Future Generation Computer Systems 166 (2025) 107640

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

PLangRec: Deep-learning model to predict the programming language from a single line of code

Oscar Rodriguez-Prieto [a], Alejandro Pato [a], Francisco Ortin [a,b,*]

[a] *University of Oviedo, Computer Science Department, c/Leopoldo Calvo Sotelo 18, 33007, Oviedo, Spain*
[b] *Munster Technological University, Computer Science Department, Rossa Avenue, Bishopstown, Cork, Ireland*

A R T I C L E   I N F O

Keywords:
Deep learning
Massive source code datasets
Language recognition
Big data
Code snippet
PLangRec

A B S T R A C T

Programming language detection from source code excepts remains an active research field, which has already been addressed with machine learning and natural language processing. Identifying the language of short code snippets poses both benefits and challenges across various scenarios, such as embedded code analysis, forums, Q&A systems, search engines, source code repositories, and text editors. Existing approaches for language detection typically require multiple lines or even the entire file contents. In this article, we propose a character-level deep learning model designed to predict the programming language from a single line of code. To this aim, we construct a balanced dataset comprising 434.18 million instances across 21 languages, significantly exceeding the size of existing datasets by three orders of magnitude. Leveraging this dataset, we train a deep bidirectional recurrent neural network that achieves a 95.07% accuracy and macro-F_1 score for a single-line code. To predict the programming language of multiple lines (e.g., code snippets) and entire files, we build a stacking ensemble meta-model that leverages our single-line model to efficiently recognize the language of multiple lines of code. Our system outperforms the state-of-the-art approaches not only for a single line of code, but also for snippets of 5 and 10 lines and whole files of source code. We also present *PLangRec*, an open-source language detection system that includes our trained models. *PLangRec* is freely available as a user-friendly web application, a web API, and a Python desktop program.

1. Introduction

In the last decades, there has been a substantial growth in the availability of source code published in many ways [1]. Noteworthy among these are source code repositories such as GitHub, SourceForge, GitLab, BitBucket, and CodePlex, which represent databases of vast amounts of source code. For example, GitHub hosts at the beginning of 2024 more than 420 million repositories and boasts a community of 100 million developers [2]. Besides source code repositories, source code snippets are frequently encountered in other diverse digital sources, including forums, chats, wikis, online courses, and web pages covering many distinct topics.

Source code is written in a particular programming language. Programming languages follow different paradigms (e.g., object-oriented, functional, and logical), provide distinct features (e.g., explicit or implicit memory management, static or dynamic typing, and binary or managed by a virtual machine), and are often suitable for particular domains (e.g., web systems, scientific computation, and system programming) [3]. Thus, to discuss the quality, performance, security, or behavior of a source code excerpt, we must first know the programming language in which it has been written.

There are situations where detecting the programming language from a source code excerpt may be beneficial. Here are some common scenarios where this capability is useful:

– Text editors. Most text editors infer the programming language based on the file extension rather than the source code itself [4]. Consequently, when a new file is created, syntax highlighting, code indentation, and other formatting features are not applied until the file is saved with a specific file extension [5]. If an incorrect file extension is chosen upon saving, the code may be interpreted as being written in the wrong language. This issue is particularly significant for file extensions associated with multiple languages—for example, `.h` for C, C++, and Objective-C, and `.pl` for Perl and Prolog.
– Source code intermingled with natural language text. Numerous platforms predominantly feature natural language content

* Corresponding author at: University of Oviedo, Computer Science Department, c/Leopoldo Calvo Sotelo 18, 33007, Oviedo, Spain.
E-mail addresses: rodriguezoscar@uniovi.es (O. Rodriguez-Prieto), UO214630@uniovi.es (A. Pato), ortin@uniovi.es (F. Ortin).
URL: http://www.reflection.uniovi.es/ortin (F. Ortin).

https://doi.org/10.1016/j.future.2024.107640
Received 24 April 2024; Received in revised form 17 October 2024; Accepted 23 November 2024
Available online 30 November 2024

```
SELECT * FROM Customers WHERE Name = 'John';
```

```
int **p;
p = new int*[10];
```

Fig. 1. Code excerpts showing language detection from a single line of code (left) and an example where the first line is insufficient to determine the language (right).

(e.g., English, French, or Spanish text). However, it is not uncommon to find instances where source code is also included. Examples of such platforms are forums, chats, emails, Q&A systems such as Stack Overflow and Code Review, and code snippets tools such as gist and pastebin. In some of the previous examples, there is no standard way to represent the source code and its language. In others, systems rely on tags written by the users that are sometimes mis-tagged [6]. This leads to posts being downvoted and flagged by moderators, even though they may add value to the community. Furthermore, there are posts about language comparison (e.g., Python vs. R for machine learning) where two or more language tags are used. There are also language-agnostic questions where snippets in different languages are utilized in the responses.

– Embedded code. Source code sometimes embeds other code excerpts written in a different language. For example, SQL code to access databases is usually embedded into code in another language. Similarly, JavaScript and CSS code is commonly included in HTML. For these case scenarios, file extensions do not represent a reliable mechanism to know the programming language used in (each part of) a text file.
– Search engines. In cases where source code or its programming language cannot be accurately identified within natural language text, conventional search engines, including those on the Internet, may fail to yield optimal results. By implementing a mechanism to detect both code and language from text, search engines could provide more accurate responses to queries related to code written in a particular programming language. This detection technique would also be valuable for dedicated code search engines (e.g., SearchCode or Codebase), allowing them to move beyond reliance on file extensions and effectively determine the language of embedded code.
– Source code repositories. Metadata describing the programming language of source code repositories are often inaccurately tagged, primarily due to reliance on manual tagging and file extensions [4]. This issue is especially prevalent in projects that involve the use of multiple programming languages.

Despite the importance of accurately identifying the programming language, there are situations where discerning the language of a source code fragment proves challenging. Fig. 1 illustrates two contrasting examples. On the left, a line of SQL code is shown. A proficient language detection system could accurately identify the language from this single line. Conversely, the first line of the code snippet on the right is valid for both C and C++. However, by examining the second line, it becomes evident that the code is in C++ due to the presence of the `new` operator, which is not supported in C. Additionally, even the SQL code on the left could be part of a larger program written in another language, such as Python, where the SQL statement would be embedded to perform a database query. Thus, while it is important for a system to accurately detect the language from a single line of code, it must also consider the broader context provided by multiple lines to enhance its prediction accuracy.

To deduce the programming language of a source code excerpt, the research community has already leveraged machine learning and natural language processing (NLP) techniques [5]. Some systems achieve remarkably high accuracies (up to 97.5%) by analyzing entire source files [7–9]. Unfortunately, the analysis of complete files does not solve the problems discussed in the previous scenarios. Thus, certain works reduce the size of their inputs to code snippets, resulting in a notable decrease in performance (ranging from 75% to 93% accuracy) [5,10–13]. Notably, all these approaches require various lines of source code to perform their predictions.

As mentioned, open-source code repositories provide massive databases of source code written in diverse programming languages. With such a vast amount of data, it becomes plausible to train deep-learning models with a high number of parameters. Such models could thereby enhance the performance of the existing systems aimed at detecting the language of source code. This improvement could also allow the reduction of the input size to a single line of source code. Aligned with this idea, this article presents the following contributions:

1. A 113.3 GB corpus of 8.5 million source files labeled with 21 distinct categories representing their respective programming languages. We develop a GitHub crawler to download these files. A language verification module validates the actual language of each code file. Subsequently, the files are processed to construct a perfectly balanced dataset containing 434.18 million instances for 21 languages. This dataset significantly surpasses the scale of the one curated by Yang et al. [11] (228,000 instances for 19 languages), widely used in many research studies (see Section 2).
2. A character-level deep-learning model designed to predict the programming language of a single line of code from a pool of 21 different languages, achieving a 95.071% macro-F_1 score and 95.069% accuracy. We detail the methodology used to create the model, present its evaluation, and compare it to state-of-the-art systems.
3. A multi-line meta-model that leverages our single-line prediction model to identify the programming language from multiple lines of code. This meta-model is created using a stacking ensemble approach, incorporating a frequency distribution analysis of the languages predicted for each line by our single-line model. Our meta-model surpasses existing systems in classifying 5- and 10-line code snippets as well as entire source code files.
4. *PLangRec* (Programming Language Recognizer): an open-source web application, web API, and Python desktop software that uses our model and meta-model. With *PLangRec*, users can take any source code and know the programming language in which it has been written. The model is also available for download, ready to be integrated into any application.

The rest of the paper is structured as follows. The next section outlines the related work, while Section 3 elaborates on the construction of the dataset (the first contribution of the article). The methodology used to create the deep learning models (second and third contributions) is depicted in Section 4. Subsequently, in Section 5, we evaluate our system in comparison with the related work. Section 6 introduces *PLangRec* (fourth contribution), the web application, web API, and Python desktop software that integrates our meta-model and the top-performing single-line model. Conclusions and future work are presented in Section 7.

2. Related work

Various research projects are focused on predicting the programming language of source code excerpts. None of these methods facilitate language prediction with a single line of code. Instead, they typically deal with code snippets, which consist of a few consecutive lines of code. One such method is Source Code Classification (SCC), which employs a multinomial naïve Bayes classifier to identify the programming language of code snippets across 21 different languages [10].

Their model is constructed upon a bag-of-words word-level classifier, which overlooks the order of word occurrence in the code. To train the dataset, Alreshedy et al. downloaded 237,787 snippets from Stack Overflow. Their dataset comprises 12,000 snippets for 19 languages, plus 8428 snippets for Lua and 1359 snippets for Markdown, resulting in an imbalanced dataset. Snippets must contain at least two lines of code. They build a multinomial naïve Bayes classifier with an accuracy of 75%.

DeepSCC is an LLM-based (Large Language Model) system designed to classify the programming language from source code snippets of at least two lines of code [11]. DeepSCC utilizes RoBERTa (Robustly optimized BERT approach), a transformer-based deep learning model for natural language processing (NLP) tasks [14] built upon the architecture of Google's BERT (Bidirectional Encoder Representations from Transformers) [15]. Yang et al. fine-tune the pre-trained word-level RoBERTa model for language classification. The LLM encoder is fed into a linear layer with a Softmax activation function for multi-class classification. To mitigate the impact of the out-of-vocabulary (OOV) problem in word-label models, they employ a Byte-Pair Encoding (BPE) tokenizer [16]. They utilize the dataset created by Alreshedy et al. [10], excluding Lua and Markdown, to compose a balanced dataset. They achieve 87.2% accuracy for those 19 languages.

Guesslang is a Python tool for detecting the programming language of source code snippets [12]. Guesslang utilizes a deep neural network model combined with a linear classifier. The model's hyperparameters are fine-tuned, achieving a 93.45% accuracy for 54 programming languages. To train their model, they download 1.9 million source code files, randomly selected from 170,000 public GitHub repositories. The authors indicate that Guesslang encounters difficulty differentiating C from C++ and JavaScript from TypeScript [12]. It is also noted that Guesslang may not accurately identify the programming languages of very small code snippets.

EL-CodeBert [13] is a code snippet classifier based on the CodeBert pre-trained model, which is designed for both programming and natural languages [17]. EL-CodeBert leverages the representational information from each layer of CodeBert by treating them as a sequence of representational information. All the 12 elements of that sequence are passed to a bidirectional recurrent neural network (Bi-LSTM). Subsequently, an attention mechanism is employed to assign weights to each layer based on their significance. Finally, two fully connected networks are utilized for language classification. The model is trained on the balanced dataset created by Yang et al., consisting of 12,000 snippets for 19 programming languages [11]. With that dataset, EL-CodeBert achieves an accuracy of 86% and a macro F_1-score of 88.08%.

Other investigations focus on predicting the programming language from entire source code files, typically achieving higher performance compared to the approaches based on snippets. However, these approaches require the entire file content for prediction. Van Dam and Zaytsev utilize statistical language models from the NLP field, including *n*-grams, skip-grams, multinomial naïve Bayes, and normalized compression distance, achieving 97.5% accuracy in classifying 20 different languages [7].

Shlok Gilda employs a neural network with a word embedding layer, followed by a multi-layered convolutional network with multiple filters and max-pooling layers [8]. The model achieves a 97% accuracy in classifying 60 programming languages. Similarly, Khasnabish et al. perform language classification for entire source code files using naïve Bayes, Bayesian network, and multinomial naïve Bayes models, classifying 10 languages with 93.48% accuracy [18]. Odeh et al. also apply multinomial naïve Bayes models to classify 12 languages of entire code files with 95.09% accuracy [19].

While the previous systems train models exclusively with source code, others incorporate natural language alongside code input. Data is sourced from Q&A platforms like Stack Overflow and Code Review. Although the natural language text enriches the input providing better performance, the resulting models are specific for Q&A and related systems.

SCC++ (Source Code Classification) identifies the programming language of Stack Overflow questions by combining features from the title, body, and code snippets [5]. SCC++ achieves an accuracy of 88.9% for 21 programming languages, but the classifier's performance decreases to 78.9% when using only the title and body, and 78.1% with only the code snippet. The dataset used is the same as the one for SCC, described at the beginning of this section [10]—SCC++ is an improvement of SCC. Alrashedy et al. trained XGBoost and Random Forest models from the same dataset, obtaining slightly better results with XGBoost.

HyperSCC is another programming language classifier that combines code snippets and natural language [6]. XGBoost and Random Forest models are created, optimizing their hyperparameters with seven different automatic rule-based labeling approaches. The models are trained with a dataset comprising Stack Overflow posts, obtaining an accuracy of 85.5% for 24 programming languages.

The last kind of source code classification system is based on leveraging the advances of machine learning for image classification instead of applying NLP approaches. A good example of this method is the study of Kiyak et al., where they compare the performance of deep learning algorithms for code classification on both image and text files [9]. They create a dataset from GitHub with 40,000 source code files and images for eight different languages. For text files, they build a 1D convolutional neural network (CNN) with a filter of size three, ending with a softmax output over eight categories. For images, a similar approach with 2D CNN is followed. The comparative results indicate that the image-based classifier show slightly better performance than the text-based model (99.38% accuracy vs. 98.81%).

Hong, Mizuno, and Kondo propose a classifier based on the existing ResNet, AlexNet, and SimpleNet convolutional neural networks [20]. They create two datasets by generating images from source code: one with code snippets taken from Stack Overflow in ten different languages; and another with functions from open-source GitHub repositories in five programming languages. In their evaluation, ResNet is the CNN model with better performance, obtaining 92% accuracy for code snippets (identifying 10 languages) and 99% accuracy for functions (5 languages).

3. Dataset construction

Deep-learning models involve learning numerous parameters from data, necessitating a substantial amount of data to effectively train those many parameters. The first part of our system is focused on creating such a dataset (Fig. 2). Then, the dataset is used to train and test different models, as we detail in Section 4.

3.1. Language selection

Before constructing the dataset, we need to determine the list of programming languages from which we aim to create a source code database. To that aim, we consult the 15 more popular languages according to the Tiobe [21], PYPL [22], and RedMonk [23] programming language rankings. Additionally, we also utilize BigQuery to ask GitHub for the 15 languages with the highest number of open-source code repositories. The findings are presented in Table 1, illustrating the 25 most popular languages based on the aforementioned criteria. All these languages were considered for the construction of our dataset.

3.2. GitHub crawler

The construction of the dataset follows the steps outlined in Fig. 2. All the input files are taken from public open-source repositories in GitHub. We implement a Python crawler for GitHub using the PyGithub library 1.53 [24]. The crawler queries GitHub for repositories in one

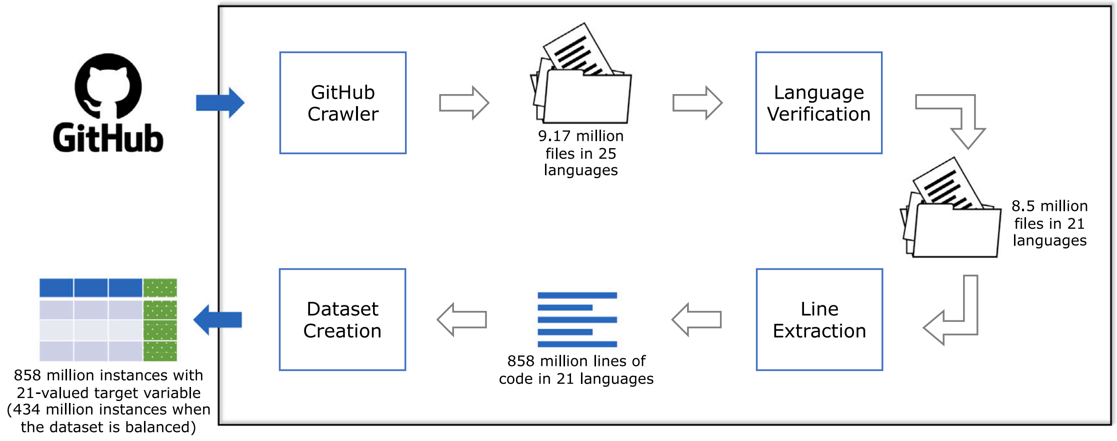

GitHub
GitHub Crawler
9.17 million files in 25 languages
Language Verification
8.5 million files in 21 languages
Line Extraction
858 million lines of code in 21 languages
Dataset Creation
858 million instances with 21-valued target variable (434 million instances when the dataset is balanced)

Fig. 2. Architecture of the system that creates the dataset to train our single-line model.

Table 1
The 15 most popular languages in the Tiobe, PYPL, and RedMonk rankings along with the 15 languages with the highest number of open-source repositories in GitHub.

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

Table 2
File extensions, number of files, and sizes of the source code files for the 25 programming languages selected.

Language	File extensions	Number of files	Size (MB)
Assembly	`.asm, .s, .S`	228,425	5,128.56
C	`.c`	295,052	10,419.27
C++	`.cpp, .cc, .cx`	242,096	5,133.16
C#	`.cs`	425,687	5,122.58
CSS	`.css`	137,399	5,121.30
Go	`.go`	375,558	8,648.69
HTML	`.html, .htm`	210,792	5,120.19
Java	`.java`	448,579	5,122.19
JavaScript	`.js`	185,192	5,124.80
Kotlin	`.kt, .kts, .ktm`	924,591	5,120.11
Makefile	`.cmake`	32,813	421.34
Matlab	`.m`	640,005	5,120.41
Objective-C	`.mm, .M`	27,334	352.92
Perl	`.pl`	368,872	5,120.23
PHP	`.php`	433,230	5,169.38
Python	`.py`	534,101	10,247.04
R	`.r`	396,190	5,120.17
Ruby	`.rb`	807,868	5,128.45
Scala	`.scala, .sc`	456,885	5,121.60
SQL	`.sql`	182,432	5,300.63
Swift	`.swift`	510,825	5,158.72
TypeScript	`.ts`	670,163	5,165.41
Unix shell	`.sh`	592,844	5,122.12
VBA	`.vba`	16,401	211.12
VB.Net	`.vb`	33,587	435.80
Total		9,176,921	123,256.19

of the 25 selected languages, receiving the repository URLs. Subsequently, the crawler analyzes these repositories, downloading files with the language extensions outlined in Table 2. We then verify that the actual language is the expected one (Section 3.3), rejecting all the files that have not been validated. Notably, the crawler is designed to seamlessly resume its execution from the point of interruption, ensuring continuity in the event of any errors (for further details on its design and implementation, please refer to [25]).

Our crawler operated for six months, downloading a total of 9.17 million files, and accumulating more than 123 GBs (see Table 2). After that process, four languages – Visual Basic for Applications (VBA) and .NET (VB.NET), Objective-C, and Makefile – had notably fewer files than the other languages. The four of them fall below 500 MBs, whereas the subsequent language with fewer files have 5.12 GBs. That would make it difficult to build a sufficiently large and balanced dataset. Consequently, we opted not to include these four languages in our dataset, focusing on the remaining 21 languages.

3.3. Language verification

The language verification module validates that a source file is actually written in the language associated with its file extension—otherwise, its content is not considered. For this purpose, we implement a Java application that, using context-free grammars specifying the selected languages, tells us whether the language used is the expected one [25]. We use the ANTLR 4.10.1 parser generation tool [26] and its collection of grammars for multiple programming languages [27].

Various challenges were addressed during the development of the language verifiers. First, for some languages such as C and C++ we ran a preprocessor prior to language recognition. Second, languages like SQL exhibit numerous variants. To handle this, we used the grammars for SQLite, MySQL, PostgreSQL, PL/SQL, and TSQL, requiring at least one positive recognition to interpret the code as a valid SQL file. Third, the grammars of Ruby and Matlab were obsolete, so we had to update them. Fourth, the grammar of Unix Shell was not available, so we had to implement it from scratch. Finally, issues arose with different versions of the same language, where we applied the same approach as for language variants (exemplified in the SQL scenario described earlier).

The following error analysis process was followed. We pass the language verifier to the source files. If the language is not the one derived from its file extension (Table 2), we analyze its cause. In cases where the code is indeed written in the expected language, we refine its grammar as outlined in the preceding paragraph. Otherwise, the verifier accurately rejects the source file. The refinement of the grammar for a particular language ceases when the last 100 analyzed

Table 3
Number of verified and accepted files. Languages with ψ are supported by all the systems evaluated (see Section 4.5).

Language	Number of files	Number of accepted files	Acceptance rate
Assembly	228,425	221,572	97.0%
C$^{\psi}$	295,052	286,200	97.0%
C++$^{\psi}$	242,096	222,244	91.8%
C#$^{\psi}$	425,687	387,375	91.0%
CSS$^{\psi}$	137,399	129,155	94.0%
Go	375,558	349,644	93.1%
HTML	210,792	194,139	92.1%
Java$^{\psi}$	448,579	400,132	89.2%
JavaScript$^{\psi}$	185,192	174,080	94.0%
Kotlin	924,591	878,361	95.0%
Matlab	640,005	582,404	91.0%
Perl$^{\psi}$	368,872	332,722	90.2%
PHP$^{\psi}$	433,230	421,099	97.2%
Python$^{\psi}$	534,101	502,054	94.0%
R$^{\psi}$	396,190	346,270	87.4%
Ruby$^{\psi}$	807,868	767,474	95.0%
Scala$^{\psi}$	456,885	452,316	99.0%
SQL$^{\psi}$	182,432	170,209	93.3%
Swift$^{\psi}$	510,825	485,283	95.0%
TypeScript	670,163	643,356	96.0%
Unix shell$^{\psi}$	592,844	567,351	95.7%
Total	9,066,786	8,513,440	93.9%

files are consistently verified correctly.

Table 3 presents the outcomes of the language verification process. In total, we verified 9 million files. The language verifiers detected 553,346 files (6.1%) as incorrectly classified by its language extension. The remaining files were included in constructing the dataset. The resultant corpus (the first part of the first contribution of this article) encompasses 113.3 GBs, comprising 8.5 million source files across 21 different languages.

3.4. Line extraction

Our system aims to predict the programming language from a single line of code. To this aim, the line extraction module extracts *valid* lines of code from the verified corpus of files. There are some lines of code that we do not consider valid to predict its language since they do not contain source code.

One scenario for discarding lines involves those containing natural language elements, such as comment lines and multiline strings. While certain comments, like Javadoc comments, might offer language-related information, most simply contain natural language text. Regarding strings, we exclude only multiline strings, allowing lines with simple strings. We also consider interpolated strings (e.g., f-strings in Python) since they usually include expressions in the corresponding language.

Additionally, we exclude leading and trailing white spaces and tabular characters (not those in the middle). These characters do not convey information about the programming language and hence are not considered in our analysis—while in Python leading space and tabular characters may provide information about scope with multiple lines, our predictions are based on individual lines. We also apply a criterion for valid lines based on a minimum character threshold. Lines with fewer than ten characters are discarded.

We implement the line extraction module as a native application developed in Go 1.18.1. We selected this language because of its runtime performance and strong support for concurrent programming, enabling parallel processing of numerous files. The line extraction process heavily depends on the programming language being processed, so we manually implemented all the patterns of characters to be discarded described above. Several notable cases arise, such as the presence of nested comments in Kotlin and Swift, or the nested string interpolation feature found in languages like C#. The language extraction application comprises 33 Go classes, detailed in [25].

3.5. Dataset creation

Utilizing the extracted lines as input, we create the dataset, categorizing each line with its respective programming language (an integer number from 0 to 20). The initial step in the dataset creation module involves transforming the source code files from UTF-16 (the file format used by the GitHub crawler in Section 3.2) to UTF-8. Subsequently, we conduct a character frequency analysis to identify the most frequently used characters in the source code files. Following a closed-vocabulary NLP approach [28], we employ a fixed number of characters, designating those rarely used as a special out-of-vocabulary (OOV) character.

Fig. 3 depicts a histogram illustrating the distribution of characters across all the verified lines. Characters within the ASCII code range of 32 to 125 plus tabular (code 9) constitute 99.97% of the occurrences. Table 4 shows the selected characters from the first 128 in the UTF-8 table. All the characters outside this range (0.03% of occurrences) are represented as a special OOV character. Another padding special character is used to make all the lines have the same fixed size. We set the maximum length of source code lines (after removing leading and trailing blanks) at 40, truncating longer lines and adding padding characters to shorter ones.

With these transformations, the vocabulary is streamlined to 97 characters. They are then converted to integers ranging from 0 to 96, where 0 represents padding, 1 corresponds to OOV, and subsequent numbers represent the ordered characters in Table 4. Each character (integer) represents a category value, devoid of any inherent order (i.e., a nominal variable, not an ordinal one). Consequently, we convert each character into one-hot encoding, a more fitting representation for machine learning. Each sample becomes a sparse vector of 97 dimensions, where each binary feature represents the presence or absence of the corresponding character. In this way, an instance (line of code) is represented as a 40×97 matrix of binary values.

To ensure dataset randomness, we execute various shuffling actions. Initially, all files are shuffled, irrespective of their programming language. Subsequently, after the line extraction module (Section 3.4), we shuffle all the lines within a file. Finally, the dataset creation module shuffles all individuals within the final dataset.

The resulting dataset comprises 858.223 million individuals. We store it in binary as a NumPy array, serialized using the `pickle` Python module. While less portable than a comma-separated file (CSV), its binary representation allows for faster loading. The dataset creation module was implemented in Python 3.10 [25].

4. Methodology

The generated dataset with 858.223 million instances is not balanced. Thus, we randomly downsample our dataset to obtain a perfectly balanced dataset for the 21 different languages. The balanced dataset, now totaling 434.18 million samples (the second part of our first contribution), is then divided into three stratified sets: the training set, comprising 432.18 million samples; and the validation and test sets, each containing 1,000,020 instances.

4.1. Deep single-line models

We tried to construct various single-line models for predicting the language of a single line of code. However, all the offline models we experimented with, including XGBoost, Random Forest, Support Vector Machine, and Softmax Regression [25], encountered memory resource issues when handling datasets exceeding 3.5 million instances. To fully leverage our dataset, we opted for training neural network models using mini-batch learning, wherein model parameters are updated based on the average gradient computed for each mini-batch of instances [29]. This approach eliminates the need to load the entire dataset into memory while ensuring that all instances contribute to the model training

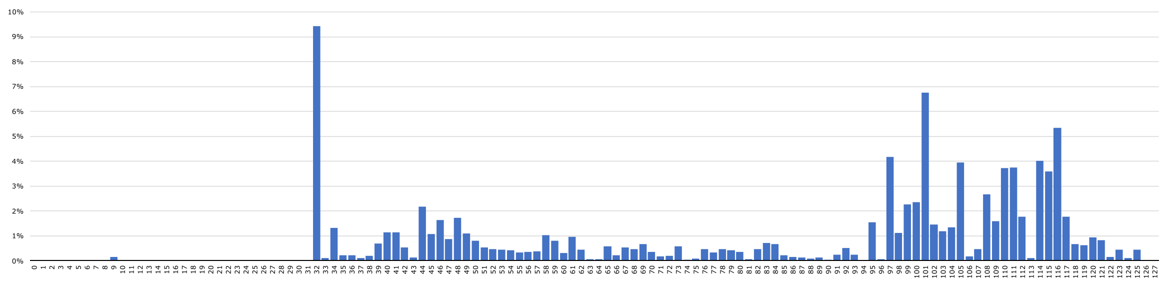


Fig. 3. Histogram showing the occurrence of characters in all the source-code lines extracted.

Table 4
Selected characters (bold font) from the first 128 in the UTF-8 table.

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
0	NUL	16	DLE	**32**	**ESP**	**48**	**0**	**64**	**@**	**80**	**P**	**96**	**‘**	**112**	**p**
1	SOH	17	DC1	**33**	**!**	**49**	**1**	**65**	**A**	**81**	**Q**	**97**	**a**	**113**	**q**
2	STX	18	DC2	**34**	**"**	**50**	**2**	**66**	**B**	**82**	**R**	**98**	**b**	**114**	**r**
3	ETX	19	DC3	**35**	**#**	**51**	**3**	**67**	**C**	**83**	**S**	**99**	**c**	**115**	**s**
4	EOT	20	DC4	**36**	**$**	**52**	**4**	**68**	**D**	**84**	**T**	**100**	**d**	**116**	**t**
5	ENQ	21	NAK	**37**	**%**	**53**	**5**	**69**	**E**	**85**	**U**	**101**	**e**	**117**	**u**
6	ACK	22	SYN	**38**	**&**	**54**	**6**	**70**	**F**	**86**	**V**	**102**	**f**	**118**	**v**
7	BEL	23	ETB	**39**	**’**	**55**	**7**	**71**	**G**	**87**	**W**	**103**	**g**	**119**	**w**
8	BS	24	CAN	**40**	**(**	**56**	**8**	**72**	**H**	**88**	**X**	**104**	**h**	**120**	**x**
9	**TAB**	25	EM	**41**	**)**	**57**	**9**	**73**	**I**	**89**	**Y**	**105**	**i**	**121**	**y**
10	LF	26	SUB	**42**	*****	**58**	**:**	**74**	**J**	**90**	**Z**	**106**	**j**	**122**	**z**
11	VT	27	ESC	**43**	**+**	**59**	**;**	**75**	**K**	**91**	**[**	**107**	**k**	**123**	**{**
12	FF	28	FS	**44**	**,**	**60**	**<**	**76**	**L**	**92**	\	**108**	**l**	**124**	**|**
13	CR	29	GS	**45**	**-**	**61**	**=**	**77**	**M**	**93**	**]**	**109**	**m**	**125**	**}**
14	SO	30	RS	**46**	**.**	**62**	**>**	**78**	**N**	**94**	**ˆ**	**110**	**n**	126	~
15	SI	31	US	**47**	**/**	**63**	**?**	**79**	**O**	**95**	**_**	**111**	**o**	127	DEL

process.

We devise two deep artificial neural network (ANN) topologies: one based on the Multi-Layer Perceptron (MLP) architecture and the other on Recurrent Neural Networks (RNN). For the MLP, Fig. 4 illustrates the architecture used. We explore two different inputs. The first one utilizes the one-hot representation described in Section 3.5, so the embeddings layer in Fig. 4 is not employed. The second input representation assigns each character a unique integer, which is then translated into a dense vector (embedding) of fixed size—the length of the embedding is a hyperparameter described in Section 4.3. The embedding layer learns to represent characters as fixed-size vectors, capturing the semantics required for language recognition.

Whether using the one-hot input or the embeddings layer, each line of code is represented as a matrix with 40 rows and 97 (or the size of the embedding) columns. As each hidden block of layers requires a one-dimensional input, a flatten layer converts the matrix to a 1D vector. Subsequently, the MLP concatenates n hidden blocks of layers (where n is a hyperparameter to be determined). Each hidden block starts with a fully connected (dense) layer of m neurons (another hyperparameter). A batch normalization layer is incorporated to maintain the mean output close to 0 and its standard deviation close to 1, thereby mitigating the vanishing gradient problem commonly observed in deep MLP networks [30]. Additionally, an optional dropout layer in each hidden block serves as a regularization mechanism to prevent overfitting.

After the concatenation of n hidden blocks, the MLP network concludes with a dense layer of 21 outputs (one for each programming language). This layer employs a softmax activation function, assigning to each output the probability that the input line belongs to one of the 21 programming languages.

Fig. 5 depicts the second neural network employed: a deep Bidirectional Recurrent Neural Network (BRNN). This type of network has demonstrated notable efficacy in classifying sequences—lines are sequences of characters—, considering both past and future contexts within sequences—the context of a character involves information from both preceding and succeeding characters [31].

Similar to the MLP architecture, we explore two input configurations for the BRNN: one utilizing one-hot encoding for characters and another one with an embedding layer to learn vector representations for each character. Subsequently, a series of bidirectional recurrent network blocks are stacked, the number of which is a hyperparameter. Each block features an initial layer with a recurrent forward cell (GRU or LSTM, determined by another hyperparameter) and a corresponding recurrent backward cell. The outputs of both cells (with the number of neurons being another hyperparameter) are concatenated and, optionally, passed through a dropout layer for regularization.

Following the BRNN blocks, we incorporate d dense layers (a hyperparameter) to take the information acquired from the sequence of characters and employ it in classifying the programming language. An optional dropout layer precedes the final dense layer, consisting of 21 output neurons with a softmax activation function. Analogous to the MLP topology, the output comprises 21 values between 0 and 1, representing the probability that the input line of code has been written in each programming language. For both architectures, we use the cross-entropy loss function.

4.2. Meta-model

The previous single-line models return 21 real values indicating the probability that the input line belongs to one of the 21 programming languages. To predict the language of a sequence of loc lines of code, the loc vectors produced by the model can be used. Various approaches exist for predicting the language of loc lines, such as averaging the probabilities for each line, using a majority voting system to determine the most frequent language, or performing a length-weighted average of the probabilities.

A more sophisticated approach is *stacking* (also called stack generalization or blending) [32]. Stacking is an ensemble learning technique used to improve the predictive performance of other machine learning models. In our case, we use stacking to train a meta-model (or blender) that predicts the language of loc lines of code by combining the

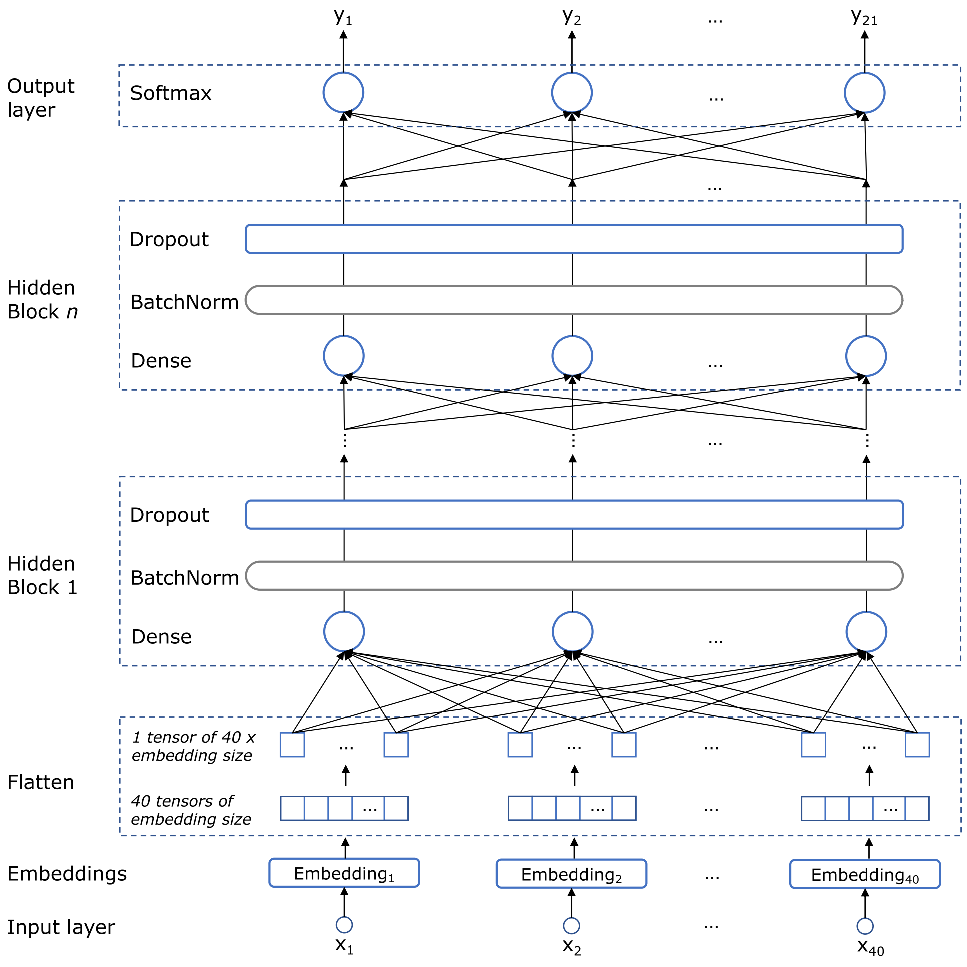


Fig. 4. Multi-layer perceptron architecture followed.

predictions produced by the best single-line model.

The input to the meta-model is a sequence of loc vectors (one per line). RNNs are commonly employed for such sequential inputs, processing each value in the sequence one step at a time, while maintaining a hidden state that captures information from previous steps. However, for long sequences (e.g., all the lines in a file) the effectiveness of RNNs is limited by sequence length and model architecture [33]. For this reason, we perform a frequency distribution analysis that aggregates the sequence of vectors into a fixed-size matrix.

The probability that each input line belongs to one of the l programming languages ($l = 21$) is categorized into b bins. Then, we compute a (l, b) matrix as indicated in Eq. (1). Each cell $M_{i,j}$ in the matrix counts the number of lines the ith language predicted by our single-line model within the probability of the given j bin ($prob_{k,i} \in bin(j)$). These values are finally divided by $loc \times l$ so that they represent probabilities—i.e., they all sum 1, regardless of the number of lines.

Once the sequence of loc vectors is converted into a fix-sized (l, b) matrix (both l and b are constant), we train a MLP neural network as our meta-model. The first layer performs batch normalization on the $l \times b$ inputs. This is followed by n dense hidden layers with h neurons each (where n and h are hyperparameters). The output is a dense layer with l ($l = 21$) neurons and a softmax activation function. The cross-entropy loss function is used for training.

$$M_{i,j} = \frac{\sum_{k=1}^{loc} \begin{cases} 1 & \text{if } prob_{k,i} \in bin(j) \\ 0 & \text{otherwise} \end{cases}}{loc \times l} \quad (1)$$

where $prob_{k,i}$ is the probability predicted by our single-line model

that the line k belongs to the language i

4.3. Hyperparameter tunning and model training

The different ANN topologies identified encompass numerous hyperparameters that play an important role in shaping the architecture and behavior of the networks. These hyperparameters significantly influence the network's capacity to learn and generalize from data. Some hyperparameters define the network architecture (e.g., the number of hidden layers and neurons per layer) and others pertain to the training process (e.g., batch size and learning rate).

To ensure the optimal performance of the designed ANN topologies, we conduct an exhaustive grid hyperparameter search for the single-line models and the meta-model. This method systematically explores a predefined set of hyperparameter values to identify the combination that produces the most favorable model performance. In each iteration of the search, a model is trained with the train set using a specific combination of hyperparameters. Subsequently, the validation set is employed to assess the model's performance—we use the accuracy measure since all the datasets are perfectly balanced. After completing the search process, we select the hyperparameters yielding the best performance.

Tables 5, 6, and 7 show all the values used for each hyperparameter in each model and meta-model, along with the values corresponding to the best performances. For each iteration of the hyperparameter search process, models are trained with 10 epochs. Concerning the initialization methods, He is utilized for the ReLU, Leaky ReLU, and

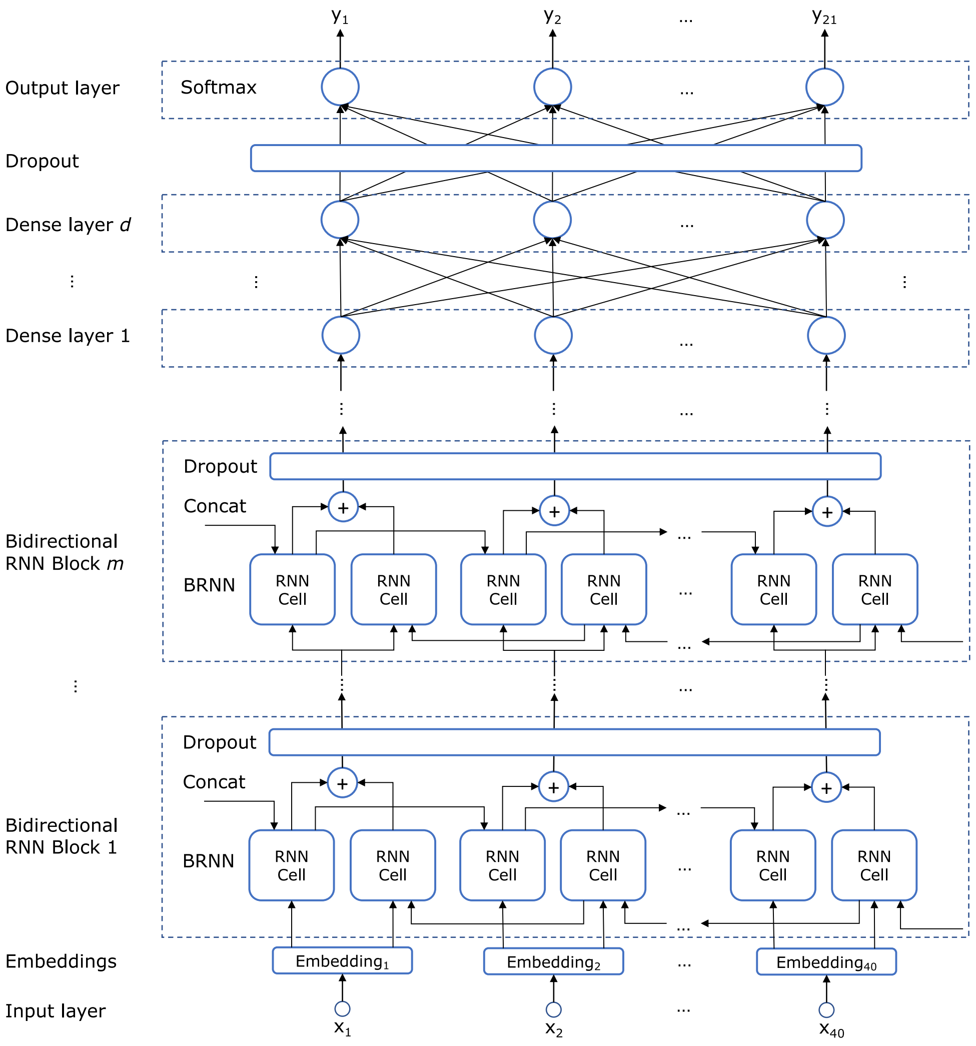


Fig. 5. Bidirectional recurrent neural network architecture followed.

Table 5
Hyperparameters of the deep multi-layer perceptron architecture followed (embedding size=0 means no embedding layer and input code in one hot).

Hyperparameter	Range	Result
Hidden blocks	[2, 4, 6, 8, 10]	8
Neurons in hidden dense layers	[500, 1000, 2000, 3000, 4000, 5000]	3000
Embedding size	[0, 7, 14, 28, 56]	28
Dropout factor	[0, 0.1, 0.2]	0
Activation function	[ELU, SELU, ReLU, Leaky ReLU]	ELU
Learning rate	[10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 0.1, 1, 10]	10^{-2}
Batch size	[32, 64, 128, 256, 512, 1024, 2048]	1024

Table 6
Hyperparameters of the bidirectional recurrent neural network architecture followed (embedding size=0 means no embedding layer and input code in one hot).

Hyperparameter	Range	Result
BRNN blocks	[2, 4, 6, 8, 10]	8
RNN cell	[LSTM, GRU]	LSTM
Embedding size	[0 , 8, 16, 24, 32, 64]	32
Output neurons of RNN cells	[32, 64, 128, 256, 512, 1024]	256
Hidden dense layers	[1, 2]	1
Neurons in hidden dense layers	[32, 64, 128, 256, 512, 1024]	512
Dropout factor	[0, 0.1, 0.2]	0.2
Activation function dense layers	[ELU, SELU, ReLU, Leaky ReLU]	ReLU
Learning rate	[10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 0.1, 1, 10]	10^{-3}
Batch size	[32, 64, 128, 256, 512, 1024, 2048]	256

ELU activations, while LeCun is applied for SELU [34].

Once the final set of hyperparameters is determined, we conduct a concluding fine-tuning training session. The stopping criterion is defined as an increase in validation loss over three consecutive epochs. The learning rate is dynamically reduced by a factor of 0.2 when validation loss is not reduced in the last epoch. We select the model that exhibits the best validation performance. The best MLP and BRNN models have 66,499,977 and 9,005,621 trainable parameters, respectively. The best MLP meta-model has 640,821 parameters.

Various optimizers were explored for the distinct ANN architectures. For the Multi-Layer Perceptron (MLP) topologies, we opted for the Nesterov Accelerated Gradient (NAG) optimizer with a momentum of 0.9 [35]. As for the Bidirectional Recurrent Neural Network (BRNN), the most favorable outcomes were achieved using Adam, a stochastic gradient descent method based on adaptive estimation of first-order and second-order moments [36].

For hyperparameter tuning and model training, we use two different computers, each dedicated to one of the ANN architectures presented in Sections 4.1 and 4.2. For the BRNN, we utilize a Bull Atos equipped with 2 AMD EPYC 7413 2.65 GHz processors, 512 GB RAM, and an

Table 7
Hyperparameters of the meta-model (multi-layer perceptron architecture).

Hyperparameter	Range	Result
Number of bins	[5, 10, 20, 50, 100, 500]	100
Hidden dense layers	[1, 2, 3]	1
Neurons in hidden dense layers	[50, 100, 200, 300, 400]	300
Activation function	[ELU, SELU, ReLU, Leaky ReLU]	ReLU
Learning rate	[10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 0.1, 1, 10]	10^{-2}
Batch size	[32, 64, 128, 256, 512]	128

Nvidia A100 GPU with 80 GB HBM2e. On the other hand, the two MLP models are trained with a Dell PowerEdge T630 featuring 2 x Intel Xeon E5-2630 2.4 GHz processors, 128 GB RAM, and an Nvidia GeForce RTX 2060 6 GB GDDR6.

4.4. Evaluation

As mentioned, the training set is utilized to train the models, and the validation set is employed for hyperparameter tuning. Model evaluation is conducted using the test set since these data have been utilized in neither the training nor the validation processes.

The metrics used to assess all the models encompass accuracy, precision, recall, and F_1-score. As detailed in Eq. (2), accuracy gauges how often a model correctly predicts the outcome, calculated by dividing the number of correct predictions by the total number of predictions—sometimes expressed as a percentage.

$$Accuracy = \frac{Correct\ predictions}{Total\ number\ of\ instances} \tag{2}$$

Another performance metric we use is F_1-score, computed as the harmonic mean of precision and recall metrics. Eqs. (3) and (4) elucidate the precision and recall metrics for each target class (output). Based on those two equations, macro-precision, and macro-recall are calculated by computing the arithmetic mean of precision and recall for all the classes in the output variable. On the other hand, micro-precision and micro-recall consider precision and recall globally across all classes, treating the entire set of predictions as a single binary classification problem.

$$Precision_i = \frac{TruePositives_i}{TruePositives_i + FalsePositives_i} \quad \text{where } i \in \text{classes} \tag{3}$$

$$Recall_i = \frac{TruePositives_i}{TruePositives_i + FalseNegatives_i} \quad \text{where } i \in \text{classes} \tag{4}$$

Micro F_1-score is computed as the harmonic mean of micro-precision and micro-recall (Eq. (5)). For macro F_1-score (Eq. (6)), individual F_1-scores are computed for each class (F_1^i in Eq. (6)) and then averaged. These metrics collectively provide a comprehensive evaluation of the models' performance across various aspects, contributing to a thorough understanding of their effectiveness.

$$Micro\ F_1\text{-}score = 2 \times \frac{Micro\text{-}Precision \times Micro\text{-}Recall}{Micro\text{-}Precision + Micro\text{-}Recall} \tag{5}$$

$$\begin{gathered} Macro\ F_1\text{-}score = \frac{F_1^1 + F_1^2 + \cdots + F_1^n}{n} \\ \text{where } F_1^i = 2 \times \frac{Precision_i \times Recall_i}{Precision_i + Recall_i} \quad \text{and} \quad i \in \text{classes} \end{gathered} \tag{6}$$

4.5. Selected systems

We assess the performance of the proposed models and compare them with the existing state-of-the-art systems described in Section 2. We specifically choose the systems designed for predicting programming languages from code snippets, excluding those requiring entire input files, natural language text, and code images. The selected works for comparison are DeepSCC [11], Guesslang [12], SCC++ [5], and EL-CodeBert [13] (we exclude SCC since SCC++ is an enhancement of SCC [10]).

As the five systems to be evaluated support different languages, we select those common to all, resulting in 15 languages (represented with ψ in Table 3). Subsequently, we construct test sets comprising snippets of various sizes and entire source code files for these 15 languages. As detailed in Section 3.3, we acquire 113.3 GB of verified source code files, allocating 5% of these files for testing purposes. These files are utilized to generate four test sets: one using the entire files, and three with code snippets with 10, 5, and 1 line(s) of code. For snippets, we extract n sequential and contiguous lines of code from each file (n being 10, 5, and 1). We construct four perfectly balanced (15 languages) test sets: 1,000,005 samples for each type of snippet and 90,000 samples for entire files.

To compare the performance of different models and determine statistically significant differences, confidence intervals are provided alongside the performance measures outlined in Section 4.4. To achieve this, we employ bootstrap resampling (10,000 boot samples) to estimate the models' performance (average performance of all boot samples) and calculate the 95% confidence intervals [37]. Bootstrap resampling involves repeatedly sampling with replacement from the observed data to create multiple resampled datasets. The resulting 95% confidence intervals enable us to determine whether statistically significant differences exist among the compared systems [38].

4.6. Resource consumption

In addition to evaluating the classification performance of our models, we also assess the computational resources they consume during inference. That is, we measure the execution time and memory consumption when the models are utilized by an application for source code classification.

For runtime performance analysis, we follow the approach proposed by [38]. First, we gauge the *startup* execution time required to load the model into memory and conduct a single prediction—one line of code for the single-line models and two lines for the meta-model. The clock time of the whole process is recorded 30 times. Subsequently, we compute the arithmetic mean of these values along with the corresponding 95% confidence interval [38].

The second evaluation method, termed *steady state* by Georges et al., involves measuring the execution time of a single operation (in our case, model and meta-model prediction) when the system is in a stable state [38]. This steady state is reached when the model has been loaded into memory, and previous predictions have been executed. The steady-state methodology only measures the execution time of model prediction. It assumes system stability when the coefficient of variation of the last 10 predictions is below 2%. Under such conditions, the execution time is computed as the arithmetic mean of the last 10 values. Similar to the startup evaluation, this process is repeated 30 times, returning the arithmetic mean and 95% confidence interval for the steady-state execution times [38].

Memory consumption assessment follows the startup methodology. For each program execution, we measure the peak size of the working set memory utilized by each process since its inception (i.e., `PeakWorkingSet`). The working set of a process is the set of memory pages currently visible to a process in physical RAM memory, which is available to be used without triggering a page fault. Memory consumption measurements are conducted using the `psutil` Python package [39] for CPU memory and `gpustat` for GPU memory [40]. The evaluation is performed on the Dell PowerEdge T630 computer equipped with Nvidia GeForce RTX 2060 described in Section 4.3.

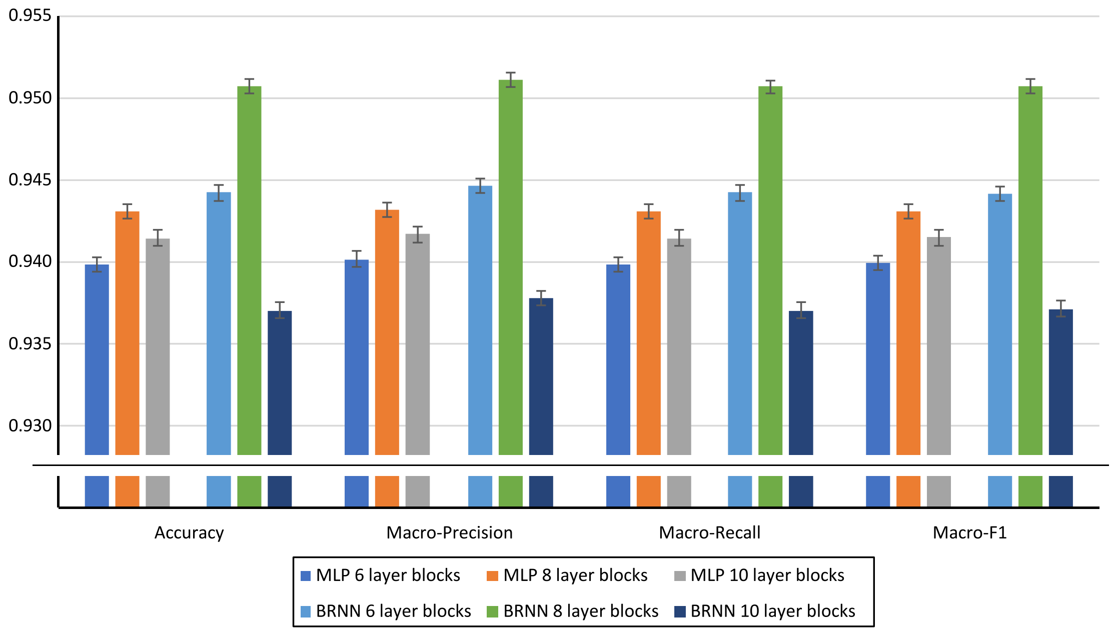


Fig. 6. Evaluation of the single-line models following the MLP and BRNN architectures outlined in Section 4.1. Bars show the average performance using the bootstrap resampling method described in Section 4.5, with whiskers indicating 95% confidence intervals. Notice that the Y-axis has been truncated to enhance visibility and highlight distinctions among the compared systems.

5. Results

We initially show the performance of the single-line models introduced in Section 4.1 for predicting 21 languages. Subsequently, we conduct a comparative analysis between our meta-model (that includes the top-performing single-line model) and the related work outlined in Section 4.5.

5.1. MLP and BRNN single-line models

Fig. 6 presents the accuracy and macro-averaging metrics for the top-performing MLP and BRNN single-line models, achieved through the hyperparameter tuning process outlined in Section 4.3. In a multiclass classification problem, where each instance can only have one single class, micro-averaging measures for precision, recall, and F_1-score have the same values as accuracy [41]. For this reason, we only show accuracy and macro-averaging metrics.

The peak performance, with an accuracy of 95.069% and a macro-F_1 of 95.071%, is attained by the BRNN architecture described in Section 4.1 for the hyperparameters outlined in Section 4.3. Fig. 6 provides a comparative analysis of models with 6, 8, and 10 BRRN blocks, showing how the one with 8 layers outperforms the other ones with significant differences—95% confidence intervals exhibit no overlap. The most effective MLP configuration features 8 layer blocks, delivering an accuracy of 94.31% and a macro F_1-score of 94.308%.

5.1.1. Discussion

Fig. 7 illustrates the proficiency of the optimal BRNN model in detecting the programming language of a single line of code. The upper part of Fig. 7 portrays the distribution of a stratified random sample comprising 1000 instances drawn from the test set. Given that these instances are 40 × 32-dimensional, we apply the t-distributed Stochastic Neighbor Embedding (t-SNE) algorithm [42] to reduce their dimensionality to 2D. The lower part of Fig. 7 shows the output of the second last dense layer in the BRNN for the same random sample. It can be seen how the deep network effectively segregates instances across the 21 distinct programming languages.

The confusion matrix presented in Fig. 8 offers more detailed insight into the BRNN model's performance. Due to its distinctive syntax, CSS achieves the highest classification accuracy (99.8%). Following closely are Go (98.4%), Assembly (98.3%), Matlab (98.2%), Perl (97.7%), Python (97.5%), Unix Shell (97.2%), and SQL (97.1%). R, C, PHP, TypeScript, and Ruby exhibit classification accuracies above 95%.

We discuss classifications with an error rate exceeding 1% in the confusion matrix in Fig. 8. The predominant misclassification involves labeling C++ as C (7.3% error)—a predictable outcome given that C is a subset of C++. Something similar happens between TypeScript and JavaScript: TypeScript is a superset of JavaScript, causing 2.9% and 2.3% classification error between them. Certain misclassifications arise among C-syntax-based languages like Java, C#, C, and C++, owing to their analogous syntax. HTML is occasionally mistaken for other languages commonly embedded within it, such as CSS, JavaScript, and PHP.

We perform a more detailed analysis to identify some other minor wrong classifications—all the misclassified instances can be found on [43]. After analyzing the misclassifications of Java, Kotlin, and Scala, we found out they are caused by the following features shared by these languages: they all provide `package` and `import` clauses, share the same syntax for method invocation and field access, utilize the same API comprising all the classes of the Java platform, employ the same *camelCase* naming convention for methods and fields (unlike C#, where methods and fields start with capital letter), and the syntax of `if` and `while` statements and most operators are the same.

Swift is occasionally misclassified as Kotlin. That is mostly because of the construction "`var` *<ID>* `:` *<type>* `=` *<exp>*" for variable definition provided by both languages. Something similar happens between Kotlin and Scala: they both provide "`val` *<ID>* `:` *<type>* `=` *<exp>*" for constant definition and "`private var` *<ID>* `:` *<type>* `=` *<exp>*" for field definition. Ruby and Python share different pretty rare features: `def` for function and method definition, `self` for the implicit object, *snake_case* naming convention, `raise` for throwing exception, lists literals (arrays in Ruby) with `[` and `]`, and multiple (parallel in Ruby) assignment with comma separated expressions (Python tuples). Finally, Swift is sometimes classified as Go because they both provide `nil`, and `if` and `switch` constructs without parenthesis.

After performing the error analysis described in the previous paragraphs, we can raise the following discussion. We found out that all the misclassified individuals indeed represent valid lines of code for

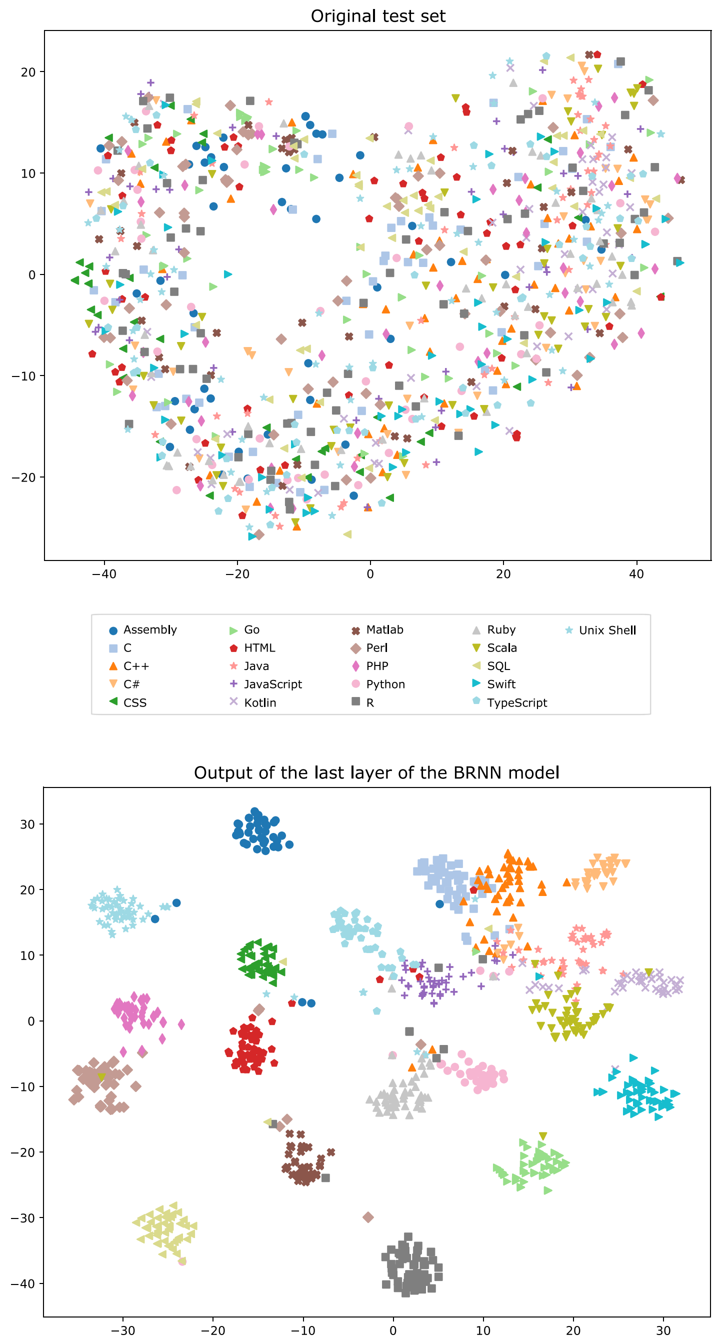


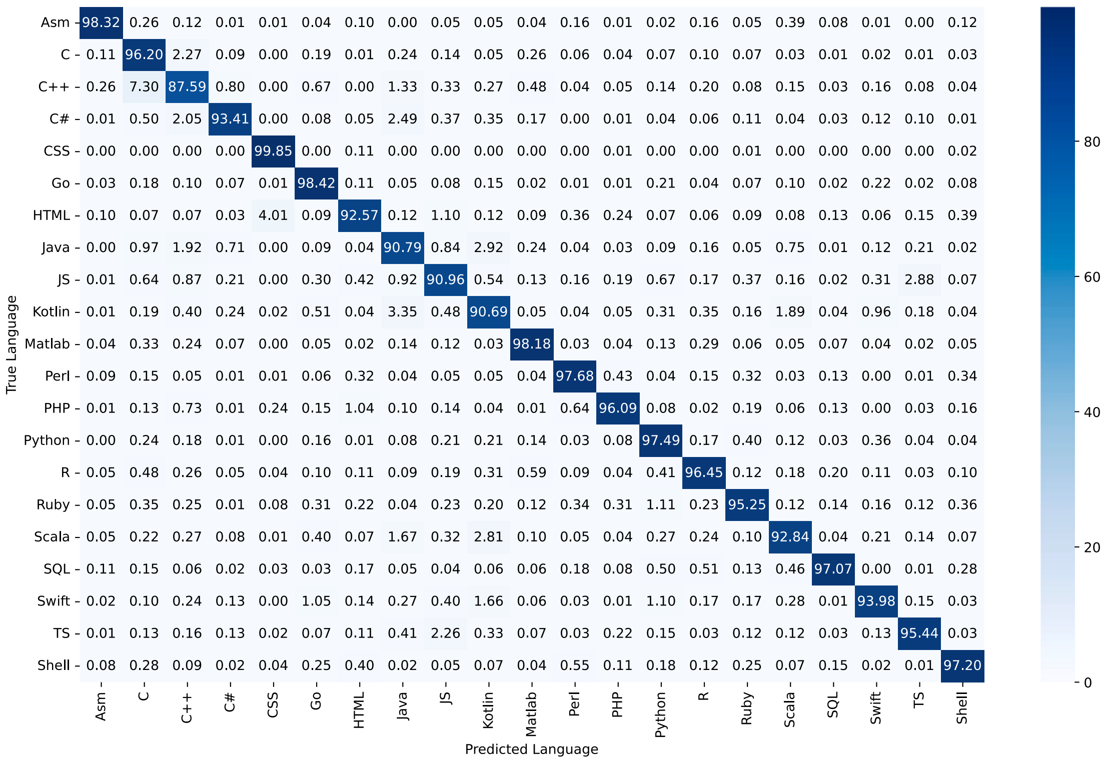
Fig. 7. Above, the distribution of a stratified random sample of 1000 instances from the test set, after reducing its dimensionality with the t-SNE algorithm. Below, the output of the second last layer of the BRNN model for the same instances, with its dimensionality reduced using t-SNE.

both the actual and predicted languages [43]. Moreover, many of these lines are also correct in other languages (e.g., "`return false;`"). In such cases, the classifier learns to assign a language to one line of code based on how likely it is for that language, even though it is correct for other languages. For instance, the statement "`object.Method()`" is identified as C# instead of Java because, although it is possible, it is not common to name Java methods with the first letter as capital. As the programmer adds more lines of code, this misclassification will be corrected, allowing the meta-model to accurately determine the actual language of the source code (next subsection).

5.2. Comparison with related work

Fig. 9 and Table 8 present the evaluation results of the selected systems detailed in Section 4.5.[1] Our system, *PLangRec*, uses the single-line model with the best performance obtained in Section 5.1: BRNN architecture with 8 layers. As highlighted in Section 4.5, we utilize

[1] It can be seen how the performance outcomes for a single line of code in Table 8 are higher than those presented in Section 5.1 This difference is caused by the distinct number of languages classified (15 vs. 21).



	Asm	C	C++	C#	CSS	Go	HTML	Java	JS	Kotlin	Matlab	Perl	PHP	Python	R	Ruby	Scala	SQL	Swift	TS	Shell
Asm	98.32	0.26	0.12	0.01	0.01	0.04	0.10	0.00	0.05	0.05	0.04	0.16	0.01	0.02	0.16	0.05	0.39	0.08	0.01	0.00	0.12
C	0.11	96.20	2.27	0.09	0.00	0.19	0.01	0.24	0.14	0.05	0.26	0.06	0.04	0.07	0.10	0.07	0.03	0.01	0.02	0.01	0.03
C++	0.26	7.30	87.59	0.80	0.00	0.67	0.00	1.33	0.33	0.27	0.48	0.04	0.05	0.14	0.20	0.08	0.15	0.03	0.16	0.08	0.04
C#	0.01	0.50	2.05	93.41	0.00	0.08	0.05	2.49	0.37	0.35	0.17	0.00	0.01	0.04	0.06	0.11	0.04	0.03	0.12	0.10	0.01
CSS	0.00	0.00	0.00	0.00	99.85	0.00	0.11	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.02
Go	0.03	0.18	0.10	0.07	0.01	98.42	0.11	0.05	0.08	0.15	0.02	0.01	0.01	0.21	0.04	0.07	0.10	0.02	0.22	0.02	0.08
HTML	0.10	0.07	0.07	0.03	4.01	0.09	92.57	0.12	1.10	0.12	0.09	0.36	0.24	0.07	0.06	0.09	0.08	0.13	0.06	0.15	0.39
Java	0.00	0.97	1.92	0.71	0.00	0.09	0.04	90.79	0.84	2.92	0.24	0.04	0.03	0.09	0.16	0.05	0.75	0.01	0.12	0.21	0.02
JS	0.01	0.64	0.87	0.21	0.00	0.30	0.42	0.92	90.96	0.54	0.13	0.16	0.19	0.67	0.17	0.37	0.16	0.02	0.31	2.88	0.07
Kotlin	0.01	0.19	0.40	0.24	0.02	0.51	0.04	3.35	0.48	90.69	0.05	0.04	0.05	0.31	0.35	0.16	1.89	0.04	0.96	0.18	0.04
Matlab	0.04	0.33	0.24	0.07	0.00	0.05	0.02	0.14	0.12	0.03	98.18	0.03	0.04	0.13	0.29	0.06	0.05	0.07	0.04	0.02	0.05
Perl	0.09	0.15	0.05	0.01	0.01	0.06	0.32	0.04	0.05	0.05	0.04	97.68	0.43	0.04	0.15	0.32	0.03	0.13	0.00	0.01	0.34
PHP	0.01	0.13	0.73	0.01	0.24	0.15	1.04	0.10	0.14	0.04	0.01	0.64	96.09	0.08	0.02	0.19	0.06	0.13	0.00	0.03	0.16
Python	0.00	0.24	0.18	0.01	0.00	0.16	0.01	0.08	0.21	0.21	0.14	0.03	0.08	97.49	0.17	0.40	0.12	0.03	0.36	0.04	0.04
R	0.05	0.48	0.26	0.05	0.04	0.10	0.11	0.09	0.19	0.31	0.59	0.09	0.04	0.41	96.45	0.12	0.18	0.20	0.11	0.03	0.10
Ruby	0.05	0.35	0.25	0.01	0.08	0.31	0.22	0.04	0.23	0.20	0.12	0.34	0.31	1.11	0.23	95.25	0.12	0.14	0.16	0.12	0.36
Scala	0.05	0.22	0.27	0.08	0.01	0.40	0.07	1.67	0.32	2.81	0.10	0.05	0.04	0.27	0.24	0.10	92.84	0.04	0.21	0.14	0.07
SQL	0.11	0.15	0.06	0.02	0.03	0.03	0.17	0.05	0.04	0.06	0.06	0.18	0.08	0.50	0.51	0.13	0.46	97.07	0.00	0.01	0.28
Swift	0.02	0.10	0.24	0.13	0.00	1.05	0.14	0.27	0.40	1.66	0.06	0.03	0.01	1.10	0.17	0.17	0.28	0.01	93.98	0.15	0.03
TS	0.01	0.13	0.16	0.13	0.02	0.07	0.11	0.41	2.26	0.33	0.07	0.03	0.22	0.15	0.03	0.12	0.12	0.03	0.13	95.44	0.03
Shell	0.08	0.28	0.09	0.02	0.04	0.25	0.40	0.02	0.05	0.07	0.04	0.55	0.11	0.18	0.12	0.25	0.07	0.15	0.02	0.01	97.20

Fig. 8. Confusion matrix for the BRNN classifier.

Table 8
Accuracy and macro F_1-score measures of the evaluated systems for snippets of 1, 5, and 10 lines of code, and entire files. 95% confidence intervals are expressed as percentages. XGB stands for XGBoost and RF stands for Random Forest.

System	Accuracy				Macro F_1-score			
	1-line	5-lines	10-lines	Entire files	1-line	5-lines	10-lines	Entire files
Guesslang	0.2457 ± 0.3%	0.6519 ± 0.1%	0.7854 ± 0.1%	0.9511 ± 0.1%	0.1942 ± 0.4%	0.6421 ± 0.1%	0.7806 ± 0.1%	0.9508 ± 0.1%
SCC++ XGB	0.3812 ± 0.3%	0.6268 ± 0.2%	0.7176 ± 0.1%	0.9071 ± 0.2%	0.4015 ± 0.2%	0.6268 ± 0.1%	0.7155 ± 0.1%	0.9049 ± 0.2%
SCC++ RF	0.3831 ± 0.3%	0.6130 ± 0.2%	0.7060 ± 0.1%	0.9100 ± 0.2%	0.3958 ± 0.2%	0.6125 ± 0.2%	0.7041 ± 0.1%	0.9082 ± 0.2%
DeepSCC	0.6269 ± 0.2%	0.8065 ± 0.1%	0.8524 ± 0.1%	0.9340 ± 0.2%	0.6288 ± 0.1%	0.8060 ± 0.1%	0.8517 ± 0.1%	0.9339 ± 0.2%
EL-CodeBert	0.6697 ± 0.1%	0.8367 ± 0.1%	0.8715 ± 0.1%	0.8985 ± 0.2%	0.6696 ± 0.1%	0.8360 ± 0.1%	0.8715 ± 0.1%	0.8981 ± 0.2%
PLangRec	0.9604 ± 0.1%	0.9886 ± 0.1%	0.9916 ± 0.1%	0.9953 ± 0.1%	0.9605 ± 0.1%	0.9886 ± 0.1%	0.9916 ± 0.1%	0.9953 ± 0.1%

entire files and snippets spanning 1, 5, and 10 lines of code written in 15 distinct languages (those supported by all the assessed models). When classifying a single line of code, *PLangRec* uses the BRNN model. For multiple lines, it utilizes the meta-model that, in turn, leverages the BRNN model.

Fig. 9 shows how *PLangRec* outperforms the rest of the systems for all kinds of snippets and whole files. Its macro F_1-score ranges from 96.05% (one line of code) up to 99.53% (entire files). For code snippets, EL-CodeBert ranks as the second-best model, achieving 66.96% (1 line) to 87.15% (10 lines) macro-F_1, showing significant lower performance than our system. For entire files, Guesslang is the second-best system, with a macro-F_1 of 95.08%, considerably lower than the 99.53% achieved by *PLangRec*. The third model, another LLM-based system named DeepSCC, shows F_1-scores ranging from 62.88% to 93.39%. The XGBoost model of SCC++ performs slightly better than its Random Forest model across most of the evaluation scenarios.

5.2.1. Discussion

We have seen how LLM-based models (EL-CodeBert and DeepSCC) have the closest performance to *PLangRec* for code snippets. There are some differences between those models and ours. EL-CodeBert and DeepSCC are word-level deep models, whereas *PLangRec* is a char-level model. The multi-headed attention mechanism implemented by these systems might be not as powerful for programming language recognition as it is for NLP, when using char-level inputs. Another important difference between our system and the LLM-based models lies in the datasets used to train the models—our dataset is 3.2 orders of magnitude larger.

As expected, the classification performance of all the systems grows as the number of lines of code in the snippets increases. However, the growing curve is significantly different. Our system provides the greatest performance gains compared to the related systems when one single line of code is used, showing its greatest benefits when the input is small. The stacking ensemble meta-model used also allows *PLangRec* to outperform the rest of the systems for increasing lines of code, resulting in the best system to classify not only single lines of code but also snippets and entire files.

In the particular case of multi-line inputs, there are potential improvements that can be made to the preprocessing of the source code (Section 3.4). Currently, we remove comments, as they do not typically represent code and most comments are in natural language. However, when processing multiple lines, delimiting characters of comments could provide valuable context. For this scenario, the input preprocessor could consider the opening and closing comment characters to provide additional information about the language. Another way the preprocessor could be modified to enhance the performance of our system would be to detect and discard embedded code. For example, HTML and PHP often include embedded JavaScript and CSS code.

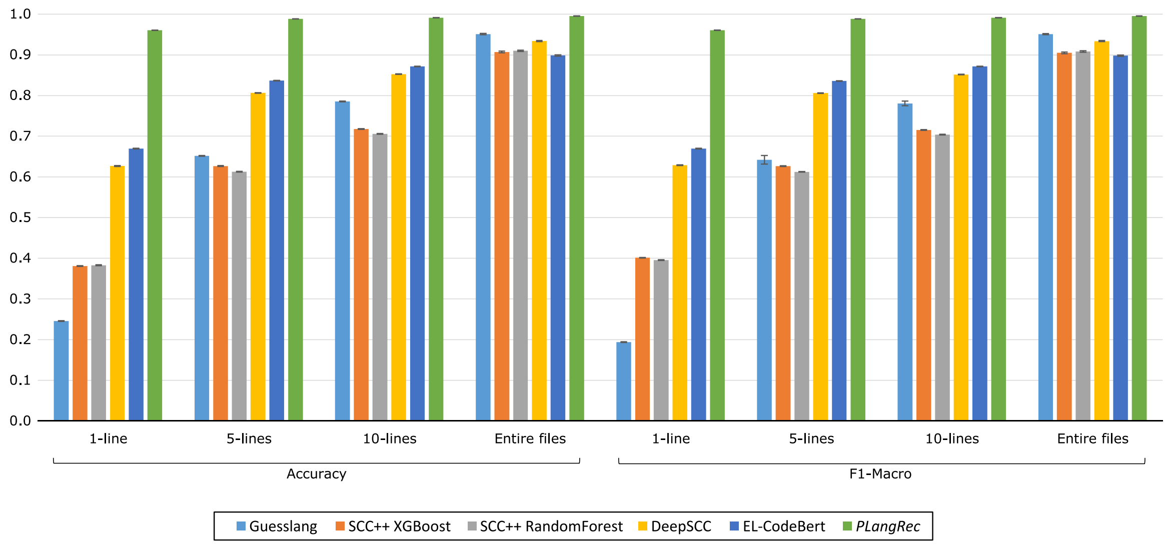


Fig. 9. Evaluation of the six systems for programming language prediction from source code. Bars show the average performance using the bootstrap resampling method described in Section 4.5, with whiskers indicating 95% confidence intervals.

This embedded code is easily identifiable through specific tags and attributes, and its removal could streamline the preprocessing and improve overall accuracy.

PLangRec currently supports 21 languages. To extend the system to predict new languages, an incremental learning approach could be followed [44]. This method updates the existing model with new language data without requiring a complete retraining from scratch. Specifically, the final layer of the single-line model should be extended to accommodate the new number of languages. Subsequently, the existing model can be fine-tuned on a dataset that includes both the new languages and a subset of the original languages, thereby saving training time and computational resources. To prevent the model from forgetting previously learned languages, Elastic Weight Consolidation (EWC) regularization can be applied [45]. While the meta-model will need to be retrained due to modifications in its input layer, this process is not computationally expensive—it takes 66 min to train the meta-model in the Dell PowerEdge T630 computer described in Section 4.3.

5.3. Resource consumption

Tables 9 and 10 depict the runtime performance of our top MLP and BRNN single-line models (Section 5.1), as well as the entire *PLangRec* system (i.e., the meta-model combined with the BRNN model), under both CPU and GPU configurations on the computer identified in Section 4.6. Table 9 highlights how the BRNN model takes significantly longer to load into memory (and perform one first prediction) than MLP (15.4 vs. 8.3 s) when a GPU is not used. For two lines of code, the meta-model invokes the BRNN single-line model twice, constructs the meta-model input, and passes it to the MLP meta-model (Section 4.2). This process requires only 5.7% more execution time than using the BRNN model alone, as the majority of the startup execution time is spent loading the models into memory.

When employing a GPU, execution time experiences a 7.5% increase for MLP, 6.3% for BRNN, and 6% for the whole system. Therefore, using a GPU does not improve startup execution time for a single prediction. Next, we evaluate whether this is the case for multiple predictions and when the models are already loaded into memory.

In Table 10, the execution time for language prediction is presented, assuming that the models have been loaded into memory and the system has reached a steady state (Section 4.6). Across all configurations, the MLP single-line model is faster than BRNN. The recurrent processing of data makes BRNN require more execution time than a feed-forward MLP, even though the latter has more parameters than the former. Predicting a single line of code takes between 54.9 (MLP GPU) and 185.4 ms (BRNN GPU). However, when 1000 instances are fed to the models in one batch, one prediction ranges between 0.36 (MLP GPU) and 6.5 ms (BRNN CPU). For two lines of code, the meta-model requires 1.73 more execution time than BRNN since it performs two invocations to that single-line model plus the forthcoming call to the MLP meta-model—this value is reduced to 1.1 when 1000 instances are classified in the CPU configuration.

Similar to the startup phase, using a GPU for a single prediction does not provide any runtime performance improvement under steady-state conditions. However, the GPU offers substantial performance gains during inference when 1000 instances are processed by *PLangRec*. Under these conditions, the improvements are 66.5% for the MLP model, a tenfold increase for BRNN, and an eightfold gain for the meta-model.

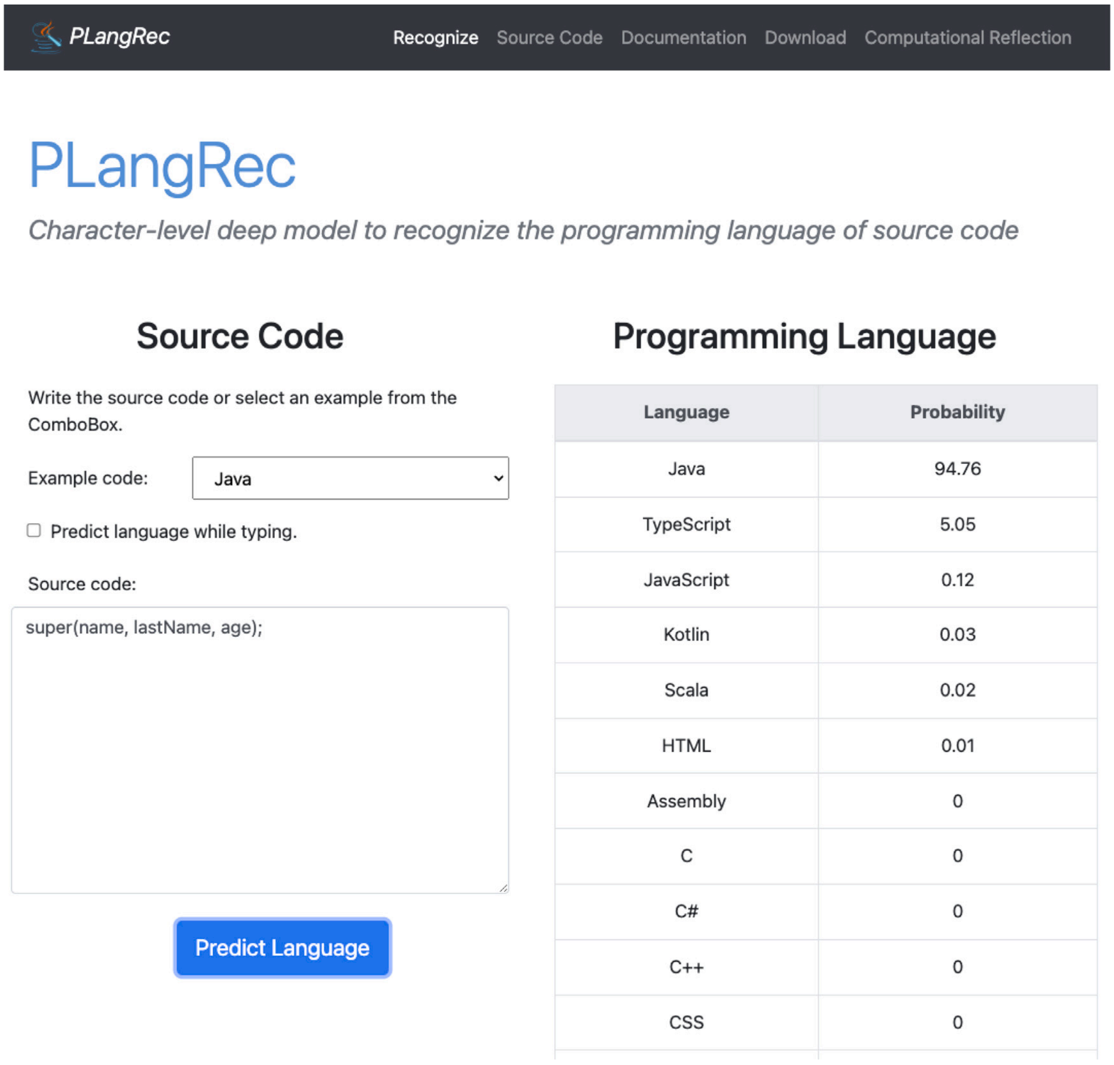
Table 11 outlines the virtual memory consumption measurements for the two top single-line models and the meta-model. In CPU, the MLP single-line model consumes more memory than BRNN because of its higher parameter count. Surprisingly, the meta-model, which includes the BRNN model, only consumes 0.66% more memory than BRNN, showing the small memory resources consumed by the MLP ensemble meta-model.

When a GPU is used, CPU memory usage increases by an average of 1.1 GB. Analyzing GPU memory – a critical resource due to its high cost – reveals that the BRNN single-line model requires 12 times more memory than MLP. This discrepancy may be due to differences in how GPUs handle parallel processing by managing memory differently from CPUs. When predicting two lines of code using the meta-model, CPU memory usage remains largely unchanged, while GPU memory requirements increase by 26.4%.

Although the required hardware resources are not excessively demanding, they may pose a barrier to deploying *PLangRec*. To address this, we provide a web API for programming language prediction using *PLangRec* (Section 6). This solution avoids any hardware limitations, requiring only an Internet connection.

6. Software applications

To show the functionality and assess the performance of our system, we have developed three different software applications. These applications are freely available for download. Our system, called *PLangRec* (Programming Language Recognizer), is accessible as a web application, web API, and a Python desktop application. All these versions of



PLangRec

Character-level deep model to recognize the programming language of source code

Source Code

Write the source code or select an example from the ComboBox.

Example code: Java

☐ Predict language while typing.

Source code:

```
super(name, lastName, age);
```

Programming Language

Language	Probability
Java	94.76
TypeScript	5.05
JavaScript	0.12
Kotlin	0.03
Scala	0.02
HTML	0.01
Assembly	0
C	0
C#	0
C++	0
CSS	0

Fig. 10. *PLangRec* web application.

Table 9
Execution time (seconds) of a process that loads the models into memory and perform one single prediction (startup)–the meta-model includes the single-line BRNN model.

Hardware	Model	Mean	95% Confidence interval
CPU	MLP	8.308	(8.276, 8.380)
	BRNN	15.449	(15.391, 15.606)
	Meta-model	16.329	(16.267, 16.495)
GPU	MLP	8.931	(8.734, 9.041)
	BRNN	16.430	(16.237, 16.501)
	Meta-model	17.308	(17.244, 17.484)

Table 10
Execution time (milliseconds) of one prediction, when the models have been loaded into memory and the system reaches a steady state—the meta-model includes the single-line BRNN model.

Hardware	N. instances in batch	Model	Mean	95% Confidence interval
CPU	1	MLP	126.823	(124.614, 130.698)
	1	BRNN	173.435	(172.113, 174.449)
	1	Meta-model	472.865	(469.261, 475,630)
	1000	MLP	0.596	(0.591, 0.602)
	1000	BRNN	6.464	(6.278, 6.825)
	1000	Meta-model	13.624	(13.490, 13.727)
GPU	1	MLP	54.904	(54.858, 55.469)
	1	BRNN	185.352	(170.544, 192.577)
	1	Meta-model	505.356	(501.504, 508.311)
	1000	MLP	0.358	(0.348, 0.371)
	1000	BRNN	0.552	(0.535, 0.556)
	1000	Meta-model	1.505	(1.494, 1.514)

PLangRec can be downloaded from [46].

The *PLangRec* web API offers a single `predict` GET method, which accepts the input source code (via the `source_code` parameter) and returns the probabilities of the code being written in 21 different programming languages. In cases where the input code spans multiple lines, the meta-model described in Section 4.2 is employed to predict the programming language. The web API is hosted at https://www.reflection.uniovi.es/plangrec/webapi/BRNN/predict—the source code must be passed as a `source_code` GET parameter.

The *PLangRec* web application is a straightforward JavaScript application that consumes the web API. Fig. 10 illustrates its user-friendly interface. Users can input the source code into the text area or select examples from different pieces of code for the 21 programming languages supported, using the dropdown list. Upon clicking the “predict language” button, the probabilities for the 21 languages (sorted in descending order) inferred by *PLangRec* are displayed in the “programming language” area. Both languages and probabilities can be sorted by clicking on their respective titles. Additionally, the application supports

Table 11
CPU and GPU memory (GBs) consumed by the models—the meta-model includes the single-line BRNN model.

Hardware	Model	CPU Memory	GPU Memory
CPU	MLP	5.479	–
	BRNN	5.310	–
	Meta-model	5.345	–
GPU	MLP	6.116	0.060
	BRNN	6.710	0.780
	Meta-model	6.754	0.986

language prediction while typing if the checkbox above the source code text area is enabled.

The web application is compatible with any web browser and requires no additional software installation. We have implemented it following responsive web design principles and tested the application across various devices and window sizes. It can be accessed at https://www.reflection.uniovi.es/bigcode/plangrec.

Furthermore, *PLangRec* is available as a Python desktop application [46]. Its functionality closely mirrors that of the web application and represents an example of how the meta-model, together with the single-line BRNN model, could be included in any Python application. All the required packages are detailed in the `requirements.txt` file, to be installed with the PIP Python package manager. Both the meta-model and the BRNN model are also available for download [43]. The first time you run *PLangRec*, it downloads the models from the Internet. A detailed description of *PLangRec*'s functionality, installation instructions, and examples are available at https://github.com/ComputationalReflection/PLangRec.

7. Conclusions

We present a character-level bidirectional RNN deep neural network to predict the programming language of a single line of code with 95.07% accuracy and macro-F_1 score. To train all the parameters of our deep model, we create a perfectly balanced dataset of 434.18 million individuals sourced from 123 GB of code downloaded from GitHub. The neural network architecture comprises eight blocks of bidirectional LSTM recurrent layers, augmented by one dense layer featuring 512 units for final classification, resulting in a total of 9 million parameters. Notably, this architecture outperforms a deep MLP network possessing seven times more parameters. For predictions involving multiple lines of code, a stacking ensemble meta-model leverages the BRNN single-line model to efficiently determine the programming language.

Compared to the state-of-the-art systems, *PLangRec* outperforms the rest of the systems in classifying individual lines, 5- and 10-line snippets, and entire source code files. The key elements that make our system surpass the existing ones are its char-level approach, the comprehensive dataset built to train its model, the deep neural network architecture proposed, and the stacking ensemble meta-model for classifying multiple lines of code.

We provide detail insights into the runtime performance and memory consumption of our model. Additionally, for users with limited resources, we offer a free web API for source code classification, requiring only an Internet connection. A web application and desktop program are also freely available.

Future research endeavors will explore the evaluation of our single-line model trained with varying numbers of characters and see its impact on the systems's performance. Furthermore, we plan to assess the effectiveness of multiheaded attention for char-level source code classification, since word-level multiheaded attention has been shown effective in addressing long-range dependencies in RNNs [47].

The source code and binaries of *PLangRec*, the serialized MLP and BRNN single-line models, the meta-model, the source code used to train all the models, the evaluation data, the corpus, and the dataset presented in this article are freely available for download (MIT license) at https://www.reflection.uniovi.es/bigcode/download/2024/plangrec.

CRediT authorship contribution statement

Oscar Rodriguez-Prieto: Methodology; Software; Validation; Formal analysis; Investigation; Resources; Data curation; Writing – review & editing; Supervision. **Alejandro Pato:** Software; Investigation; Data curation. **Francisco Ortin:** Conceptualization; Methodology; Software; Validation; Formal analysis; Investigation; Resources; Writing – original draft; Writing – review & editing; Visualization; Supervision; Project administration; Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially funded by the Spanish Department of Science, Innovation and Universities: project RTI2018-099235-B-I00. We have also received funds from the University of Oviedo, Spain through its support of official research groups (GR-2011-0040). Project GRU-GIC-24-070 from the Government of the Principality of Asturias, funded by the European Regional Development Fund (ERDF).

Data availability

Data is available for download on the support webpage created for the article.

References

[1] F. Ortin, G. Facundo, M. Garcia, Analyzing syntactic constructs of java programs with machine learning, Expert Syst. Appl. 215 (2023) 119398.
[2] GitHub, Where the world builds software, 2024, https://github.com/about.
[3] M. Gabbrielli, S. Martini, S. Giallorenzo, Programming Languages: Principles and Paradigms, Second Edition, Undergraduate Topics in Computer Science, Springer, 2023.
[4] F.D. Bonifro, M. Gabbrielli, S. Zacchiroli, Content-based textual file type detection at scale, in: Proceedings of the 2021 13th International Conference on Machine Learning and Computing, ICMLC '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 485–492.
[5] K. Alrashedy, D. Dharmaretnam, D.M. German, V. Srinivasan, T. Aaron Gulliver, SCC++: Predicting the programming language of questions and snippets of Stack Overflow, J. Syst. Softw. 162 (C) (2020).
[6] M.M. Öztürk, Developing a hyperparameter optimization method for classification of code snippets and questions of stack overflow: HyperSCC, EAI Endorsed Trans. Scalable Inf. Syst. 10 (1) (2022) 1–15.
[7] J.K. Van Dam, V. Zaytsev, Software language identification with natural language classifiers, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER, Vol. 1, 2016, pp. 624–628.
[8] S. Gilda, Source code classification using neural networks, in: 2017 14th International Joint Conference on Computer Science and Software Engineering, JCSSE, Vol. 1, 2017, pp. 1–6.
[9] E.O. Kiyak, A.B. Cengiz, K.U. Birant, D. Birant, Comparison of image-based and text-based source code classification using deep learning, SN Comput. Sci. 1 (5) (2020).
[10] K. Alreshedy, D. Dharmaretnam, D.M. German, V. Srinivasan, T.A. Gulliver, SCC: Automatic classification of code snippets, in: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation, SCAM, 2018, pp. 203–208.
[11] G. Yang, Y. Zhou, C. Yu, X. Chen, DeepSCC: Source code classification based on fine-tuned RoBERTa, in: International Conference on Software Engineering and Knowledge Engineering, in: SEKE' 21, 2021, pp. 499–502.
[12] Y. Somda, Guesslang: detect the programming language of a source code, 2024, https://guesslang.readthedocs.io/en/latest/.
[13] K. Liu, G. Yang, X. Chen, Y. Zhou, EL-CodeBert: Better exploiting CodeBert to support source code-related classification tasks, in: Proceedings of the 13th Asia-Pacific Symposium on Internetware, Internetware '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 147–155.
[14] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, RoBERTa: A robustly optimized BERT pretraining approach, 2019, arXiv:1907.11692.

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

[16] R. Sennrich, B. Haddow, A. Birch, Neural machine translation of rare words with subword units, in: K. Erk, N.A. Smith (Eds.), Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Berlin, Germany, 2016, pp. 1715–1725.

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A pre-trained model for programming and natural languages, in: T. Cohn, Y. He, Y. Liu (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2020, Association for Computational Linguistics, Online, 2020, pp. 1536–1547.

[18] J.N. Khasnabish, M. Sodhi, J. Deshmukh, G. Srinivasaraghavan, Detecting programming language from source code using Bayesian learning techniques, in: IAPR International Conference on Machine Learning and Data Mining in Pattern Recognition, 2014, pp. 513–522.

[19] A.H. Odeh, M. Odeh, H. Odeh, N. Odeh, Using natural language processing for programming language code classification with multinomial naive Bayes, Int. Inf. Eng. Technol. Assoc. (2023) 1229–1236.

[20] J. Hong, O. Mizuno, M. Kondo, An empirical study of source code detection using image classification, in: Proceedings of 10th International Workshop on Empirical Software Engineering in Practice, IWESEP 2019, 2019, pp. 1–15, Tokyo, Japan.

[21] Tiobe, Tiobe language ranking, 2024, https://www.tiobe.com/tiobe-index/.

[22] PYPL, The PYPL popularity of programming language index, 2024, https://pypl.github.io/PYPL.html.

[23] S. O'Grady, The RedMonk programming language ranking, 2024, https://redmonk.com/sogrady/2024/03/08/language-rankings-1-24/.

[24] PyGithub, Typed interactions with the GitHub API, 2024, https://github.com/PyGithub/PyGithub.

[25] A. Pato, Identificación Del Lenguaje De Programación a Partir Del Código Fuente, Trabajo Fin de Grado, Universidad de Oviedo, 2023, p. 89, doi:https://hdl.handle.net/10651/71443.

[26] T. Parr, The Definitive ANTLR 4 Reference, second ed., Pragmatic Bookshelf, Raleigh, NC, 2013.

[27] A. Project, Grammars written for ANTLR v4, 2024, https://github.com/antlr/grammars-v4.

[28] J. Eichstaedt, M. Kern, D. Yaden, H. Schwartz, S. Giorgi, G. Park, C. Hagan, V. Tobolsky, L. Smith, A. Buffone, J. Iwry, M. Seligman, L. Ungar, Closed and open vocabulary approaches to text analysis: A review, quantitative comparison, and recommendations, Psychol. Methods 26 (4) (2020) 398–427.

[29] D.E. Rumelhart, J.L. McClelland, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations, MIT Press, 1986.

[30] S. Ioffe, C. Szegedy, Batch normalization: accelerating deep network training by reducing internal covariate shift, in: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, ICML '15, JMLR.org, 2015, pp. 448–456.

[31] M. Schuster, K.K. Paliwal, Bidirectional recurrent neural networks, IEEE Trans. Signal Process. 45 (1997) 2673–2681.

[32] D.H. Wolpert, Stacked generalization, Neural Netw. 5 (2) (1992) 241–259.

[33] N. El-Naggar, P. Madhyastha, T. Weyde, Exploring the long-term generalization of counting behavior in RNNs, 2022, arXiv:2211.16429.

[34] A. Géron, Hands-On Machine Learning with Scikit-Learn and Tensorflow: Concepts, Tools, and Techniques to Build Intelligent Systems, third ed., O'Reilly Media, 2017.

[35] Y. Nesterov, A method for solving the convex programming problem with convergence rate $O(1/k^2)$, Proc. USSR Acad. Sci. 269 (1983) 543–547.

[36] D. Kingma, J. Ba, Adam: A method for stochastic optimization, in: International Conference on Learning Representations, ICLR, San Diego, CA, USA, 2015, pp. 1–13.

[37] A.C. Davison, D.V. Hinkley, Boostrap methods and their applications, Cambridge University Press, New York, 1997.

[38] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation., in: R.P. Gabriel, D.F. Bacon, C.V. Lopes, S. Guy L. Jr. (Eds.), OOPSLA, ACM, 2007, pp. 57–76.

[39] G. Rodola, Psutil, cross-platform lib for process and system monitoring in python, 2024, https://github.com/giampaolo/psutil.

[40] J. Choi, GPUstat, just less than nvidia-smi?, 2024, https://github.com/wookayin/gpustat.

[41] M. Grandini, E. Bagli, G. Visani, Metrics for multi-class classification: an overview, 2020, arXiv:2008.05756.

[42] L. van der Maaten, G.E. Hinton, Visualizing high-dimensional data using t-SNE, J. Mach. Learn. Res. 9 (2008) 2579–2605.

[43] F. Ortin, O. Rodriguez-Prieto, PLangRec: deep learning model to predict the programming language from a single line of code (support material website), 2024, https://www.reflection.uniovi.es/bigcode/download/2024/plangrec.

[44] J. Zhang, J. Zhang, S. Ghosh, D. Li, S. Tasci, L.P. Heck, H. Zhang, C.J. Kuo, Class-incremental learning via deep model consolidation, in: IEEE Winter Conference on Applications of Computer Vision, WACV 2020, Snowmass Village, CO, USA, March 1-5, 2020, IEEE, 2020, pp. 1120–1129.

[45] J. Kirkpatrick, R. Pascanu, N.C. Rabinowitz, J. Veness, G. Desjardins, A.A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, R. Hadsell, Overcoming catastrophic forgetting in neural networks., 2016, CoRR arXiv:1612.00796.

[46] F. Ortin, PLangRec: character-level deep-learning model to recognize the programming language of source code, 2024, https://github.com/ComputationalReflection/PLangRec.

[47] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: I. Guyon, U.V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 30, Curran Associates, Inc., 2017, pp. 6000–6010.

Dr. Oscar Rodriguez-Prieto is an assistant lecturer at the Computer Science Department of the University of Oviedo. He has a B.Sc. in Computer Science from the University of Oviedo, an M.Sc. in Artificial Intelligence from the National Distance Education University (UNED). In 2020, he was awarded his Ph.D. with a thesis titled "Big Code infrastructure for building tools to improve software development". He is a member of the Computational Reflection research group, and he stayed as a research intern at Cambridge University (2019). His main research interests are natural language processing, programming languages, and artificial intelligence. rodriguezoscar@uniovi.es

Mr. Alejandro Pato is a M.Sc. student at the Computer Science Department of the University of Oviedo. He has a B.Sc. in Software Engineering (Hons). His final degree project (4th year) is titled "Programming Language Identification from Source Code" classified as a research project, awarded with the highest grade. Alejandro's research interests include natural language processing, big data, and mining software repositories. uo214630@uniovi.es

Prof. Francisco Ortin is a full professor at the Computer Science Department of the University of Oviedo, Spain. He is also an adjunct lecturer for the Munster Technological University, Ireland. He is the head of the Computational Reflection research group at the University of Oviedo. His main research interests include big data for source code (BigCode), programming languages, and deep learning. ortin@uniovi.es francisco.ortin@mtu.ie