The Journal of Systems & Software 193 (2022) 111454

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Towards using visual, semantic and structural features to improve code readability classification ☆

Qing Mi, Yiqun Hao, Liwei Ou, Wei Ma *

Beijing University of Technology, Beijing, China

ARTICLE INFO

Article history:
Received 2 December 2021
Received in revised form 16 June 2022
Accepted 19 July 2022
Available online 23 July 2022

Keywords:
Code readability classification
Code representation
Neural networks
Program comprehension
Software analysis

ABSTRACT

Context: Code readability, which correlates strongly with software quality, plays a critical role in software maintenance and evolvement. Although existing deep learning-based code readability models have reached a rather high classification accuracy, only structural features are utilized which inevitably limits their model performance.
Objective: To address this problem, we propose to extract readability-related features from visual, semantic, and structural aspects from source code in an attempt to further improve code readability classification.
Method: First, we convert a code snippet into a RGB matrix (for visual feature extraction), a token sequence (for semantic feature extraction) and a character matrix (for structural feature extraction). Then, we input them into a hybrid neural network that is composed of BERT, CNN, and BiLSTM for feature extraction. Finally, the extracted features are concatenated and input into a classifier to make a code readability classification.
Result: A series of experiments are conducted to evaluate our method. The results show that the average accuracy could reach 85.3%, which outperforms all existing models.
Conclusion: As an innovative work of extracting readability-related features automatically from visual, semantic, and structural aspects, our method is proved to be effective for the task of code readability classification.

1. Introduction

Maintenance is one of the most important phases in software lifecycle costing more than 70% of the overall expenditure (Boehm and Basili, 2001), whereas reading source code is identified as the most time-consuming part during maintenance (Raymond, 1991). Being a prerequisite of software maintenance, code with bad readability could largely slow down the entire maintenance process and even lead to costly large-scale refactoring (Xia et al., 2017; Aggarwal et al., 2002; Elshoff and Marcotty, 1982). In this case, an effective code readability classifier could continuously assist developers in monitoring the readability level of their source code to avoid time and cost overrun. As a survey suggests (Buse and Zimmermann, 2012), such a code readability classifier is desired by 90% developers and managers at Microsoft.

Code readability is defined as a metric measuring how intuitive source code is for developers to read (Buse and Weimer, 2010). Along with the definition, Buse and Weimer constructed the first machine learning-based code readability model considering a set of surface-level code features such as the number of identifiers. The proposed model was proved to perform better than a human on average with a classification accuracy of 76.5%. Later, Dorn proposed a more generalized classification model incorporating geometric, pattern-based, and linguistic code features and gained a better accuracy (Dorn, 2012). Furthermore, Scalabrino et al. improved Dorn's model by including textual (or lexical) features and achieved the highest accuracy of 81.8% (Scalabrino et al., 2016).

Though Scalabrino et al.'s model incorporates various aspects and achieves a rather high performance (Scalabrino et al., 2016), there are inevitable disadvantages for such handcrafted features. To be specific, handcrafted features are inefficient because manual feature engineering is a highly labor-intensive task that requires massive time and domain-specific knowledge. More importantly, handcrafted features are still not effective enough, which can only capture surface-level, or in other words, partial readability information. Therefore, using handcrafted features that are already engineered is not sufficient, but limits the further improvement of classification accuracy. As a result, deep learning-based models with effective code representation methods were put forward. Being able to automatically extract

☆ Editor: Martin Pinzger.

* Correspondence to: Faculty of Information Technology, Beijing University of Technology, Beijing, China.
E-mail addresses: miqing@bjut.edu.cn (Q. Mi), yiqun.hao@ucdconnect.ie (Y. Hao), liwei.ou@ucdconnect.ie (L. Ou), mawei@bjut.edu.cn (W. Ma).

https://doi.org/10.1016/j.jss.2022.111454
0164-1212/

in-depth readability features, deep learning-based models have shown outstanding efficiency and effectiveness. A typical deep learning-based model, namely DeepCRM, gained a state-of-the-art classification accuracy of 83.8% outperforming all machine learning-based models (Mi et al., 2018a). Nevertheless, existing deep learning-based models merely focus on structural aspects of source code, whereas other critical features, such as semantic (e.g., the meaning of identifiers and comments) and visual features (e.g., the on-screen representation of source code) are missed. The problem is likely to result in limited model performance. Therefore, we propose to apply visual, semantic, and structural representation methods together, and construct corresponding feature extraction networks to fully capture readability-related features. The extracted features of a given code snippet are then used to classify it into either a readable or unreadable class.

The major contributions of this paper are:

- We propose a novel code representation method to reserve visual, semantic, and structural information which are crucial for code readability classification.
- We construct a hybrid neural network based on BERT, CNN, and BiLSTM to effectively extract readability-related features from source code.
- A series of experiments are conducted to evaluate our method. The results confirm the validity of visual, semantic, and structural features we proposed, where our model can reach a state-of-the-art code readability classification accuracy of 85.3%, as well as 85.0% f-measure, 90.8% AUC, and 73.8% MCC.
- We publish all our data and source code of the model online to benefit future researchers.[1]

This paper is organized as follows: Section 2 introduces prior code readability classification researches, commonly used code representation methods, and BERT. Section 3 presents our model architecture in detail. Specifically, the code representation methods, feature extraction networks, and the classifier are illustrated. Section 4 describes our dataset, evaluation metrics, and research questions we proposed. In Section 5, the results of our experiments are demonstrated with respect to the aforementioned research questions. In Section 6, we discuss the disadvantages of using handcrafted features and some critical factors that constrain our model performance. Then, some threats to the validity of our results are outlined in Section 7. This paper is wrapped up in Section 8 where we draw a conclusion and point out the future work that could be applied to further improve our method.

2. Related work

This section is comprised of three parts. In the first part, prior code readability classification researches are introduced. Then we present classical code representation methods in the second part. BERT (i.e., Bidirectional Encoder Representations from Transformers) is illustrated in the last part.

2.1. Code readability classification

The very first code readability classification model was put forward by Buse and Weimer in 2008 (Buse and Weimer, 2010). Buse and Weimer first invited 120 human annotators to manually rank 100 code snippets by their readability level based on a five-point Likert scale (Likert, 1932) ranging from one (i.e., very unreadable) to five (i.e., very readable). The data gathered by Buse and Weimer could be treated as the first formal code readability dataset. Based on the dataset collected, they put forward a set of handcrafted features that possibly affect code readability including the average number of identifiers or number of blank lines. A few experiments were carried out on the Weka toolbox (Holmes et al., 1994) where machine learning algorithms were used to examine the correlation between each proposed feature and the code readability score respectively. Consequently, a model based on a set of validated features having high correlations with code readability was constructed. As a pioneering study in the code readability classification field, Buse and Weimer substantiated that code readability is measurable by concrete metrics which created a new era of code readability research. To further improve Buse's model, Dorn incorporated geometric, pattern-based, and linguistic features into model construction which gains a higher accuracy along with a better generalization ability (Dorn, 2012). Later, Scalabrino et al. (2016) further extended the work of Dorn by enriching previous metrics with seven textual features which measure the consistency between source code and comments, specificity of the identifier, etc. Benefited by more textual features, the model of Scalabrino et al. further improves the code readability classification accuracy to 81.8%.

However, manual feature engineering which is widely used in machine learning-based methods is too time-consuming. Besides, handcrafted features are usually troubled with personal biases and neglect of certain features which limit the model performance (Mi et al., 2018a). Therefore, a deep learning-based model, IncepCRM, which could automatically extract in-depth structural features without human intervention was put forward by Mi et al. in 2018 (Mi et al., 2018b) to overcome the limitation of manual feature engineering. IncepCRM applies a simple code representation method that converts each code snippet into a character matrix. After that, Mi et al. proposed a more complicated code representation method that extracts structural features from two more granularities apart from the character-level one adopted by IncepCRM (Mi et al., 2018a). To be specific, the three representation granularities are character-level representation which treats each character (e.g., a-z, A–Z, and 0–9) as a symbol, token-level representation which treats each token (e.g., keywords and operators) as a symbol, and node-level representation which treats each node in the Abstract Syntax Tree (e.g., declarations and definitions) as a symbol. By utilizing features extracted from the three granularities, DeepCRM (Mi et al., 2018a) reaches a state-of-the-art accuracy of 83.8%.

Nevertheless, Both manual feature extraction and deep learning-based feature extraction are insufficient to capture enough code readability features. For manual feature extraction, many in-depth readability features that are not quantifiable are missed. For instance, identifiers' naming formats which is identified as crucial for code comprehension (Binkley et al., 2009) are not captured. Besides, instead of using mathematical calculations including ratios between areas occupied by each pair of colors and average bandwidths of 2D DFT of color matrices to model the effects of syntax highlighting (Dorn, 2012), we propose a more generalized approach by using the colored figures themselves as the input of CNN, which could retain as much as possible the original visual information about the source code (i.e., the distribution of colored code elements). By virtue of the advanced deep neural networks, in-depth visual features and details such as patterns of color changes (frequent color changes may indicate low readability) and the density of code snippets (i.e., is the line very packed or sparse by having whitespaces between code elements) could be automatically discovered, learned, and extracted without human intervention. With respect to textual properties of source code that can help in characterizing its readability, we argue that our BERT-based approach can capture these high-level properties more comprehensively and address the semantic

[1] https://github.com/swy0601/Readability-Features

problem of polysemy and synonym better than the traditional feature engineering method. For instance, in order to analyze the consistency between identifiers and comments, Scalabrino et al., (2016) simply proposed to use a fixed set of words to identify synonyms, while our method could capture not only synonyms (e.g., car and automobile) but also the semantic similarity between words (e.g., king and queen) more accurately and quantitatively by calculating the cosine similarity score between the obtained word embeddings. Besides, Scalabrino et al. (2016) only considered the consistency between identifiers and comments, while our method can learn deeper context from source code and thus detect consistency in a wider range including comments, keywords, identifiers, literals and even operators. Ultimately, we are able to estimate the textual coherence of code snippets (i.e., the number of "concepts" implemented by a code snippet) in an easier and more generalized way than previous methods. As for existing deep learning-based models, valuable information encapsulated in visual and semantic features is largely omitted. Thus, we propose to use visual, semantic and structural code representation to complement the omission of aforementioned valuable readability features.

2.2. Code representation

As a precondition to utilize deep learning-based code analysis, source code has to be represented in the form of vectors to be acceptable for deep neural networks. Therefore a few code representation methods are proposed for various tasks. In this section, we introduce three classical code representation methods that are most commonly used in deep learning-based code analysis researches (Zhang et al., 2018).

The simplest and most intuitive code representation method is one-hot encoding. One-hot encoding uses vectors with the length of N to represent N different states. According to its characteristics, one-hot encoding is commonly applied in the field of code generation (Alexandru, 2016; Mou et al., 2015; Shu and Zhang, 2017). Typically, Recurrent Neural Networks (RNN) are utilized along with one-hot encoding. However, though one-hot encoding is simple to use, the spatial distribution of the encoded vector will become very sparse with the increase of states making it merely suitable for low-dimensional encoding tasks.

Bag of words model is another classic code representation method. Regardless of the context, bag of words model treats each word as a unit and encodes an n-words sentence as a vector with the length of n. According to its feature, bag of words model is widely used to predict programming bugs by calculating the spatial similarity between prior bug report files and the source code files to locate files that might contain bugs (Zhou et al., 2012; Lam et al., 2015). Besides, bag of words model is also applied in many other code-related areas, such as code completion (Raychev et al., 2014), code categorization (Nguyen and Nguyen, 2017), commit message generation (Jiang et al., 2017), etc.

Word embedding, such as GloVe (Pennington et al., 2014) and Word2Vec (Mikolov et al., 2013), represents each word using a d-dimensional semantic vector which allows vectors of synonyms close to each other. Word embedding is proven to be effective in several code analysis tasks including bug location (Ye et al., 2016), code migration (Gu et al., 2016; Nguyen et al., 2017), code completion (White et al., 2015), programming language processing (Mou et al., 2016), etc.

Specifically for the code readability classification which is highly sensitive to context, it is crucial to find out a proper code representation method to represent source code as well as retain as much of the original readability information as possible. Therefore, we propose the combination of three code representation methods illustrated in Section 3.1 including the use of word embedding to extract semantic features.

2.3. BERT

Proposed in 2018 (Devlin et al., 2018), BERT (i.e., Bidirectional Encoder Representations from Transformers) has gained great success in the field of NLP (Natural Language Processing). Different from traditional NLP models, BERT adopts a multi-layer transformer rather than RNN or CNN. Transformer is an encoder–decoder structure stacked by several encoder blocks and decoder blocks. However, unlike traditional transformer, BERT contains more encoder blocks to thoroughly capture context information. In addition, Bert applies attention mechanisms, which also allow a better context capture. As a result, BERT could perform effective semantics extraction and analysis.

BERT has been pre-trained on a very large-scale corpus that is huge enough for generalization. During training, the input is encoded into three vectors: a token embedding to separate each word, a segment embedding to separate each sentence, and a position embedding to record the word location. Then, BERT is trained using an unsupervised learning strategy on two tasks that do not need labeled data: masked language model and next sentence prediction. After Bert has been pre-trained, it is ready to be applied for NLP tasks. Considering that BERT has performed well in text classification and semantic recognition (Müller et al., 2020; de Vries et al., 2019; Tenney et al., 2019), we decided to take advantage of BERT in semantic feature extraction.

3. Model architecture

As shown in Fig. 1, our model mainly consists of three parts. First, we preprocess a code snippet into an RGB matrix, a token sequence, and a character matrix with respect to each representation method (see Section 3.1). Subsequently, we construct a hybrid model comprised of three feature extraction networks to extract features from visual, semantic, and structural aspects (see Section 3.2). Finally, features captured are input into a classifier which then gives a code readability prediction (see Section 3.3).

3.1. Code representation

A proper code representation method is essential for reserving code readability-related information and therefore determines the effectiveness of a classification model. Thus, to capture more readability features, we propose three different code representation methods and demonstrate them in this section.

3.1.1. Visual representation

Defined as how intuitive a snippet of code is for developers to understand (Buse and Weimer, 2010), code readability models should not ignore the effect of the visual perception of code which is one of the most intuitive readability features. For instance, code that has neat indentations and that is consistent in the size of code blocks or commented regions should provide a more readable visual framework. Besides, as Dorn emphasized, syntax highlighting also positively impacts code readability (Dorn, 2012). To capture those features, Dorn took the two-dimensional discrete Fourier transform (DFT) of the image formed by the colored characters and performed mathematical calculations to measure the relative noise or regularity (Dorn, 2012). Differently, we propose to use CNN to extract visual features. Thus, we propose to take actual screenshots of code snippets and convert them into RGB matrices where we could concern more code elements and capture more detailed color differences than Dorn's method. Similar to the "zooming out" view that Dorn applied to hide individual letters and highlight visual difference (Dorn, 2012), we propose a keyword labeling method to amplify the distinction among different Java elements (Gosling et al., 2000) before

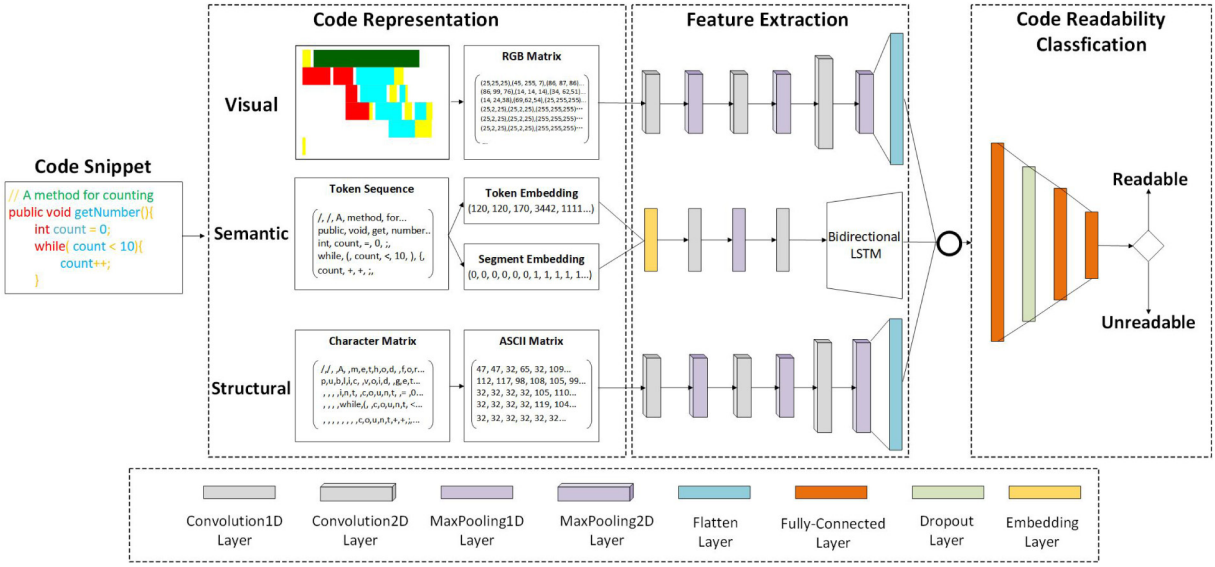

Fig. 1. Approach overview.

taking screenshots to gain a better result. To be specific, we first divide Java elements into five categories (i.e., comments, keywords, operators and separators, identifiers, literals). Then, we fill different categories of elements with the corresponding syntax highlighting color used in the code editor Eclipse to remain consistent with developers' reading habits. To capture the overall visual framework only and eliminate the influence of letters, we leave color blocks and hide specific characters. Finally, we take screenshots of those preprocessed data snippets and convert them into RGB matrices with a unified size of (128, 128, 3). Compared to the visual representation method Dorn applied, our representation method highlights more differences between code elements (Dorn, 2012). Using CNN also allows deeper visual features to be captured resulting in a better classification performance.

3.1.2. Semantic representation

It is commonly believed that code is just a particular form of text, where the readability of identifiers and comments largely affects the overall code readability (Dorn, 2012; Scalabrino et al., 2016). As shown in the research of Scalabrino et al. (2016), the addition of textual features (e.g., the relative number of identifiers composed by words present in an English dictionary) boosts the classification accuracy with a minimum improvement of 3.9%. Therefore, we propose a semantic representation method to preserve semantic information in the source code. As an effective pre-trained model that has an outstanding performance for capturing semantic features (González-Carvajal and Garrido-Merchán, 2020; Gao et al., 2019), we adopt BERT to be the word embedding method mapping source code into word vectors. As shown in Fig. 2, BERT accepts a token embedding, a segment embedding, and a position embedding as input (Devlin et al., 2018). Specifically, token embedding is an integer sequence in which each token is uniquely represented by an integer. Segment embedding is made up of sentence indexes representing which sentence every token is in. By using segment embedding, relationships between sentences are captured. Note that even the same token in the same sentence could express different semantics. Therefore, the position embedding consisting of position indexes of each token is needed. Due to the position embedding could be automatically generated from the token embedding, we only need to represent code snippets into token embeddings

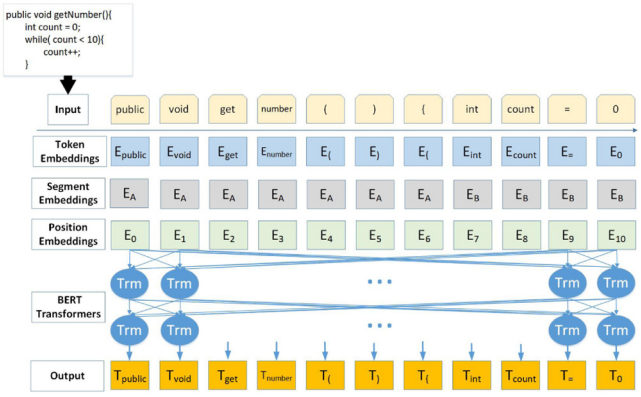

Fig. 2. Word embedding using BERT.

and segment embeddings. The detailed representation process is demonstrated in Figs. 1 and 2.

To represent a code snippet into the token embedding, comments and strings are equally divided into words where each word is considered as a token just like nature language processing which means we do not distinguish comments and strings. Furthermore, we treat other code elements (e.g., keywords, identifiers, separators) as a token and concatenate all tokens into a sequence to form a token embedding. To complete this transition, BERT tokenizer with the model size of BERTBASE (L=12, H=768, A=12, Total Parameters=110M) (Devlin et al., 2018) is used, because this size is suitable for the small dataset with fewer tokens. In addition, it is case sensitive which falls in line with our need, because variables with the same word but different cases are considered different in the context of code. Importantly, as Scalabrino et al.'s research (Scalabrino et al., 2016) suggested, a large portion of identifiers are composed of many words connected through naming conventions (i.e., upper camel case, and underscore case) where each of these words remarkably affects the code readability. To address this issue, we propose to split each identifier (including identifiers in comments) into words according to the naming convention it applied before using the tokenizer. To be specific, we replace underscores with whitespaces for identifiers using the underscore case (e.g., "has_next_int" is converted into "has next int"). For identifiers with camel case, we separate

words based on capital letters (e.g., "hasNextInt" is converted into "has next int"). Then, we convert code snippets into segment embeddings by using the sentence index of each token. Due to the difference between code and natural language, we treat each line as a sentence. Last but not least, to ensure a unified size and consider the impact of different sizes on feature extraction, we specify that the size of each embedding is (100,) and a special number 0 is used to pad.

3.1.3. Structural representation

Proved to be the most effective structural representation granularity proposed by Mi et al. (2018a), character matrix (i.e., character-level representation (Mi et al., 2018b)) is used to capture the structural features as shown in Fig. 1. Character matrix is similar to RGB matrix where each character could be viewed as a pixel. However, RGB matrices produced by visual representation are specialized to reveal the visual perception of code elements with the omission of detailed structures and specific characters. In this case, character matrices could complement this omission and retain more original readability information such as the logical structure of a program (e.g., if-else selection structure, loop structure). Specifically, code snippets are treated as two-dimensions character matrices in which every character, digit, and symbol (e.g., operators and punctuations) is converted into its corresponding ASCII value. Besides, to fully remain the structure of source code, we also reserve all whitespaces (i.e., spaces, horizontal tabs, line terminators). Finally, a special value, -1, is used in our experiments to pad ensuring that all matrices have the unified size of (50, 350). It is noticeable that the character matrix (for structural representation) and RGB matrix (for visual representation) differ in size. Because the size of the RGB matrix is rescaled for balancing processing speed and information reserved. Whereas the size of the character matrix is determined by the maximum numbers of rows and columns of all code snippets used in this experiment. Nevertheless, this decision on sizes is not applicable for practical use where any parts beyond the predetermined length will be discarded. To address this problem, we propose to reasonably partition longer snippets which will be demonstrated in our future work (see Section 8).

3.2. Feature extraction

Corresponding to the three proposed code representations, we build a hybrid model comprised of three feature extraction neural networks as presented in Fig. 1. Detailed explanations for them are shown in this section.

3.2.1. Visual features extraction network

For the sake of dealing with RGB matrices and extracting features from them, we decide to leverage CNN (i.e., Convolutional Neural Network) because it has achieved great success in the field of image classification due to the accurate feature extraction ability (Krizhevsky et al., 2012; He et al., 2016). Specifically, the visual features extraction network is comprised of three pairs of 2D convolutional layers and max-pooling layers with one flatten layer at the end. For the first two pairs, the convolutional layer contains 32 filters with the size of 2 whereas the size of max-pooling is set to be 2. Differently for the last layer, there are 64 filters with the size of 3 in the convolutional layer. The size of the last max-pooling layer also increases to 3. Visual features extraction network accepts input matrices of size (128, 128, 3). By utilizing the network, visual features of source code could be extracted automatically without human interference.

3.2.2. Semantic features extraction network

To combine the speed of CNN with the sequential sensitivity of RNN, we propose the use of CNN-BiLSTM (i.e., Convolutional Neural Network, Bidirectional Long Short Term Memory) to extract semantic features. We propose to use BiLSTM (Siami-Namini et al., 2019) because LSTM (Hochreiter and Schmidhuber, 1997) is a typical RNN which is proved to be effective in semantic feature extraction (Ravuri and Stolcke, 2015). On top of that, BiLSTM could take both previous and subsequent contexts into account, therefore utilizing more information than unidirectional LSTM (Graves and Schmidhuber, 2005). However, LSTM could not efficiently extract features from long sequences. Therefore, we add a CNN before LSTM to preprocess the input into shorter sequences composed of critical features identified and extracted by CNN (Zhou et al., 2015). To be specific, the CNN consists of two one-dimensional convolutional layers with filters of size 32 and a one-dimensional max-pooling layer with the size of 3. Relu is used as the activation function for this CNN. Apart from that, we adopt BERT (Devlin et al., 2018) as the word embedding method at the beginning of the semantic features extraction network. Therefore, the semantic features extraction network accepts two tensors, a token embedding, and a segment embedding, both with the size of (100,) as its input where the position embedding is derived from those two embeddings. The detailed word embedding process is shown in Fig. 2.

3.2.3. Structural features extraction network

Due to the similar structure between character matrices and RGB matrices, the structural features extraction network also utilizes CNN to extract structural features. (50, 350, 1) is the unified character matrices' size input into this feature extraction network. Having the same structure as the visual features extraction network, the structural features extraction network is comprised of three pairs of convolutional layers and max-pooling layers with a flatten layer at the end.

3.3. Code readability classification

After extracting visual, semantic, and structural features, a concatenation is performed to simply concatenate all features together with the output size of (21056,). Then, we use a fully-connected layer to reduce its dimensions into (64,) and finally input the result into a classifier. The classifier we used consists of two fully-connected layers with a dropout layer in the middle with a rate of 50%. The first fully-connected layer uses Relu as the activation whereas the second one uses Sigmoid. RMSprop is chosen to be the optimizer because of its ability to adaptively adjust the learning rate. RMSprop calculates the running average ($E[g^2]_t$) in terms of means square as shown in Eq. (1).

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2 \tag{1}$$

where g_t refers to the gradient of the parameters at the t-th iteration and ρ is a constant controlling the decay of the previous parameter updates. The parameters are updated as Eq. (2):

$$\Delta x_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{2}$$

where η is a learning rate that controls how large of a step to take in the direction of the negative gradient and ϵ is a constant added to better condition the denominator. The initial learning rate is set to be 0.0015 and it would be adjusted by RMSprop during training. Because code readability classification is considered as binary classification, we employ binary cross-entropy as the loss function due to its outstanding performance presented while dealing with such tasks (Chollet, 2016).

Table 1
Statistical summary of the code corpus.

Data source	# of Code snippets	# of Java snippets	# of Readable Java snippets	# of Unreadable Java snippets	# of annotators	Average lines	Average identifiers	Average keywords
Buse	100	100	25	25	121	7.80	15.26	3.91
Dorn	360	120	30	30	5468	30.81	36.88	11.11
Scalabrino	200	200	50	50	9	26.61	51.65	10.99

4. Experiment setup

In this section, we present our experiment design in terms of dataset, evaluation metrics, and research questions.

4.1. Dataset

We collected open-source code readability datasets from Buse and Weimer (2010), Dorn (2012) and Scalabrino et al. Some critical characteristics of those datasets are presented in Table 1. It is also noteworthy that all snippets are sourced from open-source projects such as JUnit and Hibernate. Besides, snippets from Buse and Weimer are tagged by 121 Computer Science students which are doubted by Dorn. Therefore, Dorn conducted a large-scale web-based survey involving over 5000 humans to label code snippets. On top of that, many code snippets are incomplete fragments to imitate a piece of code on the screen. However, incomplete snippets are not conducive to semantic analysis. Thus, Scalabrino et al. asked 9 Computer Science students to label a new dataset that is only comprised of complete code snippets. To align with prior deep learning-based code readability classification researches (Mi et al., 2018b; Mi et al., 2018a), Java code snippets from aforementioned three datasets are combined and used in this experiment to avoid over-fitting. Though combining datasets with different annotation approaches adds some bias, it could increase the data diversity and improve model generalization performance which is the major reason why previous deep learning-based code readability classification researches use combined dataset. Therefore, there are 210 code snippets in total, half of which is readable data and the other half is unreadable data.

To evaluate our method, we constructed and trained our model on TensorFlow. We split out 10% of the original dataset as the test set and the rest as the training set. While training, we adopt ten-fold cross-validation (Kohavi et al., 1995) to mitigate the danger of over-fitting. To be specific, we randomly partition the training set into 10 sub-sets training on nine of them and validating on the last one. This process is repeated ten times so that each subset is used as the validation set exactly once. Besides, we set the batch size as 42 and iterated our model for 20 epochs during training.

4.2. Evaluation metrics

Instead of merely using accuracy like prior code readability classification researches (Buse and Weimer, 2010; Scalabrino et al., 2016; Dorn, 2012), we include three more evaluation metrics, f-measure, AUC, and MCC, to comprehensively evaluate our model. For all evaluation metrics we adopted, a higher value indicates a better model performance.

Accuracy is the most commonly used evaluation metric by all code readability classification research. It could be simply interpreted as the ratio of correctly classified samples over all samples as shown in Eq. (3).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

where TP (True Positive) refers to the number of code snippets correctly classified as readable and TN (True Negative) refers to the number of code snippets correctly classified as unreadable. FP (False Positive) is the number of code snippets misclassified as readable and FN (False Negative) is the number of code snippets misclassified as unreadable.

F-measure is a comprehensive evaluation metric defined as the weighted harmonic average of precision and recall as shown in Eq. (4).

$$F - Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{4}$$

where Precision refers to $\frac{TP}{TP+FP}$ and Recall refers to $\frac{TP}{TP+FN}$.

AUC (Hanley and McNeil, 1982) is defined as the area under the ROC curve in the coordinate system where the x-axis is FPR (False Positive Rate) and the y-axis is TPR (True Positive Rate). With regard to binary code readability classification, FPR represents the ratio of code snippets misclassified as readable among all unreadable snippets, whereas TPR represents the ratio of code snippets correctly classified as readable among all readable snippets.

MCC (Matthews, 1975) is another balanced evaluation metric specialized for binary classification evaluation which fully takes TP, TN, FP, and FN into account. As a result, MCC could produce a more informative and truthful evaluation score in binary classifications than accuracy and f-measure because a higher MCC score indicates the prediction obtained positive results in all four confusion matrix categories (i.e., TP, FN, TN, FP) (Chicco and Jurman, 2020). The calculation of MCC is shown in Eq. (5):

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5}$$

4.3. Research questions

We formulate three research questions to validate our method.

RQ1: How do different parameter settings affect the model performance?

Approach: Two hyperparameters, word embedding methods and classifiers will be discussed in this RQ. For the word embedding methods, we compare BERT and GloVe. As for the classifiers, we compare our proposed FC-based classifier (Fully-Connected based classifier) with two classical classifiers, random forest classifier (Breiman, 2001) and k-nearest neighbors classifier (Peterson, 2009). Accuracy, f-measure, MCC, and AUC are chosen to be the evaluation metrics for this RQ to give a comprehensive comparison.

RQ2: Which feature type is more critical in the field of code readability classification?

Approach: We propose three code representation methods and corresponding feature extraction networks to extract three types of features. In this RQ, we explore the validity of those proposed feature types and compare their importance. Specifically, we decide to conduct an ablation study which allows us to assess the relative contribution of three types of readability features more clearly. In the ablation study, we compare ModelST (i.e., the model without visual features), ModelTV (i.e., the model without semantic features), ModelSV (i.e., the model without structural features) with the combined model (i.e., the model using all

Table 2
Comparison of different word embedding methods.

Model	Accuracy	F-Measure	AUC	MCC
BERT	0.790	0.777	0.851	0.691
GloVe	0.783	0.760	0.835	0.699

Table 3
Comparison of different classifiers.

Model	Accuracy	F-Measure	AUC	MCC
FC-based Classifier	0.853	0.850	0.908	0.738
Random Forest Classifier	0.779	0.769	0.781	0.563
K-Neighbors Classifier	0.775	0.760	0.777	0.559

features) on the validation set to explore the impact of each kind of features. In this RQ, we also use accuracy, f-measure, MCC, and AUC as evaluation metrics. Besides, hyperparameters selection aligns with the results of RQ1.

RQ3: Can our model outperform existing models?

Approach: To verify the validity of our model, we compare our model with six existing code readability classification models including four traditional machine learning-based models which use handcrafted features (Buse and Weimer, 2010; Dorn, 2012; Posnett et al., 2011; Scalabrino et al., 2016) and two advanced deep learning-based models put forward by Mi et al. (2018b,a). Accuracy is proposed to be the evaluation metric in comparison because it is the most commonly used metric by all aforementioned research.

5. Results

In this section, results according to each RQ are presented. It is crucial to note that RQ1 and RQ2 are discussed based on results gained on ten-fold validation, whereas RQ3 is addressed by results on the test set.

RQ1: How do different parameter settings affect the model performance?

In this RQ, two hyperparameters, word embedding methods and classifiers are discussed to explore their effectiveness. First, we conduct control experiments to compare BERT and GloVe with the same dataset, model architecture, and training parameters. The results are presented in Table 2.

It can be seen that there is no significant difference between BERT and GloVe. This is probably because code has simpler semantics making no remarkable difference between a state-of-the-art word embedding model, such as BERT, and others. However, BERT still outperforms GloVe to a small extent on most evaluation metrics. We conjecture the reason behind that is BERT not only considers the semantics of each token but also takes the segmentation and position information into account which is more comprehensive than GloVe. Additionally, the bi-directional transformer used by BERT also allows it to address the previous and subsequent context simultaneously. Therefore, we adopt BERT to be the word embedding method in the semantic features extraction network.

In the comparison of different classifiers, we remain the feature input into three classifiers the same and conduct control experiments to evaluate them independently. The results are shown in Table 3.

From Table 3, the FC-based classifier outperforms the other two classifiers. Compared with the other two classifiers, the average accuracy of the FC-based classifier is improved by about 8%–10% which is a statistically significant difference. In order to compare the stability of different classifiers, we repeat the cross-validation ten rounds and present the results in Fig. 3. It is noticeable that the accuracy of the FC-based classifier fluctuates within a relatively small range, which indicates better stability. Based on the result, we use the FC-based classifier as the classifier.

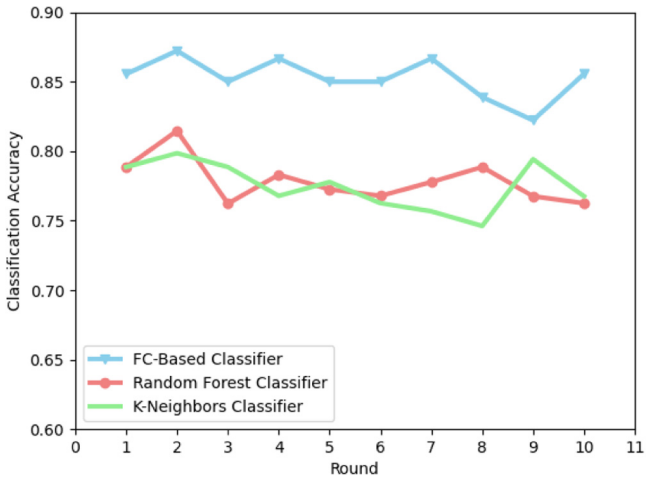

Fig. 3. Accuracy of different classifiers in ten rounds cross-validation.

RQ2: Which feature type is more critical in the field of code readability classification?

To facilitate fair comparison, we train ModelST, ModelTV, ModelSV, and the combined model on the same dataset while using BERT and the FC-based classifier based on results from RQ1.

The result is shown in Table 4. It is remarkable that the combined model outperforms other three models on all four evaluation metrics. This result proves that the absence of any of three proposed feature types degrades model performance. Therefore, the validity of proposed three code readability feature types is proven. Apart from that, the relative contribution could be inferred from the magnitude of the decline in model performance. We could see that ModelSV performs worse than the other two on most evaluation metrics which indicates structural features make a greater contribution for code readability classification. We conjecture the reason is that structural representation could maximumly remain the code layout and whitespaces that mainly affects code readability. By contrast, ModelST has the least performance degradation making visual features the least important. Besides, the result also indicates that semantic features are good complements to provide semantic information missed by structural and visual features. In conclusion, structural features are more critical in code readability classification. However, visual and semantic features are also indispensable for a good classification performance because they could allow a more comprehensive capture of readability information.

RQ3: Can our model outperform existing models?

RQ3 compares our proposed model with six classical code readability classification models based on accuracy. Because deep learning-based model requires more data in training, three code readability datasets (i.e., datasets from Buse and Weimer, Dorn and Scalabrino et al.) are combined and used in training to avoid severe over-fitting. Results are shown in Fig. 4.

Compared to traditional machine learning-base models (i.e., the models of Buse and Weimer, (2010), Dorn, (2012), Posnett et al., (2011), and Scalabrino et al., (2016)), our model improves the accuracy ranging from 3.5% to 13.8%. Importantly, in comparison with state-of-the-art deep learning-based models, IncepCRM and DeepCRM, our proposed model improves the accuracy by 2.5% and 1.5% respectively. The major reason behind

Table 4
Ablation study to explore relative contribution of different feature types.

Model	Accuracy	F-Measure	AUC	MCC
Combined Model	0.853	0.850	0.908	0.738
ModelST	0.829 (↓ 0.024)	0.828 (↓ 0.022)	0.887 (↓ 0.021)	0.700 (↓ 0.038)
ModelTV	0.827 (↓ 0.026)	0.807 (↓ 0.043)	0.883 (↓ 0.025)	0.691 (↓ 0.092)
ModelSV	0.815 (↓ 0.038)	0.806 (↓ 0.044)	0.881 (↓ 0.027)	0.692 (↓ 0.046)

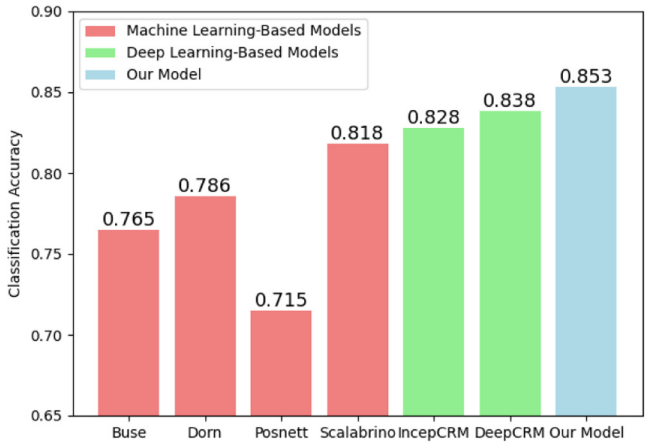

Fig. 4. Accuracy comparison of readability classification models.

this improvement is that although DeepCRM proposed three granularities of code representation, IncepCRM and DeepCRM are still limited to the structural representation only. In contrast, we capture code readability features from two other critical aspects and complement the structural representation with more code readability information. Besides, on three other evaluation metrics, our model also performs well with 85.0% f-measure, 90.8% AUC, and 73.8% MCC. Though our model cannot run on a single dataset (e.g., only use Dorn's dataset in training and testing) because of severe result fluctuation (see Section 6 for details), we decide to track back all test instances in order to calculate a respective accuracy on each dataset to give a more cogent comparison. Results have shown that our model gains an average accuracy of 79.7% on Buse's dataset, 85.5% on Dorn's dataset, 88.0% on Scalabrino's dataset, where all three accuracies get improved as compared to state-of-the-art code readability classification models.

6. Discussion

In this section, we first discuss why we use deep neural networks to extract features instead of reusing previous features that were already engineered. Then, we discuss some factors that remarkably limit our model performance.

Although readability-related features were already engineered in previous studies (e.g., Buse and Weimer's structural features and Scalabrino et al.'s textual feature), we argue that reusing these features is not without cost.

- First, the process of manually extracting features from source code is effort-intensive. Actually, there are around 100 readability-related features defined in the literature (Scalabrino et al., 2016). The workload is heavy if we want to reuse them. For instance, to extract Buse and Weimer's structural features, we have to implement a parser for source code analysis and manipulation, whereas Dorn's DFT-related visual features and Scalabrino et al.'s well-designed textual feature (e.g., narrow meaning identifiers) require more domain-specific knowledge and effort to obtain. Even with the help of useful tools such as srcML[2] and WordNet,[3] it is going to be more time-consuming and labor-intensive than simply using a deep neutral network for automatic feature extraction.
- Second, redundancies and overlaps may exist between manually designed features and features extracted by our deep neural networks, which is likely to limit the model performance. If we want to reuse previous features, we have to examine their validity (some features may not work well with our method) as well as retrain (even redesign) our classifier, which would cause a great deal of extra work.
- Third, most existing code readability studies focus solely on Java code (Buse and Weimer, 2010; Scalabrino et al., 2016; Mi et al., 2018a). When we need to extend to other programming languages like JavaScript (which is quite different from Java), it definitely requires a large amount of work to add, delete, modify and re-examine the handcrafted features according to the characteristics of the targeted language (it is also unavoidable to reimplement a new code parser for feature extraction), indicating a lack of generality in previous methods. In contrast, our representation method is adaptable with only slight adjustments, since most parts of transforming code snippets into RGB matrices, token sequences and character matrices are language-independent. We could simply retrain our model with JavaScript code snippets which has less work to do.

In conclusion, although time was already spent on handcrafting good features, reusing them along with our method would still take a lot time and labor. Besides, although programs contain rich statistical properties, they are difficult for humans to capture (Hindle et al., 2012; Peng et al., 2015). Accordingly, we consider that the traditional feature engineering process cannot capture sufficient information about the source code, which may well limit the model performance. To address this issue, we turn to learning-based approaches that can automatically discover, learn, and extract complicated underlying features from the source code. According to the experimental results, we do obtain an improved performance on the task of code readability classification across all evaluation metrics.

While constructing our model, multiple representation methods were tried out along with fine-tuning the architecture and parameters for each feature extraction network and the classifiers. Finally, our model has gained a better performance than all existing code readability classification models. However, the performance of our model reaches a bottleneck with an average accuracy of 85.3%. Thus, we discuss potential factors still limiting our model in this section.

The limited dataset is a critical issue that primarily limits the model performance and results in over-fitting. While we tried to run experiments on a single dataset instead of combining three datasets to better compare the performance with previous models, the result is problematic with a large fluctuation. As shown in Table 5, our model gains an average accuracy of 0.770 on Buse's dataset, 0.775 on Dorn's dataset and 0.845 on Scalabrino's dataset, with variance of 0.100, 0.092, 0.069 respectively.

[2] https://www.srcml.org
[3] https://wordnet.princeton.edu

Table 5
Model performance on different datasets.

Dataset	Average accuracy	Highest accuracy	Lowest accuracy	Variance
Buse's Dataset	0.770	0.900	0.600	0.100
Dorn's Dataset	0.775	0.917	0.667	0.092
Scalabrino's Dataset	0.845	0.900	0.700	0.069
Combined Dataset	0.853	0.872	0.822	0.014

In comparison, the variance is only 0.014 after combining three datasets. Indeed, there is a severe result fluctuation as well as performance deterioration. The major reason of this phenomenon is that deep neural networks require more data in training where a single dataset cannot match. Actually, even we have used all available data for model training in our experiments (see RQ3 in Section 5 for details), we argue that the model performance is still limited by the shortage of training data. Since in our latest research (Mi et al., 2021), we have preliminarily demonstrated that the classification performance of deep neural networks can be significantly improved when they are trained on an augmented corpus.

Besides, a small test set also leads to a phenomenon that the distribution of the training set is very likely to be disparate from the test set distribution. For instance, if the training set only contains samples collected from Buse and the test set happens to be samples from Scalabrino which have different characteristics, the result is likely to be negative. Although we use ten-fold validation to alleviate this issue, it still brings some influences on our results. To solve the problem of over-fitting and result fluctuations, we plan to refine data augmentation techniques in our future work to better support code readability classification.

The selection of the word embedding method also affects our result to a considerable extent. Even though BERT performs well in natural language processing tasks, the significant difference between natural language and source code limits its performance. As a result, we might not make full use of BERT to extract code-based semantic features. If we could pre-train BERT using collected code snippets rather than natural text to make it suitable for the context of program analysis, our model performance is likely to be further improved.

7. Threats to validity

In this section, we outline some threats to the validity of our results.

Internal Validity. The lack of labeled data is a major internal threat to our conclusions. For instance, due to limited sample size, each of classifiers is more or less constrained, which means the ideal performance of these classifiers is undiscovered. Therefore, the conclusions made in the comparison of different classifiers might be overturned when more data are available. Besides, because our model requires relatively more data in training, we could not train and test our model using a single dataset at a time. To avoid severe over-fitting, we combine all three previous datasets and get the conclusions for RQ3. Although previous deep learning-based models are trained on the same dataset as ours (Mi et al., 2018b; Mi et al., 2018a), the datasets of machine learning-based models are just a part of our dataset (Buse and Weimer, 2010; Dorn, 2012; Scalabrino et al., 2016; Posnett et al., 2011). In this case, the characteristics and evaluation of datasets are different. Therefore, the conclusions made in RQ3 might not be rigorous.

External Validity. Currently, our model has not been trained on other programming languages but Java which limits its generalizability. Besides, our model could not handle longer code snippets due to the predefined maximum length we set. To address this issue, we propose to partition longer code snippets in our future work.

Construct Validity. Most previous code readability classification researches merely used accuracy and f-measure. This approach might not be comprehensive enough in evaluating model performance. Thus, we propose to add AUC and MCC in our evaluation metrics to comprehensively evaluate our model and minimize threats to construct validity.

8. Conclusions and future work

To further improve code readability classification, we propose a method based on BERT, CNN, and BiLSTM to extract code readability-related features through visual, semantic, and structural representations aiming to retain as much original information of source code as possible. A series of experiments are performed to evaluate the effectiveness of the proposed method. The experimental results show that our model can reach a state-of-the-art code readability classification accuracy of 85.3%, which is 1.5%–13.8% higher than existing models. Accordingly, our method of using visual, semantic, and structural features has been proved to be most effective for the task of code readability classification.

Future work mainly includes potential methods to improve the performance of our model and make it applicable in practical use. First and foremost, the size of the readability dataset could be deemed as the major problem limiting our model performance. However, the traditional way of enlarging datasets, conducting a large-scale survey to collect samples (Buse and Weimer, 2010; Dorn, 2012)(Scalabrino et al., 2016), is too costly in both time and budget. Therefore, we consider the data augmentation method proposed by Mi et al. (2021) as a feasible way to enlarge the readability dataset. Apart from the dataset, we will explore using graphs (together with Graph Neural Networks) to represent both the syntactic and semantic structure of source code (Allamanis et al., 2017).

Besides, treating code readability classification as binary classification is not applicable in real scenarios. For instance, an actual code file is likely to contain thousands of lines and is even written by multiple programmers which leads to a phenomenon that parts of the code are readable and others are unreadable. In this case, our proposed model is too rough to give readability judgments. In the future, we will try to treat code readability classification as multi-class classification just as text readability (Paasche-Orlow et al., 2003). We will also try to improve our code readability model which can deal with very long code files and give a readability score to each partition of long code files. For instance, any long code snippets exceeding the predefined maximum length will be partitioned into smaller sub-snippets based on code blocks. Then, their code readability levels will be calculated respectively. Besides, we will also define a maximum line length according to coding guidelines. For lines that are longer than the predefined value, we will discard them and decrease its readability level to a certain extent because an over long line diminishes readability.

CRediT authorship contribution statement

Qing Mi: Conceptualization, Methodology, Data curation, Writing – review & editing, Supervision, Funding acquisition. **Yiqun Hao:** Writing – original draft, Visualization. **Liwei Ou:** Software, Formal analysis, Investigation. **Wei Ma:** Writing – review & editing, Supervision, 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 was supported by the GHfund B (20220202, ghfund202202028015) and the Spark Project of Beijing University of Technology, China (Project No. XH-2021-02-24).

References

Aggarwal, K.K., Singh, Y., Chhabra, J.K., 2002. An integrated measure of software maintainability. In: Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No. 02CH37318). IEEE, pp. 235–241.

Alexandru, C.V., 2016. Guided code synthesis using deep neural networks. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 1068–1070.

Allamanis, M., Brockschmidt, M., Khademi, M., 2017. Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740.

Binkley, D., Davis, M., Lawrie, D., Morrell, C., 2009. To camelcase or under_score. In: 2009 IEEE 17th International Conference on Program Comprehension. IEEE, pp. 158–167.

Boehm, B., Basili, V.R., 2001. Software defect reduction top 10 list. Computer 34 (1), 135–137. http://dx.doi.org/10.1109/2.962984.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.

Buse, R.P., Weimer, W.R., 2010. Learning a metric for code readability. IEEE Trans. Softw. Eng. 36 (4), 546–558. http://dx.doi.org/10.1109/TSE.2009.70.

Buse, R.P., Zimmermann, T., 2012. Information needs for software development analytics. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE, pp. 987–996.

Chicco, D., Jurman, G., 2020. The advantages of the matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC Genomics 21 (1), 1–13.

Chollet, F., 2016. Building powerful image classification models using very little data, Vol. 5. Keras Blog.

de Vries, W., van Cranenburgh, A., Bisazza, A., Caselli, T., van Noord, G., Nissim, M., 2019. Bertje: A dutch bert model. arXiv preprint arXiv:1912.09582.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Dorn, J., 2012. A general software readability model, vol. 5. pp. 11–14, MCS Thesis Available from (http://www.cs.virginia.edu/weimer/students/dorn-mcs-paper.pdf).

Elshoff, J.L., Marcotty, M., 1982. Improving computer program readability to aid modification. Commun. ACM 25 (8), 512–521.

Gao, Z., Feng, A., Song, X., Wu, X., 2019. Target-dependent sentiment classification with BERT. IEEE Access 7, 154290–154299.

González-Carvajal, S., Garrido-Merchán, E.C., 2020. Comparing BERT against traditional machine learning text classification. arXiv preprint arXiv:2005.13012.

Gosling, J., Joy, B., Steele, G., Bracha, G., 2000. The Java Language Specification. Addison-Wesley Professional.

Graves, A., Schmidhuber, J., 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Netw. 18 (5–6), 602–610.

Gu, X., Zhang, H., Zhang, D., Kim, S., 2016. Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 631–642.

Hanley, J.A., McNeil, B.J., 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology 143 (1), 29–36.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 770–778.

Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P., 2012. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering. In: ICSE '12, IEEE Press, Piscataway, NJ, USA, pp. 837–847, URL http://dl.acm.org/citation.cfm?id=2337223.2337322.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Holmes, G., Donkin, A., Witten, I.H., 1994. Weka: A machine learning workbench. In: Proceedings of ANZIIS'94-Australian New Zealnd Intelligent Information Systems Conference. IEEE, pp. 357–361.

Jiang, S., Armaly, A., McMillan, C., 2017. Automatically generating commit messages from diffs using neural machine translation. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE pp. 135–146.

Kohavi, R., et al., 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Ijcai, Vol. 14. (2), Montreal, Canada pp. 1137–1145.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. 25, 1097–1105.

Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2015. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 476–481.

Likert, R., 1932. A technique for the measurement of attitudes. Arch. Psychol..

Matthews, B., 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. Biochimica Et Biophys. Acta (BBA) - Protein Struct. 405 (2), 442–451.

Mi, Q., Keung, J., Xiao, Y., Mensah, S., Gao, Y., 2018a. Improving code readability classification using convolutional neural networks. Inf. Softw. Technol. 104, 60–71.

Mi, Q., Keung, J., Xiao, Y., Mensah, S., Mei, X., 2018b. An inception architecture-based model for improving code readability classification. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. pp. 139–144.

Mi, Q., Xiao, Y., Cai, Z., Jia, X., 2021. The effectiveness of data augmentation in code readability classification. Inf. Softw. Technol. 129, 106378.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. 26,

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Thirtieth AAAI Conference on Artificial Intelligence.

Mou, L., Men, R., Li, G., Zhang, L., Jin, Z., 2015. On end-to-end program generation from user intention by deep neural networks. arXiv preprint arXiv:1510.07211.

Müller, M., Salathé, M., Kummervold, P.E., 2020. Covid-twitter-bert: A natural language processing model to analyse covid-19 content on twitter. arXiv preprint arXiv:2005.07503.

Nguyen, A.T., Nguyen, T.N., 2017. Automatic categorization with deep neural network for open-source java projects. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion. ICSE-C, IEEE, pp. 164–166.

Nguyen, T.D., Nguyen, A.T., Phan, H.D., Nguyen, T.N., 2017. Exploring API embedding for API usages and applications. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 438–449.

Paasche-Orlow, M.K., Taylor, H.A., Brancati, F.L., 2003. Readability standards for informed-consent forms as compared with actual readability. N. Engl. J. Med. 348 (8), 721–726.

Peng, H., Mou, L., Li, G., Liu, Y., Zhang, L., Jin, Z., 2015. Building program vector representations for deep learning. In: Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management - Volume 9403. In: KSEM 2015, Springer-Verlag New York, Inc., New York, NY, USA, pp. 547–553. http://dx.doi.org/10.1007/978-3-319-25159-2_49.

Pennington, J., Socher, R., Manning, C.D., 2014. Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. EMNLP, pp. 1532–1543.

Peterson, L.E., 2009. K-nearest neighbor. Scholarpedia 4 (2), 1883.

Posnett, D., Hindle, A., Devanbu, P., 2011. A simpler model of software readability. In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 73–82.

Ravuri, S., Stolcke, A., 2015. Recurrent neural network and LSTM models for lexical utterance classification. In: Sixteenth Annual Conference of the International Speech Communication Association.

Raychev, V., Vechev, M., Yahav, E., 2014. Code completion with statistical language models. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 419–428.

Raymond, D.R., 1991. Reading source code. In: Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research pp. 3–16.

Scalabrino, S., Linares-Vasquez, M., Poshyvanyk, D., Oliveto, R., 2016. Improving code readability models with textual features. In: 2016 IEEE 24th International Conference on Program Comprehension. ICPC, IEEE, pp. 1–10.

Shu, C., Zhang, H., 2017. Neural programming by example. In: Thirty-First AAAI Conference on Artificial Intelligence.

Siami-Namini, S., Tavakoli, N., Namin, A.S., 2019. The performance of LSTM and BiLSTM in forecasting time series. In: 2019 IEEE International Conference on Big Data (Big Data). IEEE, pp. 3285–3292.

Tenney, I., Das, D., Pavlick, E., 2019. BERT rediscovers the classical NLP pipeline. arXiv preprint arXiv:1905.05950.

White, M., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., 2015. Toward deep learning software repositories. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp. 334–345.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A.E., Li, S., 2017. Measuring program comprehension: A large-scale field study with professionals. IEEE Trans. Softw. Eng. 44 (10), 951–976.

Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C., 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering. pp. 404–415.

Zhang, F., Peng, X., Chen, C., Zhao, W., 2018. Research on code analysis based on deep learning. Comput. Appl. Softw. 35 (06), 15–23.

Zhou, C., Sun, C., Liu, Z., Lau, F., 2015. A C-LSTM neural network for text classification. arXiv preprint arXiv:1511.08630.

Zhou, J., Zhang, H., Lo, D., 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE, pp. 14–24.