



Comparison of Image-Based and Text-Based Source Code Classification Using Deep Learning

Elife Ozturk Kiyak¹ · Ayse Betul Cengiz¹ · Kokten Ulas Birant² · Derya Birant²

Received: 31 March 2020 / Accepted: 30 July 2020 / Published online: 14 August 2020
© Springer Nature Singapore Pte Ltd 2020

Abstract

Source code classification (SCC) is a task to assign codes into different categories according to a criterion such as according to their functionalities, programming languages or vulnerabilities. Many source code archives are organized according to the programming languages, and thereby, the desired code fragments can be easily accessed by searching within the archive. However, manually organizing source code archives by field experts is labor intensive and impractical because of the fast-growing available source codes. Therefore, this study proposes new convolutional neural network (CNN) architectures to build source code classifiers that automatically identify programming languages from source codes. This is the first study in which the performances of deep learning algorithms on programming language identification are compared on both image and text files. In this study, the experiments are performed on three source code datasets to identify eight programming languages, including C, C++, C#, Go, Python, Ruby, Rust, and Java. The comparative results indicate that although text-based SCC and image-based SCC approaches achieve very high (> 93.5%) and similar accuracies, text-based classification has significantly better performance in terms of execution time.

Keywords Source code classification · Software engineering · Programming languages · Deep learning · Image classification · Text mining

Introduction

Until now, various programming languages have been developed such as C, C++, C#, Java, and Python, and used for many software engineering projects. The source codes

written in different languages have been continuously pushing into active repositories such as GitHub, SourceForge, and Bitbucket. With the increase of open-source programming environments in recent years, the number of users who benefit from these environments is also growing. They can add their own codes written in different programming languages, or easily access the ready-written codes and make some changes on them. There is a significant increase in the use of online coding platforms such as CodeForces, and Google Colab. Thus, a substantial volume of source codes has become accessible on many online platforms. In addition to the online platforms mentioned so far, which contain textual information-based sources, a significant amount of platform contains visual information (video or image), such as YouTube or online course platforms such as Udemy, and Coursera.

All these archives or online platforms categorize source code files according to the programming languages and when a source code is searched, these categories are checked. Nevertheless, manually organizing code repositories and labeling texts, images and videos are non-trivial tasks since they are large and grow rapidly. Therefore, some

This article is part of the topical collection “Deep learning approaches for data analysis: A practical perspective” guest edited by D. Jude Hemanth, Lipo Wang and Anastasia Angelopoulou.

✉ Derya Birant
derya@cs.deu.edu.tr

Elife Ozturk Kiyak
elif.ozturk@ceng.deu.edu.tr

Ayse Betul Cengiz
ayse.simsek@ceng.deu.edu.tr

Kokten Ulas Birant
ulas@cs.deu.edu.tr

¹ The Graduate School of Natural and Applied Sciences, Dokuz Eylul University, 35390 Izmir, Turkey

² Department of Computer Engineering, Dokuz Eylul University, 35390 Izmir, Turkey

automatic source code classification methods have been developed based on text classification [1–4]. In these studies, the source codes are considered as text, and classification methods are applied with the help of natural language processing (NLP) techniques. Although many studies have focused on the source code text data, only several studies [5, 6] considered identifying programming languages from the source codes embedded in the videos, due to the difficulty of extracting the codes from images. These studies used neural network-based methods with the popularity of deep learning on image classification.

The main contributions of this study are twofold. (i) Current studies focus on source code classification (SCC) using either text files or image files. The comparison of text-based SCC and image-based SCC in terms of accuracy is yet to be explored. This is the first study that investigates the impact of the source code data representation (text or image) on programming language identification. (ii) This study proposed two new convolutional neural network (CNN) architectures to build source code classifiers that identify programming languages from textual code files and visual image files separately. We designed successful CNN models to produce considerable results in the field of source code classification.

In the experimental studies, the effectiveness of the proposed CNN models was demonstrated on three source code datasets to identify eight programming languages, including C, C++, C#, Go, Python, Ruby, Rust, and Java. The comparative results indicate that although text-based SCC and image-based SCC approaches have similar accuracies, text-based classification has significantly better performance in terms of execution time.

In this study, two terms, source code classification (SCC) and programming language identification (PLI), are used interchangeably since both of them are currently used by the researches in the literature. Our study is important since code classification according to its programming language is an important step in archiving and reusing the source code.

The remainder of this study is structured as follows. Section 2 gives an overview of previous works related to source code classification. Section 3 explains the methods and algorithms used in this study. Section 4 presents experimental results and discusses the performances of text-based and image-based SCC. Finally, Sect. 5 provides some concluding remarks and possible future directions.

Related Work

Various studies have been conducted to analyze source codes such as generating pseudocode from source code [7], identifying topics in source codes [8], source code vulnerability analysis [9], code suggestion [10], classifying source codes according to their functionalities [11], clustering of source

codes [12], code review [13], source code author identification [14] and summarization of code fragments [15]. Although all these studies used source codes as training dataset for learning task, they did not focus on identifying the languages of source codes.

In the literature, various studies have been conducted for source code classification (SCC), as shown in Table 1. As an example, Alvares et al. [16] used lexical analysis, scoring strategies and genetic algorithm (GA) to classify source codes. In the lexical analysis, they extracted keywords as tokens and used as evidence in scoring by calculating the probability of given code files to determine which programming language belongs to. The purpose of the text classification is to assign the documents into one of the predefined classes. Since programming languages have specific features and keywords as a human language, text classification techniques can be used as a solution. Therefore, Gilda [3] proposed a CNN model to classify source code by using lexical analysis. First, he preprocessed the source code data and then applied tokenization to extract features. Another classification model was proposed by Alrashedy et al. [17] to identify the programming languages of code fragments written in twenty-one different languages. First, they converted code snippets into numerical feature vectors and then applied the multinomial naive Bayes (MNB) method, which is the modified form of traditional naive Bayes (NB) specially used for text documents. The authors have mentioned that identifying the language from the code snippets is far more challenging than from code files; therefore, their accuracy values are lower than the previous studies. In 2020, Alrashedy et al. [1] predicted the programming languages from both code snippets and body of questions using random forest (RF) and XGBoost.

Since the source code classification problem syntactically depends on the sequence of text appearance, some researchers used a Recurrent Neural Network (RNN) based on Long Short-Term Memory (LSTM) with word embeddings. For instance, Reyes et al. [18] automatically extracted the feature vector from a preprocessed source code input and trained the RNN by taking into account the order for the text sequences. Dam and Zaytsev [19] utilized statistical language models such as n -grams and skip grams in natural language processing (NLP) for programming language identification. Baquero et al. [4] proposed a model to predict the programming language from both comment text data and code snippets of Stack Overflow questions. They used Word2Vec for text feature extraction and n -gram for source code feature extraction. They built two classifiers using the support vector machine (SVM) method, one using textual feature and another using source code features.

It is clear from Table 1 that natural language techniques have been generally applied to identify programming languages from source code text files [2, 20, 21]. However,

Table 1 Comparison of our study with the previous source code classification studies

Publication	Year	Method	Features	#Languages	#Files	Text	Image
Alrashedy et al. [1]	2020	RF, XGBoost	Textual information features (functions, libraries) and code snippet features (identifiers/ keywords)	21	252,000	✓	X
Zhao et al. [6]	2019	CNN	Features extracted by the convolutional layers	2	50	X	✓
Ott et al. [5]	2018	CNN (VGG)	Syntactic and contextual features	4	19,200	X	✓
Alrashedy et al. [17]	2018	MNB	Code snippets converted into numerical feature vectors	21	252,000	✓	X
Alahmadi et al. [22]	2018	CNN	Features extracted by the convolutional layers	3	23,576	X	✓
Gilda [3]	2017	CNN	Lexicalized tokens which are represented as the matrix of word embeddings	60	118,324	✓	X
Zevin and Holzem [2]	2017	Maximum Entropy, NB	<i>n</i> -grams of sequences of lexicalized tokens	29	293,319	✓	X
Baquero et al. [4]	2017	SVM	Word2Vec for text feature extraction and <i>n</i> -gram for source code feature extraction	18	18,000	✓	X
Van Dam and Zaytsev [19]	2016	MNB	<i>n</i> -grams sequences	20	14,000	✓	X
Reyes et al. [18]	2016	LSTM - RNN	Word embedding vector obtained from tokens	10	756 MB	✓	X
Khasnabish et al. [20]	2014	MNB, NB	The number of key tokens	10	19,647	✓	X
Alvares et al. [16]	2014	GA	Keywords, which are filtered by an evolutionary process	8	1,700	✓	X
Klein et al. [21]	2011	A statistical-based learning algorithm	Statistical features, such as the percentage of brackets, the percentage of the most common language keywords	25	41,000	✓	X
Our study		CNN	Word embedding vector obtained from tokens for text data and features extracted by the convolutional layers	8	40,000	✓	✓

a limited number of studies exist for image-based source code classification. As an example, Ott et al. [5] proposed a method to automatically identify code fragments in images and videos. They used the VGG (Visual Geometry Group) network to specify the absence or presence of code in video frames. Alahmadi et al. [22] proposed an approach using CNN to predict the location of source code within video frames. Zhao et al. [6] used CNN and image differencing to identify workflow actions from programming screencasts.

Today, numerous new neural network models have been presented with the popularization of deep learning. CNN is one of the most popular ones and currently widely used in many fields, such as face detection [23], speech and handwriting recognition [24] and object detection [25]. Besides these fields, CNN has been widely used in the NLP field for text recognition and generally achieved promising results. Iwasaki et al. [26] proposed a CNN model for transfer learning in NLP. Gimenez et al. [27] used the CNN technique to improve the performance in NLP for sentiment analysis. Unlike these studies, in this paper, we used the CNN method for source code classification in the software engineering field.

From Table 1, it is possible to conclude that CNN is one of the most used techniques to classify source code by programming language. For this reason, we also selected this

technique because of its superior performance. In the studies mentioned above, either only text data or only image data have been studied for source code classification. In our study, we compared two different approaches (*image-based source code classification* and *text-based source code classification*) to determine the effective one.

Methodology

Source Code Classification

Source code classification (SCC) is the information extraction process from the code content, which aims to automatically organize software repositories. Regularly organizing code repositories is beneficial to maintenance, developer cooperation, and reusing source codes. Without a suitable organization, it is very difficult to find a code compatible with our demands during the search process. Manually assigning a source code into an appropriate group by field experts is time consuming and impractical because the sizes of software projects are generally quite large and increase rapidly. Therefore, in this study, a new CNN model is provided for automatically classifying the source codes based on their programming languages.

Some software development management systems analyze the source codes to optimize them. These analyses are strongly dependent on programming languages since each one has its own syntactic structure and characteristics. Here, source code classification is a necessary preprocessing stage to identify which analyses should be conducted. To accomplish this goal, this study aims to develop an effective classifier that will identify the programming languages from the source codes.

Source code management (SCM) systems have been currently utilized as an important resource for software engineering projects. The key feature of these SCM systems is code classification. This classification task is used for allowing SCMs to organize code components and perform specific analyses based on detected technologies. Some SCM systems utilize file extension to determine programming language [16]. For example, some rule-based classifiers identify C and C++ languages from “.h” header files. However, the main limitation of this file extension-based approach is the absence of lexical analysis for some programming languages. This approach assumes that all input source code files always have an extension, and the extension will be consistent. This limitation restricts the language identification process especially in the cases of code snippets and in-memory code. In addition, file extension does not exist for the source codes embedded in the videos. Therefore, this study proposes an effective method that can automatically classify a set of source codes based on their programming languages without requiring prior knowledge about file extensions.

In the source code classification, an instance x is a d -dimensional vector in the feature space X , such that

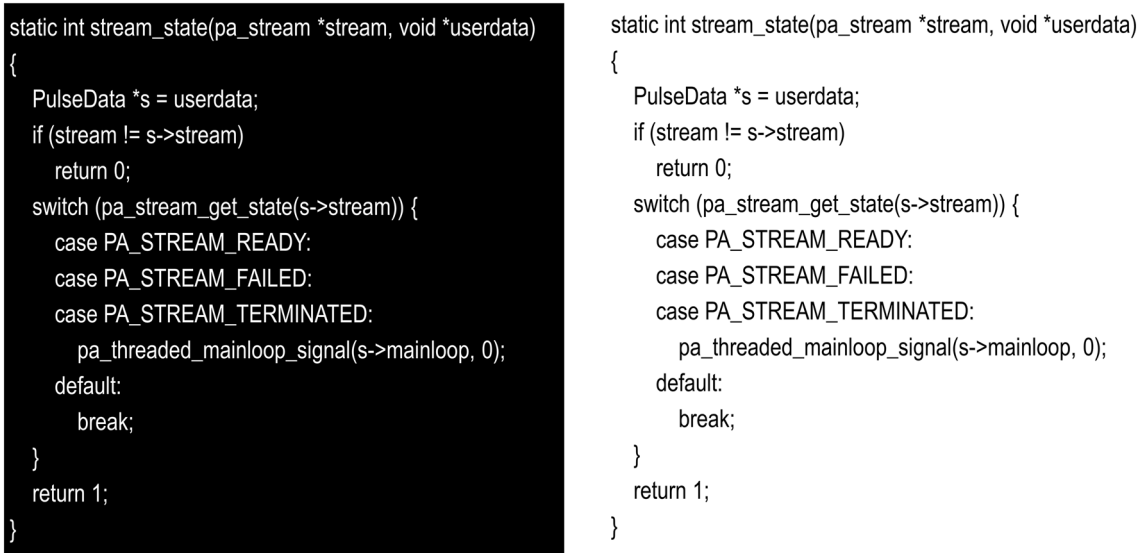
$X = \{x_1, x_2, \dots, x_d\}$ and $x_i \in X$. The dataset D includes a set of pairs of instances and their class labels, which is represented by $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$, where n is the number of training instances. The class attribute $Y = \{y_1, y_2, \dots, y_n\}$ has k class labels (actually programming languages), where $y \in \{c_1, c_2, \dots, c_k\}$. For instance, in a three-programming-language classification (Csharp, Java, Python), the class labels are $c_1 = \text{Csharp}$, $c_2 = \text{Java}$, and $c_3 = \text{Python}$. The goal is to build a classifier function $f : X \rightarrow Y$ that optimizes evaluation metric(s) and can correctly predict the programming languages of unseen source codes.

Image-Based and Text-Based Source Code Classification

In this study, we compared two different approaches, *image-based source code classification* and *text-based source code classification*, to determine the effective one. Figure 1 shows an example of source code image and text files from C programming language.

Text-Based Source Code Classification: Document classification is a well-established area with many different techniques and applications available. In essence, the source codes are high-dimensional and semi-structured textual data. Since text documents and source codes are similar enough, document classification techniques could be adapted for source code classification. Since CNN has been proven efficient in text classification, in this study we preferred this technique.

Image-Based Source Code Classification: Source code images are fed into a CNN, in which firstly abstract image features are extracted, and then a softmax classifier is trained



```
static int stream_state(pa_stream *stream, void *userdata)
{
    PulseData *s = userdata;
    if (stream != s->stream)
        return 0;
    switch (pa_stream_get_state(s->stream)) {
        case PA_STREAM_READY:
        case PA_STREAM_FAILED:
        case PA_STREAM_TERMINATED:
            pa_threaded_mainloop_signal(s->mainloop, 0);
        default:
            break;
    }
    return 1;
}
```

```
static int stream_state(pa_stream *stream, void *userdata)
{
    PulseData *s = userdata;
    if (stream != s->stream)
        return 0;
    switch (pa_stream_get_state(s->stream)) {
        case PA_STREAM_READY:
        case PA_STREAM_FAILED:
        case PA_STREAM_TERMINATED:
            pa_threaded_mainloop_signal(s->mainloop, 0);
        default:
            break;
    }
    return 1;
}
```

Fig. 1 Sample C source code image file and text file

based on the extracted image features to predict the probability of each class. In this solution, manual feature engineering is not required since the CNN model extracts features while training itself.

In this study, source code text files and image files were classified using convolutional neural networks (CNN) and the results were comparatively examined.

System Architecture

In this study, two different approaches are compared to identify the programming languages: *image-based source code classification* and *text-based source code classification*. In the first approach, source code image files are used as input for image classification; while, in the second approach source code text files are examined for text classification. Text data need a preprocessing step to transform text elements into numeric features to be used as input for deep learning algorithms. As can be seen in Fig. 2, 1D CNN is applied to text data after the preprocessing step; whereas, 2D CNN is applied to image data without any preprocessing step. Kernel moves on one dimension in 1D CNN; while, kernel slides along two dimensions in 2D CNN.

Currently, CNNs have been generally used in image classification because of its superior performance in recognizing entities, such as faces, traffic signs, and handwritten characters. This process is usually implemented via analysis over pixels of an image. Recently, CNNs have also been utilized in natural language processing tasks. In this case, the text data are transformed into feature vectors by applying several algorithms as explained below.

The *preprocessing* step is required for text data, in which the text is vectorized via tokenization and embedding into feature dimensions. *Tokenization* is the splitting of text into individual words, symbols or other elements called tokens according to some properties such as space, comma, brackets, and adding these parts into an array. *Embedding* creates vectors consisting of real numbers that represent tokens created in the previous layer. Word embedding is to represent individual words as

real-numbered vectors in a vector space, where words that have the same meaning have a similar representation. Each word is mapped to a vector, and the vectors are learned by a deep neural network model during the training phase.

CNNs apply a few distinct types of layers, including the convolution layer, padding layer, pooling layer, fully connected layer, dropout layer, and output layer, as explained below.

Convolution Layer is the core layer of a deep neural network to extract features from data. It applies some filters to the data to remove the low- and high-level features. For example, a filter may be capable of detecting edges in an image. The filter is moved on the data according to the stride size. For image data, the convolution dimension can be presented as Width \times Height \times Depth. Depth shows the channel number of the image; which is 3 for RGB and 1 for a gray-scale image. The output of a convolution layer is computed as:

$$\text{Output : width} = \frac{W - F_w + 2P}{S_w} + 1, \quad (1)$$

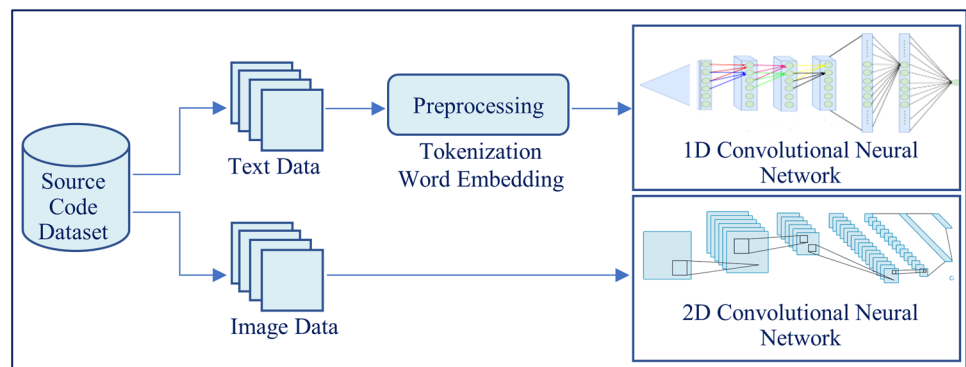
$$\text{Output : height} = \frac{W - F_h + 2P}{S_h} + 1, \quad (2)$$

where W is the width of image, F_w and F_h refer to width and height of filter size, respectively, S is stride size, and P is padding number.

The *padding layer* is a process of adding layers of zeros to guarantee that the outer boundaries of the input layer do not lose its features when the convolution operation is applied.

The *pooling layer* is frequently added between convolution layers in a network. It is used to reduce the dimensionality to decrease computational complexity and to improve the generalization ability and prevent overfitting at the same time. The most popular pooling operations are average pooling and max pooling. While average pooling returns the average of all values from the image covered by the kernel, max pooling gives the maximum value.

Fig. 2 The general architecture of the proposed approach



A *fully connected layer* combines convolutional layer outputs into a one-dimensional feature vector. It has a number of neurons and each neuron in one layer is connected to all the neurons in the next layer.

The *dropout layer* drops out some neural network units according to a certain probability to provide better efficiency by reducing overfitting. It has been observed that ignoring some nodes below a certain threshold value in fully connected layers increases performance.

The *output layer* is the final layer of the network and performs the classification task by assigning probabilities to each class. This final layer of the network has one output neuron per class label. In this layer, generally, *softmax* function is used, which is calculated as:

$$\text{softmax}(z)_j = \exp^{z_j} / \sum_{k=1}^K \exp^{z_k}, \quad (3)$$

where z_j refers to j^{th} node value, for $j = 1, 2, \dots, K$ and K is the number of class categories. The softmax function converts each element to a probability, in the range of 0–1 and the sum should be 1.

The Proposed CNN Architectures

In this study, for image-based source code classification, we designed a CNN architecture that consists of many layers. The structure of the proposed CNN model is presented in Fig. 3. The network is composed of several convolution layers + RELU, average pooling and max pooling layers, fully connected layer, and output layer (softmax). After each

max pooling, the result of two convolution layers is added to the front layer with skip connections or shortcuts, which is called a residual block. At the last, there is one average pooling, fully connected and output (softmax) layer.

The *Rectified Linear Unit* (ReLU) is one of the most common activation functions used to train deep neural networks. ReLU takes a value in the range $(0, +\infty)$ and it is defined as the maximum of zero and the given value as follows:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}, \quad (4)$$

ReLU has many advantages over other activation functions such as sigmoid and hyperbolic tangent (tanh) [28]. First, ReLU can be implemented by simple thresholding at zero, while the sigmoid/tanh functions require expensive operations like exponentials. Second, ReLU is not as severely affected by the vanishing gradient problem as other activation functions. Third, ReLU is known to significantly reduce the training time since the consequent sparse representations provide remarkable performance gains in training. Due to these advantages, in this study, we used ReLU as a non-linear and non-saturating activation function.

For text-based source code classification, a shallower CNN model is preferred as shown in Fig. 4. The network consists of three convolution layers and two fully connected layers. The proposed model starts with a tokenization task to convert input source code text into word vectors consisting of a sequence of integers. After that, zero-padding is applied to equalize the lengths of word vectors. After that, word embedding is applied to obtain word similarities. In the next step, convolution layers are applied with max pooling, and

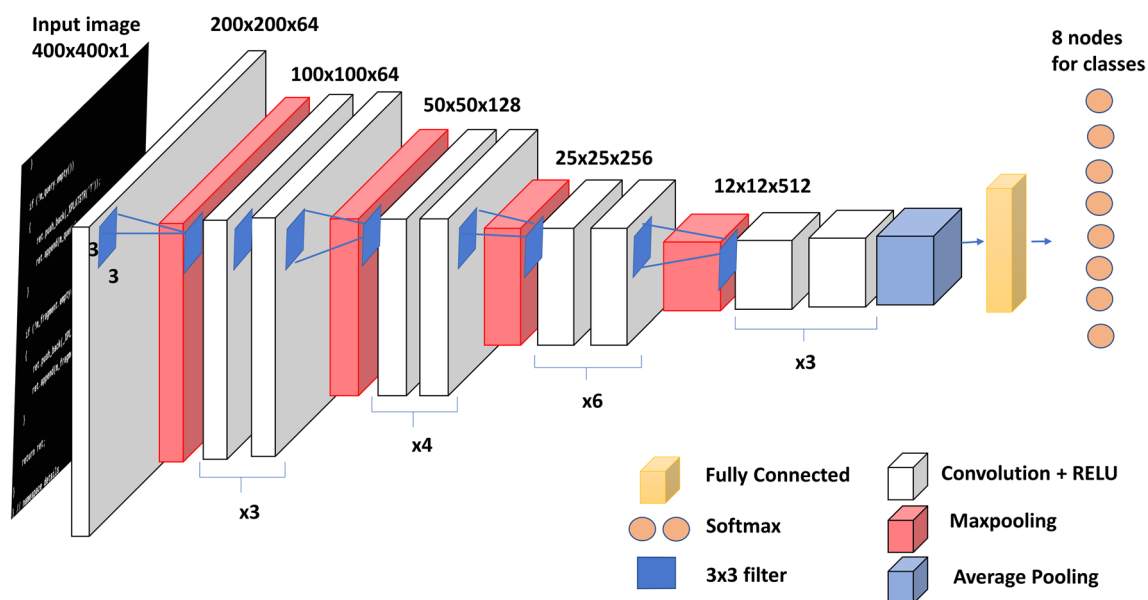


Fig. 3 CNN architecture used for image-based source code classification

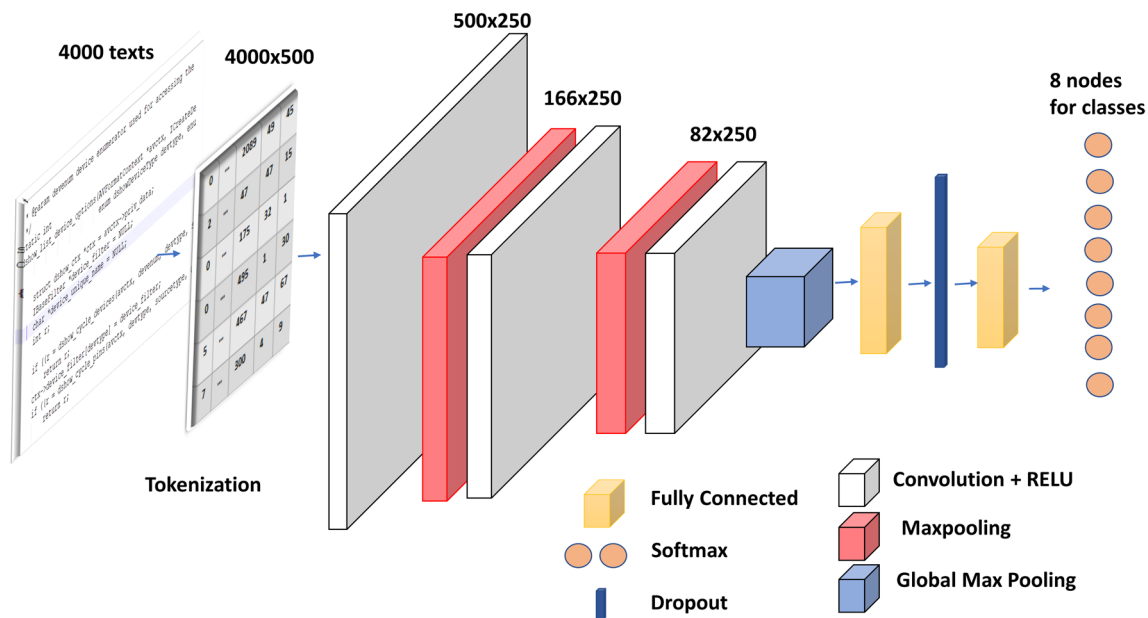


Fig. 4 CNN architecture used for text-based source code classification

Text Input:

```
if (index == 1) { printf("setting"); }
```

Tokenization:

```
['if', '(', 'index', '==', ')', '{', 'printf', '}']
```

Conversion:

```
[9, 2, 149, 38, 1196, 1756, 44, 7]
```

Padding:

```
[0, 0, 0, 0, 9, 2, 149, 38, 1196, 1756, 44, 7]
```

Fig. 5 Example of text data preprocessing step

finally, two fully connected layers and one dropout layer are designed to produce an output.

Data Preprocessing

The text-based source code classification requires a data preprocessing step to transform the text into feature vectors, as shown in Fig. 5. First, tokens within a code block are extracted by removing some unnecessary elements such as numbers and words between two apostrophes. Then, the text is separated into unique tokens using Keras text processing function. After that, these tokens are converted into integers

and then the padding layer is applied to equalize the size of text arrays. Finally, the embedding process is applied to obtain the optimal mapping for each unique word.

Experimental Studies

In the experimental studies, we compared two different approaches, *image-based source code classification* and *text-based source code classification*, to determine the effective one. Experiments were performed on Google Colab service using Jupyter Notebook since it provides free K80 GPU. PyTorch open-source machine learning framework and Fastai Python library were preferred for image classification, and Keras Python library is selected for text classification.

For the training process, the optimal input parameter values were determined as follows: batch size was specified as 64, epoch number was set to 8, dropout was applied with a probability of 20%, and filter size was initialized by 3×3 for image data and 3 for text data. The hold-out validation technique was used by splitting data as 80% and 20% for training and testing, respectively.

In this study, the results were evaluated according to accuracy values. *Accuracy* is the most commonly used measure for classification performance, which is the proportion of correctly classified cases to the total number of cases. In other words, accuracy is calculated as the ratio of correctly classified examples (TP, TN) with the total examples (TP, TN, FN, FP) as follows: $\text{Accuracy} = (\text{TP} + \text{TN}) / (\text{TP} + \text{FP} + \text{TN} + \text{FN})$, where true positive (TP) is the number of correctly classified positive

examples, true negative (TN) is the number of correctly predicted negative examples, false negative (FN) is the number of incorrectly predicted positive examples, and false positives (FP) is the number of examples incorrectly classified. It is a proper measure for our study since the dataset is balanced at class level (i.e., having the same number of examples per class).

Dataset Description

In this study, three different datasets were used for both image classification and text classification, each containing 8 different programming languages: C, C++, C#, Go, Python, Ruby, Rust, and Java. They were obtained from the codes in the master branches of the most popular titles in these 8 programming languages in GitHub [29]. The codes were downloaded from master branches, and each was manually selected and then assigned to one bundle per programming language. Therefore, we have one bundle source code file for each programming language. Each image and text data files were created from these bundles.

- *Dataset-1* includes 17-line random samples from each of the bundles, which were rendered into simple white-text-on-black-background images. It contains 500 images for each programming language, and totally, there are 4000 images. The size of each image is 224×224. In addition to the image files, text files, which are the same as the image files, were generated. Finally, 4000 image files and 4000 text files were created for Dataset-1.
- *Dataset-2* was obtained in the same way as Dataset-1 and includes 1000 image files and 1000 text files for each programming language, and totally, 8000 images and 8000 text files exist. The images are created in the form of white code on black background.
- *Dataset-3* contains 40,000 images and 40,000 text files (5,000 items per programming language).

In the data preprocessing step, the phrases of each text file were converted to tokens. For example, for the Dataset-1, a matrix with 4000 rows and 500 columns was obtained from 4000 text documents after applying zero-padding. After obtaining embedding vectors, we built 1D convolutional neural network with a filter of size 3, which is ending with a softmax output over 8 categories.

Table 2 Dataset sizes

	Training set	Test set	Total
Dataset-1	3200	800	4000
Dataset-2	6400	1600	8000
Dataset-3	32,000	8000	40,000

Table 2 gives information about data partitioning. The hold-out validation technique was used by splitting each dataset as 80% and 20% for training and testing, respectively. For example, 3200 out of the 4000 image and text files in the Dataset-1 were used as a training set and 800 ones as a test set. While 6400 source code files in the Dataset-2 were utilized as a training set, 1600 files were used as a test set. For the Dataset-3, 32,000 and 8000 files were evaluated as a training set and test set, respectively.

Results and Evaluation

In the image-based source code classification, we trained the network with different learning rates with the purpose of choosing the optimal one that has resulted in the smallest training loss. As a result of this process, based on the validated model, the ideal learning rate value was found and the network was retrained with this value. The changes in the loss value depending on different learning rates are shown in Fig. 6. The optimal learning rate was determined in the range of $1e-1$ – $1e-6$.

Table 3 shows example error rates and accuracy results obtained when determining the optimal learning rate. As the epoch increases, the accuracy increases significantly to a maximum value and error rate decreases to a minimum value. However, when the epoch exceeds a certain value, the accuracy begins to decrease since the overfitting. According to Table 3, the best accuracy value (93%) was achieved

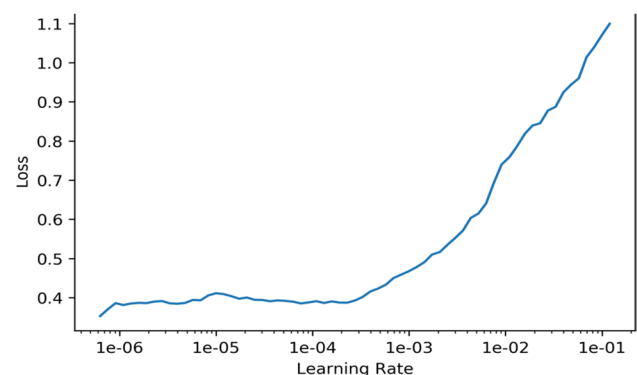


Fig. 6 Learning rate graph from the validated model

Table 3 Sample error rates and accuracy results obtained during model validation

Epoch	Error rate	Accuracy
0	0.677500	0.322500
1	0.630000	0.370000
2	0.170000	0.830000
3	0.116.250	0.883750
4	0.070000	0.930000
5	0.078750	0.921250

in epoch 4. For the validation data, the error rate converges to approximately 0.07. This model validation process was repeated with different learning rate parameters to determine the best one.

Table 4 shows the comparison of image-based source code classification and text-based source code classification for each dataset in terms of accuracy, train loss, validation loss, and error rate. From these results, we can conclude that the models built by image and text source code files achieve high accuracies (greater than 93.5% accuracy) for all datasets. When the data size is small (i.e., Dataset-1 and Dataset-2), accuracy for programming language identification on text data is higher than the accuracy obtained using image data. For example, text-based classification performed 96.37% accuracy for the Dataset-1; while, image-based classification achieved 93.5% accuracy. When the amount of data increases in terms of size, classifiers get better results. For example, the classification accuracy was 97.31% for the Dataset-2, which includes 8000 source code images; whereas, the accuracy was 99.38% when the Dataset-3 was used, containing 40,000 images.

Although accuracy was higher than 93.5% for both text data and image data, when we examined in terms of time, text classification reached the result in a shorter time as shown in Fig. 7. For example, for the Dataset-3, which contains 40,000 source codes in total, the training process took 9 minutes for text classification; while in the image classification, it went up to 40 min. However, pre-processing time is quite high in text classification, especially for big datasets.

A confusion matrix is a square matrix that represents information about the predicted versus the actual classes performed by a classifier on a test set. Each entry M_{ij} in a confusion matrix M indicates the number of times an example belonging to the class i is estimated by the classifier as class j .

Figure 8 shows the confusion matrix for both image-based and text-based classifiers. It can be concluded from all these results that by using image data or text data, the classifiers can usually distinguish the programming languages very well. According to the results, it is seen that C, C++, and Csharp are the most confused programming languages in both image-based classification and text-based

Table 4 Comparison of image-based and text-based source code classifications for each dataset in terms of accuracy

Epoch	Image-based classification				Text-based classification			
	Train loss	Val. loss	Err. rate	Acc.	Train loss	Val. loss	Train acc.	Acc.
Dataset-1								
1	0.0186	0.4658	0.1375	0.8625	1.3056	0.5897	0.5134	0.7775
2	0.4218	2.3822	0.5038	0.4963	0.1584	0.2098	0.9478	0.9237
3	0.3847	0.5501	0.1675	0.8325	0.0289	0.1292	0.9931	0.9575
4	0.2620	0.7355	0.1750	0.8250	0.0122	0.1217	0.9972	0.9625
5	0.1378	0.497	0.1238	0.8763	0.0066	0.1410	0.9991	0.9587
6	0.0656	0.2626	0.0713	0.9288	0.0040	0.1267	0.9988	0.9625
7	0.0329	0.2611	0.0675	0.9325	0.0034	0.1315	0.9991	0.9637
8	0.0182	0.2577	0.0650	0.9350	0.0027	0.1312	0.9991	0.9625
Dataset-2								
1	0.6508	2.5726	0.5934	0.4066	0.8061	0.2307	0.705	0.9256
2	0.4632	1.2681	0.4310	0.5690	0.0781	0.1065	0.9766	0.9675
3	0.3017	1.4792	0.3710	0.6290	0.0174	0.0777	0.9948	0.9769
4	0.1783	0.7511	0.2242	0.7758	0.0068	0.0768	0.9988	0.9762
5	0.1000	0.8185	0.2698	0.7302	0.0039	0.0774	0.9992	0.9812
6	0.0502	0.1302	0.0412	0.9588	0.0034	0.0677	0.9992	0.9794
7	0.0227	0.0858	0.0281	0.9719	0.0027	0.0723	0.9989	0.9769
8	0.0122	0.0846	0.0269	0.9731	0.0023	0.0797	0.9992	0.9781
Dataset-3								
1	0.2458	3.1433	0.6635	0.3365	0.2525	0.0706	0.9112	0.9772
2	0.1687	0.7341	0.223	0.7770	0.0358	0.0574	0.9892	0.9848
3	0.1067	1.2060	0.3415	0.6585	0.0227	0.0582	0.9933	0.9856
4	0.0608	0.2495	0.0813	0.9188	0.0294	0.0503	0.9919	0.987
5	0.0341	0.1030	0.0323	0.9678	0.0209	0.079	0.9944	0.9851
6	0.0225	0.0333	0.0098	0.9903	0.0197	0.1019	0.9946	0.9795
7	0.0064	0.0242	0.0064	0.9936	0.0214	0.0722	0.9946	0.9851
8	0.0041	0.0241	0.0063	0.9938	0.0111	0.0565	0.9969	0.9881

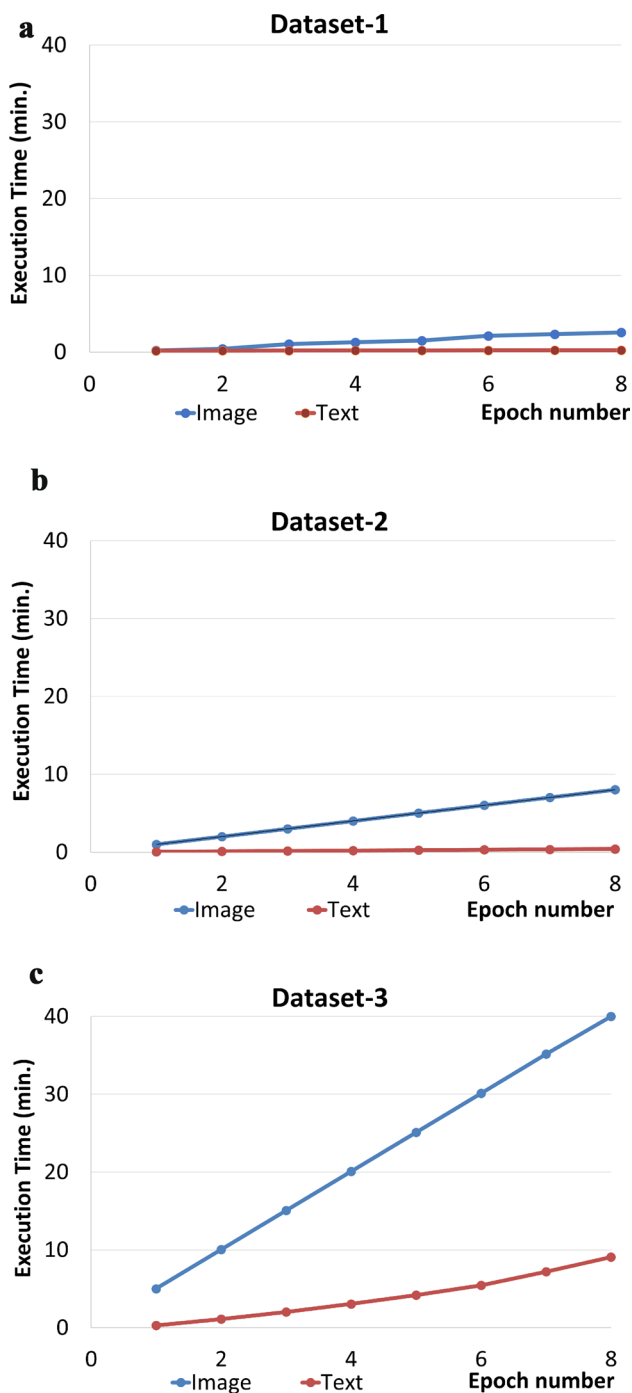


Fig. 7 Comparison of image-based and text-based source code classifications for each dataset in terms of execution time

classification. For instance; 10, 9 and 11 Csharp text files in Dataset-3 were predicted as C, Go and Java, respectively. Likewise, 4, 6 and 6 Csharp image files were estimated as C, C++ and Go, respectively. Furthermore, it is obvious that the generated classifiers correctly identified programming language Python at the most.

It can be concluded from the results that the proposed CNN models can be successfully utilized for building efficient source code classifiers.

Comparison with the Existing Studies

The previous studies in the SCC have utilized the different parts of the Github and Stack Overflow repositories or distinct environments to evaluate their methods. Table 5 shows the results of the existing text-based and image-based SCC works. It is clearly seen that the applied methods and datasets affect correctly classified source code files or code fragments. For example, Baquero et al. [4] extracted text features and source code features from the Stack Overflow posts, then used the SVM algorithm, and achieved the classification accuracy of 60.88% and 44.61%, respectively. Moreover, Reyes et al. [18] reached different accuracy values on various datasets, e.g., for the Rosetta Code dataset, for the Github repository dataset, and for the Lang dataset, the accuracy values of 80.22%, 66.46%, and 99.85% were obtained, respectively. While Gilda [3], Alvares et al. [16], and Khasnabish et al. [20] achieved the accuracy higher than 90% from the Github repositories; Klein et al. [21] only reached 48% accuracy. Alrashedy et al. [1] extracted code snippets from the online forums in the Stack Overflow repository and classified them with 75% accuracy, and so they claimed that classifying code snippets is a more challenging task than classifying source code files.

We compared the performance of our proposed model with the already existing Programming Languages Identification (PLI) tool [30]. The PLI tool that can predict the programming language of a given code snippet is available in Algorithmia, a marketplace for AI-based algorithms. PLI supports 21 programming languages, except the languages *go* and *rust*. Therefore, we removed these two programming languages from our dataset and evaluated the remaining six programming languages (*c*, *csharp*, *cpp*, *java*, *python*, *ruby*). The API calls of the PLI tool were used to predict the class of each source code file. It generates a percentage value for each language and then selects the language which has the highest probability score as the class label. According to the results, the PLI tool achieved an accuracy of 85% and the most confusing languages are *c* and *cpp*. Also, some of the *java* and *csharp* code files were classified as *scala* and *swift*, respectively. For the randomly selected small part of the same dataset, our model outperformed the PLI tool when predicting the classes.

Although we have mentioned text-based studies until now, there are a few image-based studies in SCC. We compared our models with the model presented in [29] using the same dataset (Dataset-1). While the accuracy values of 96.37% and 93.50% were achieved with the method

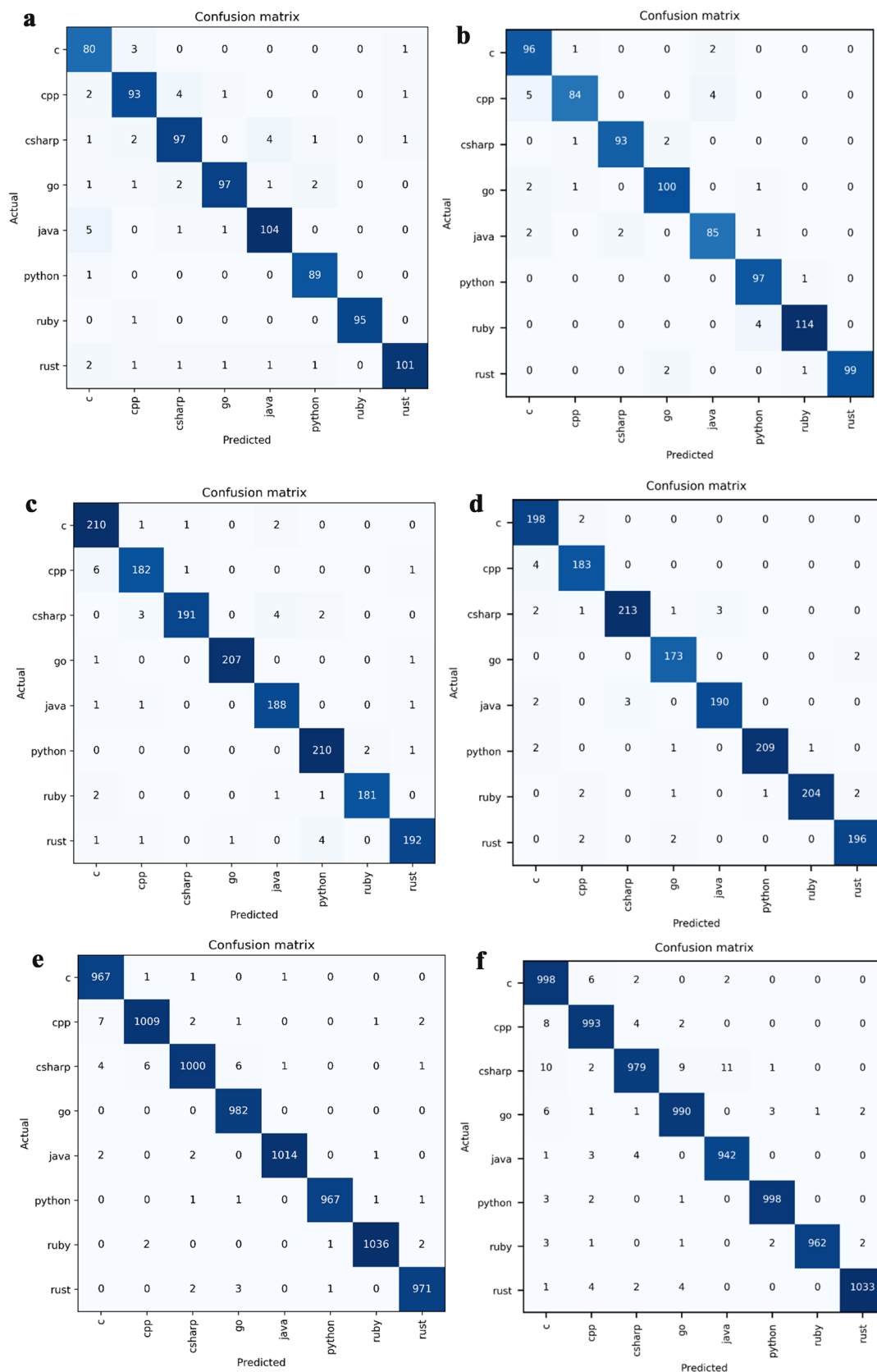


Fig. 8 Confusion matrices for different datasets

Table 5 Comparison with the existing SCC studies

Publication	Data source	Result (accuracy (%))
Text-based studies		
Alrashedy et al. [1]	Stack overflow	88.90
Gilda [3]	Github	97.00
Baquero et al. [4]	Stack overflow	Text → 60.88 Source-code → 44.61
Alvares et al. [16]	Github	96.50
Alrashedy et al. [17]	Stack overflow	75.00
Reyes et al. [18]	Rosetta code, github, lang	Rosetta code → 80.22 Github → 66.46 Lang → 99.85
Van Dam and Zaytsev [19]	Github	96.90
Khasnabish et al. [20]	Github	93.48
Klein et al. [21]	Github	48.00
Heres [30]	Github	85.00
Our study	Github	98.81
Image-based studies		
Ott et al. [5]	YouTube	85.60 - 98.60
Zhao et al. [6]	YouTube	88 ± 1.3
Alahmadi et al. [22]	YouTube	92.00
Laks [29]	Github	94.90
Our study	Github	99.38

put forward in our study, the accuracy value of 94.9% was obtained in the study [29].

Conclusion and Future Works

Programming communities have currently contributed an enormous amount of source code to the internet. Since the number of codes increases every day, human-based source code classification (SCC) has become more and more time consuming and even impractical. Therefore, this study proposes new convolutional neural network (CNN) architectures to build source code classifiers that automatically identify programming languages from source codes. This is the first study that the performances of deep learning algorithms on programming language identification are compared on two different representations of source codes, which are image files and text files.

In this study, the experiments performed on three source code datasets to identify eight programming languages, including C, C++, C#, Go, Python, Ruby, Rust, and Java. The experimental results show that the proposed CNN architectures can effectively classify source codes with very high accuracies (> 93.5%). The comparative results indicate that although text-based SCC and image-based SCC approaches achieve similar accuracies, text-based classification has significantly better performance in terms of execution time.

In the future, this study can be expanded by implementing the recurrent neural network technique for the text-based source code classification. In addition, other word

embedding techniques such as Glove, Word2Vec can be tried to determine the best one. Furthermore, video frames can be detected whether it contains source code and belongs to which programming language.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Alrashedy K, Dharmaretnam D, German DM, Srinivasan V, Gulliver TA. SCC++: predicting the programming language of questions and snippets of Stack Overflow. *J Syst Softw.* 2020;. <https://doi.org/10.1016/j.jss.2019.110505>.
2. Zevin S, Holzem C. Machine learning based source code classification using syntax oriented features 2017. arXiv preprint [arXiv:1703.07638](https://arxiv.org/abs/1703.07638).
3. Gilda S. Source code classification using neural networks. In: 14th International Joint Conference on Computer Science and Software Engineering (JCSSE), IEEE; 2017. p. 1–6.
4. Baquero JF, Camargo JE, Restrepo-Calle F, Aponte JH, González FA. Predicting the programming language: Extracting knowledge from stack overflow posts. In: Colombian Conference on Computing. Springer; 2017. p. 199–210.
5. Ott J, Atchison A, Harnack P, Bergh A, Linstead E. A deep learning approach to identifying source code in images and video. In: 15th International Conference on Mining Software Repositories (MSR), IEEE/ACM; 2018. p. 376–386.
6. Zhao D, Xing Z, Chen C, Xia X, Li G. ActionNet: vision-based workflow action recognition from programming screencasts. In:

- 41st International Conference on Software Engineering (ICSE), IEEE/ACM; 2019. p. 350–361.
7. Oda Y, Fudaba H, Neubig G, Hata H, Sakti S, Toda T, Nakamura S. Learning to generate pseudo-code from source code using statistical machine translation. In: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE/ACM; 2015. p. 574–584.
8. Kuhn A, Ducasse S, Gírba T. Semantic clustering: identifying topics in source code. *Inf Softw Technol.* 2007;. <https://doi.org/10.1016/j.infsof.2006.10.017>.
9. Darwish O, Maabreh M, Karajeh O, Alsinglawi B. Source codes classification using a modified instruction count pass. In: Workshops of the International Conference on Advanced Information Networking and Applications (WAINA), Springer; 2019. p. 897–906.
10. Nguyen AT, Nguyen TN. Graph-based statistical language model for code. In : 37th IEEE International Conference on Software Engineering (ICSE), IEEE/ACM; vol 1; 2015. p.858–868.
11. Phana AH, Chau PN, Nguyen ML, Bui LT. Automatically classifying source code using tree-based approaches. 2018;. <https://doi.org/10.1016/j.datak.2017.07.003>.
12. Wilson W, Muteteke JJ, Li L. Automatic clustering of source code using self-organizing maps. In: Proceedings of 19th Annual Conference of SAIS. 2016; p. 1–5.
13. Shi ST, Li M, Lo D, Thung F, Huo X. Automatic code review by learning the revision of source code. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol 33; 2019. p. 4910–4917.
14. Bandara U, Wijayarathna G. Source code author identification with unsupervised feature learning. *Pattern Recognit Lett.* 2013;. <https://doi.org/10.1016/j.patrec.2012.10.027>.
15. Ying AT, Robillard MP. Code fragment summarization. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. 2013; p. 655–658.
16. Alvares M, Marwala T, de Lima Neto FB. Application of computational intelligence for source code classification. In: Congress on Evolutionary Computation (CEC), IEEE; 2014. p. 895–902.
17. Alrashedy K, Dharmaretnam D, German DM, Srinivasan V, Gulliver TA. SCC: Automatic classification of code snippets 2018. arXiv preprint [arXiv:1809.07945](https://arxiv.org/abs/1809.07945)
18. Reyes J, Ramírez D, Paciello J. Automatic classification of source code archives by programming language: a deep learning approach. In: International Conference on Computational Science and Computational Intelligence (CSCI), IEEE; 2016. p. 514–519.
19. Dam V, Kennedy J, Zaytsev V. Software language identification with natural language classifiers. In: 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol 01; 2016. p. 624–628.
20. Khasnabish JN, Sodhi M, Deshmukh J, Srinivasaraghavan G. Detecting programming language from source code using bayesian learning techniques. In: International Workshop on Machine Learning and Data Mining in Pattern Recognition. Springer; 2014. p. 513–522.
21. Klein D, Muuray K, Weber S. Algorithmic programming language identification 2011. arXiv preprint [arXiv:1106.4064](https://arxiv.org/abs/1106.4064).
22. Alahmadi M, Hassel J, Parajuli B, Haiduc S, Kumar P. Accurately predicting the location of code fragments in programming video tutorials using deep learning. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE), 2018. p. 2–11.
23. Guo G, Zhang N. A survey on deep learning based face recognition. *Comput Vis Image Underst.* 2019;. <https://doi.org/10.1016/j.cviu.2019.102805>.
24. Pastor-Pellicer J, Castro-Bleda MJ, España-Boquera S, Zamora-Martínez F. Handwriting recognition by using deep learning to extract meaningful features. *AI Commun.* 2019;. <https://doi.org/10.3233/AIC-170562>.
25. Liu L, Ouyang W, Wang X, Fieguth P, Chen J, Liu X, Pietikainen M. Deep learning for generic object detection: a survey. *Int J Comput Vis.* 2020;. <https://doi.org/10.1007/s11263-019-01247-4>.
26. Iwasaki R, Hasegawa T, Mori N, Matsumoto K. Relaxation method of convolutional neural networks for natural language processing. In: International Symposium on Distributed Computing and Artificial Intelligence. Springer; 2018. p.188–195.
27. Gimenez M, Palanca J, Botti V. Semantic-based padding in convolutional neural networks for improving the performance in natural language processing. A case of study in sentiment analysis. *Neurocomputing.* 2020; <https://doi.org/10.1016/j.neucom.2019.08.096>.
28. Gao H, Lin S, Li C, Yang Y. Application of hyperspectral image classification based on overlap pooling. *Neural Process Lett.* 2019;49:1335–54.
29. Laks R. Image-based detection of programming languages. In: Github. 2018. <https://github.com/rivol/programming-language-detection>. Accessed 15 Nov 2019.
30. Heres D. Programming language identification tool. In: Algorithmia. 2016. <https://algorithmia.com/algorithms>. Accessed 8 July 2020.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.