IEEE.

[GitHub2023a] Copilot, your AI pair programmer. https://github.com/features/copilot.

[GitHub2023b]. One million repositories. https://github.blog/ 2010-07-25-one-million-repositories.

[GitHub2023c]. Where the world builds software. https://github.com/about.

[Guesslang2023] Guesslang 2.2.2 documentation – Read the docs. https://guesslang.readthedocs.io/en/latest/

[Haykin1999] Haykin, S. (1999). Neural Networks: A Comprehensive Foundation. Prentice Hall.

[He2015] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE international conference on computer vision (pp. 1026-1034).

[Hochreiter1997] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

[Hong2019] Hong, J., Mizuno, O., & Kondo, M. (2019, December). An empirical study of source code detection using image classification. In 2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP) (pp. 1-15). IEEE.

[Igl2019] Igl, M., Ciosek, K., Li, Y., Tschiatschek, S., Zhang, C., Devlin, S., & Hofmann, K. (2019). Generalization in reinforcement learning with selective noise injection and information bottleneck. Advances in neural information processing systems, 32.

[Keras2023] Keras: Deep Learning for humans, v2.9.0. https://keras.io/

[Khasnabish2014] Khasnabish, J. N., Sodhi, M., Deshmukh, J., & Srinivasaraghavan, G. (2014). Detecting programming language from source code using bayesian learning techniques. In Machine Learning and Data Mining in Pattern Recognition: 10th International Conference, MLDM 2014, St. Petersburg, Russia, July 21-24, 2014. Proceedings 10 (pp. 513-522). Springer International Publishing.

[Kiyak2020] Kiyak, E. O., Cengiz, A. B., Birant, K. U., & Birant, D. (2020). Comparison of image-based and text-based source code classification using deep learning. SN Computer Science, 1(5), 266.

[Klambauer2017] Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017). Self-normalizing neural networks. Advances in neural information processing systems, 30.

[Kleene1956] Kleene, S. C. (1956). Finite Automata and Their Decision Problems. IBM Journal of Research and Development, 2(2), 75-89.

[LazyPredict2023] Welcome to Lazy Predict's documentation! — Lazy Predict 0.2.12. https://lazypredict.readthedocs.io/en/latest/

[LeCun1998] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[LeCun2002] LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (2002). Efficient backprop. In Neural networks: Tricks of the trade (pp. 9-50). Berlin, Heidelberg: Springer Berlin Heidelberg.

[Liu2019] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... & Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

[Maas2013] Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013, June). Rectifier nonlinearities improve neural network acoustic models. In Proc. icml (Vol. 30, No. 1, p. 3).

[Mikolov2013] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. Advances in neural information processing systems, 26.

[Nair2010] Nair, V., & Hinton, G. E. (2010). Rectifier nonlinearities improve neural network acoustic models. En IEEE International Conference on Acoustics, Speech, and Signal Processing (páginas 5686-5689).

[Nesterov1983] Nesterov, Y. E. E. (1983). A method of solving a convex programming problem with convergence rate O\bigl(k^2\bigr). In Doklady Akademii Nauk (Vol. 269, No. 3, pp. 543-547). Russian Academy of Sciences.

[NumPy2023] NumPy 1.19.5 Release Notes. https://numpy.org/devdocs/release/1.19.5-notes.html

[Ortin2016] Ortin, F., Escalada, J., & Rodriguez-Prieto, O. (2016). Big Code: New Opportunities for Improving Software Construction. J. Softw., 11(11), 1083-1088.

[Öztürk2023] Öztürk, M. M. (2023). Developing a hyperparameter optimization method for classification of code snippets and questions of stack overflow: HyperSCC. EAI Endorsed Transactions on Scalable Information Systems, 10(1), e5-e5.

[PyGithub2023] Introduction - PyGithub - Read the Docs https://pygithub.readthedocs.io/en/stable/changes.html#version-1-53-august-18-2020

[PYPL2023] PYPL PopularitY of Programming Language index for July 2020. https://pypl.github.io/PYPL.html

[RedMonk2023] The RedMonk Programming Language Rankings: July 2020. https://redmonk.com/sogrady/2022/10/20/language-rankings-6-22/

[Rumelhart1986] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

[SciKitLearn2023] Available documentation for Scikit-learn. https://scikit-learn.org/dev/versions.html

[Srivastava2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

[TensorFlow2023] TensorFlow API Versions | TensorFlow v2.12.0. https://www.tensorflow.org/versions?hl=es-419

[TIOBE2023] TIOBE Index for July 2020. https://www.tiobe.com/tiobe-index/

[VanDam2016] Van Dam, J. K., & Zaytsev, V. (2016, March). Software language identification with natural language classifiers. In 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER) (Vol. 1, pp. 624-628). IEEE.

[Vaswani2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In Advances in neural information processing systems (pp. 5998-6008).

[Yang2021] Yang, G., Zhou, Y., Yu, C., & Chen, X. (2021). DeepSCC: Source Code Classification Based on Fine-Tuned RoBERTa. arXiv preprint arXiv:2110.00914.