scientific reports manuscript No.
(will be inserted by the editor)

CodePrompt: Improving Source Code-Related Classification with Knowledge Features through Prompt Learning

Yong Ma · Senlin Luo · Yu-Ming Shang(✉) · Yifei Zhang · Zhengjun Li

Received: date / Accepted: date

Abstract Researchers have explored the potential of utilizing pre-trained language models, such as CodeBERT, to improve source code-related tasks. Previous studies have mainly relied on CodeBERT's text embedding capability and the '[CLS]' sentence embedding information as semantic representations for fine-tuning downstream source code-related tasks. However, these methods require additional neural network layers to extract effective features, resulting in higher computational costs. Furthermore, existing approaches have not leveraged the rich knowledge contained in both source code and related text, which can lead to lower accuracy. This paper presents a novel approach, CodePrompt, which utilizes rich knowledge recalled from a pre-trained model by prompt learning and an attention mechanism to improve source code-related

Yong Ma
Beijing Institute of Technology
No. 5, Zhongguancun South Street, Haidian District, Beijing, China
E-mail: znsoft@bit.edu.cn
E-mail2: znsoft@163.com

Senlin Luo
Beijing Institute of Technology
No. 5, Zhongguancun South Street, Haidian District, Beijing, China
E-mail: luosenlin2019@126.com

Yu-Ming Shang
Beijing University of Posts and Telecommunications
No.10 Xitucheng Road, Haidian District, Beijing, China
E-mail: shangym@bupt.edu.cn

Zhengjun Li
Beijing Institute of Technology
No. 5, Zhongguancun South Street, Haidian District, Beijing, China
E-mail: zjli@bit.edu.cn

Yifei Zhang
Beijing Institute of Technology
No. 5, Zhongguancun South Street, Haidian District, Beijing, China
E-mail: zyf2012kaoyan@163.com

classification tasks. Our approach initially motivates the language model with prompt information to retrieve abundant knowledge associated with the input as representative features, thus avoiding the need for additional neural network layers and reducing computational costs. Subsequently, we employ an attention mechanism to aggregate multiple layers of related knowledge for each task as final features to boost their accuracy. We conducted extensive experiments on four downstream source code-related tasks to evaluate our approach and our results demonstrate that CodePrompt achieves new state-of-the-art performance on the accuracy metric while also exhibiting computation cost-saving capabilities.

Keywords Source code · Classification · Prompt learning · Attention

1 Introduction

The intersection of machine learning, programming languages, and software engineering has garnered significant interest from the software engineering community. For source code classification tasks, Bayesian [1], Random Forest [2], and XGBoost [3] have been employed to detect programming languages. Additionally, deep learning techniques, such as TextCNN [4], are widely utilized in software engineering-related tasks. Recently, large-scale pre-trained language models, such as BERT [5], RoBERTa [6], GPT [7], and T5 [8], have emerged as promising tools for various downstream Natural Language Processing (NLP) tasks [9]. The impressive performance of pre-trained language models (PLM) in NLP tasks has inspired intensive research in the field of software engineering. Source code-related tasks have achieved remarkable success with source code-dedicated pre-trained language models, such as CodeBERT [10], CodeT5 [11], and GraphCodeBERT[12], as demonstrated by recent studies [13, 14]. Previous research [15, 16] has shown that BERT can capture a rich hierarchy of linguistic information in NLP tasks. Specifically, surface features are captured in lower layers, syntactic features in middle layers, and semantic features in higher layers [16]. However, a common approach that limits the ability of BERT-based models is to utilize the first token '[CLS]' in the highest layer of model output as the representative of the entire input text sequence. Other studies have proposed utilizing the '[CLS]' tokens from multiple layers to represent the input information [17]. Nevertheless, both of these approaches only make use of the semantic features extracted from a model's output. To address this limitation, some studies have incorporated additional layers, such as Long Short Term Memory (LSTM), to enhance feature extraction, although this approach incurs a significant computational cost.

In the field of software engineering, discovering a novel classification approach with low computational costs and strong feature presentation capability has become a crucial requirement. Fortunately, the rapid development of large-scale language models [18] is bringing promising solutions by introducing prompt learning as the fourth paradigm for both programming and natural language processing tasks [19]. One common approach to leveraging prompts is to use a natural language prompt template to wrap the input text sequence, followed by performing masked language modeling with a pre-trained language model. For instance, in a text classification task, the text sequence $\mathbf{x}$ can be wrapped into a prompt template, such as "It was [MASK]. $\mathbf{x}$," and then input into a language model. The logits from the output language model contain recalled knowledge at the location of '[MASK]', which can be projected to a specific category for classification tasks. This approach bridges the gap between pre-trained tasks and downstream NLP tasks.

Prompt learning has potent capabilities for knowledge representation [19] and can avoid the need for additional neural network layers for better feature extraction. To reduce computational costs and improve the performance of source code-related tasks, we can represent the input text sequence using the knowledge stimulated from a language model. The output of a BERT-based model, such as CodeBERT, consists of multiple layers, and each output at the '[MASK]' location has a different level of knowledge about the input information. By aggregating the most representative knowledge that a language model outputs to express the input information [16] for each task, we can obtain a more comprehensive set of distinguishing features to enhance task accuracy.

Inspired by the aforementioned ideas, we propose CodePrompt, a novel method for source code-related classification that leverages multiple layers of knowledge output from a language model to boost the performance of source code-related tasks. Specifically, we first wrap the input text into a hand-crafted prompt template as a prompt, then feed it into a language model to obtain knowledge that features the input text. Finally, we utilize the attention mechanism [20] to aggregate the importance of knowledge across different layers for a given task and project it to a specific class. This enables us to fully leverage the rich knowledge output of each layer in the BERT-based model based on prompt learning and avoids the need for additional neural network layers for feature extraction, reducing computation costs. Aggregating important knowledge through the attention mechanism improves the capability of feature representation and boosts the accuracy of tasks.

To validate the effectiveness of our proposed approach, we selected four typical downstream tasks related to source code, namely code language classification, code smell classification, technical debt classification, and code comment classification. The experimental results demonstrate that our method can achieve new state-of-the-art performance on the Accuracy metric for the four tasks. Additionally, we conducted ablation experiments to verify the reliability of our components for different task settings.

To the best of our knowledge, the main contributions of our study can be summarized as follows:

1. Our paper is the first to combine the prompt-learning paradigm with source code-related tasks, advancing the technical progress of the software engineering field.
2. Our study introduces CodePrompt, a novel approach that capitalizes on the knowledge feature learned through prompt learning and aggregates

essential knowledge using an attention mechanism. As a result, we achieve new state-of-the-art results on metric Accuracy while significantly reducing computational costs compared to previous studies.

3. We evaluate our approach on four classical source code-related tasks and demonstrate its effectiveness on both programming language and natural language tasks through a comprehensive set of experiments.
4. We have made our trained models and related code publicly available in our GitHub repository[1] to facilitate researchers in reproducing the results of our study or conducting further research.

This paper is structured as follows. Section 2 provides an overview of the related works on CodePrompt. Section 3 outlines the detailed design of our proposed approach. In Section 4, we present the experimental setting of CodePrompt, along with the details of all the baselines used in the experiments. Section 5 presents the experimental results and analysis. We also discuss a variety of threats to our approach in Section 6. Finally, we conclude the paper in Section 7.

2 Related Work

2.1 Code Representation Learning

Significant advances have been made in the study of the intersection of machine or deep learning, programming languages, and software engineering, based on the assumption that programming code resembles natural language [21]. However, raw source code cannot be directly fed into machine or deep learning models, and code representation is a fundamental step in making source code compatible with these models. This involves preparing a numerical representation of the source code that can be used to solve specific software engineering problems.

In this subsection, we briefly introduce representative studies related to source code representation learning. As source code has rich structure, it cannot be treated as only a series of text tokens [21, 22]. Harer et al. [23] tokenize a code snippet and categorize all tokens into different bins, such as comments and string literals. All tokens with the same categorical information are mapped to a unified identifier, which is then transformed into a vector using the word2vec algorithm [24]. DeFreez et al. [25] proposed the Func2Vec method, which embeds the control-flow graph of a function as a vector to represent the function and facilitates estimating the similarity of functions. A context-incorporating method [22] was proposed to use syntactic and type annotations information for source code embedding, which can distinguish the lexical tokens in different syntactic and type contexts. Code2Vec [26] offers a new approach, which decomposes code into a collection of paths in its abstract syntax tree (AST), learning the atomic representation of each path and how

[1] https://github.com/BIT-ENGD/codeprompt

to aggregate a set of them. For the problem of large ASTlength, Zhang et al. [27] found that splitting a large AST into a number of small statement trees and then encoding them as vectors can capture both lexical and syntactic knowledge of the statements. Motivated by the need for code summarization, Hu et al. [28] proposed a simple representation of code that only leverages the vectors of a sequence of API names to express a code snippet. With the advancement of large language models (LLM), a number of methods based on LLM have been proposed. Yang et al. [29] offer a fresh way by utilizing LLM and convolutional neural networks (CNN) to extract the feature representation of a code snippet. Since BERT is not dedicated to source code-related tasks, Feng et al. [10] utilize both code snippets and natural language to construct a bimodal pre-trained model, CodeBERT, that can effectively represent source code-related material. Recently, Jain et al. [30] introduced contrastive learning into code representation learning, as BERT-based models are much more sensitive to source code edits and cannot represent similar code snippets with slightly different literal expressions.

2.2 Source Code Related Classification

Source code-related classification, which includes code language classification, code smell classification, code comment classification, and technical debt classification, are four crucial tasks in software engineering that have been thoroughly investigated by researchers.

2.2.1 Code Language Classification

In most source code-related tasks, code language classification is the initial step for further processing. Previously, the programming language of a piece of source code was assigned manually or determined based on its file extension [4]. SC++ [3] employs the Random Forest Classifier (RFC) and XGBoost (a gradient boosting algorithm) to build a machine learning classifier that can detect programming languages, even for code snippets from a family of programming languages such as C, C++, and C#. Khasnabish et al. [1] utilized several variants of Bayesian classifier models to detect 10 programming languages and achieved remarkable results. Multiple layers of neural networks and convolutional neural networks were trained to judge the programming language of over 60 kinds of source code [4]. Large language models have demonstrated their enormous power in natural language tasks. Inspired by this, some researchers [31] have also employed large language models, such as RoBERTa, for source code classification with successful results.

2.2.2 Code Comment classification

Code comments are a powerful tool to help programmers understand and maintain code snippets, but different comments can have different intentions

[32]. Rabi et al. [33] developed a multilingual approach to code comment classification that utilizes natural language processing and text analysis to classify common types of class comment information with high accuracy for Python, Java, and Smalltalk programming languages. The Naive Bayes classifier, the J48 tree model, and the Random Forest model underlie the classifier used. To reveal the relationship between a code block and the associated comment's category, Chen et al. [34] classified comments into six intention categories and manually labeled 20,000 code-comment pairs. These categories include "what", "why", "how-to-use," "how-it-is-done", "property", and "others".

2.2.3 Technical Debt Classificationn

Delivering high-quality, bug-free software is the goal of all software projects. When programmers are limited by time or other resources, the code they deliver is either incomplete, requires rework, produces errors, or is a temporary workaround. Such incomplete or temporary workarounds are commonly referred to as "technical debt" [35, 36, 37] at the cost of paying a higher price later on. Self-admitted technical debt (SATD) is common in software projects and can have a negative impact on software maintenance. Therefore, identifying SATD is very important for software engineering and needs to be investigated. Potdar and Shihab [35] identified SATD by studying source code comments of four projects and manually devising an approach of 62 patterns. As manually designed patterns have significant drawbacks such as less generality and some physical burden, Huang et al. [38] investigated a text mining based approach that combines multiple classifiers to detect SATD in source comments of target software projects. Since various characteristics of SATD features in code comments, such as vocabulary diversity, project uniqueness, length, and semantic variations, pose a notable challenge to the accuracy of pattern or traditional text mining-based SATD detection, Ren et al. [39] propose a convolutional neural network (CNN)-based approach for classifying code comments as SATD or non-SATD. To avoid the daunting manual effort of extracting features, Wang et al. [40] leverage attention-based neural networks to detect SATD. To take the advantage of large language models and further boost the performance, Liu et al. [17] devised a novel approach to identify SATD by leveraging CodeBERT and fine-tuning paradigm.

2.2.4 Code Smell Classification

Kent Beck defined the term "code smell" in the context of identifying quality problems in code that can be refactored to improve the maintainability of software [41], caused by design flaws or bad programmer habits. Previous works have investigated various methods for identifying code smells in source code, and among them, machine learning is an effective approach for code smell classification. Fontana et al. [42, 43] studied 16 different machine learning methods on four code smells (Data Class, Large Class, Feature Envy, Long Method) and 74 software systems with 1986 manually validated code smell samples. They found that J48 and Random Forest are able to achieve high performance in code smell classification. Das et al. [44] propose a supervised convolutional neural network-based approach to code smell classification by eliminating the effort of manually selecting features. To eliminate the manual effort of feature extraction, several neural network based methods have been proposed in the community. Neural network based methods still require a large number of labeled samples, and Liu et al. [45] proposed an automatic approach to generate labeled training data for neural network based classifiers. Transfer learning can be a reliable solution to the dilemma of better performance and more labeled samples. Sharma et al. [46] propose a transfer learning method involving convolutional neural networks and recurrent neural networks, which can transfer the learned detection ability between C# and Java language. Li et al. [47] propose a hybrid model based on deep learning for multi-label code smell classification, which utilizes both graph convolutional neural networks and bidirectional long short-term memory networks with attention mechanism. BERT-based models have been exploited for code smell classification, where Liu et al. [17] designed a two-step method based on CodeBERT and bidirectional long short-term memory network to check code smell.

2.3 Prompt Learning Method

Prompt-learning method has emerged as the fourth paradigm [19] of natural language processing (NLP) and has drawn enormous attention from the NLP community, which adapts a variety of downstream NLP tasks to pre-trained tasks of large language models. Starting from GPT-3 [18], prompt learning has demonstrated its unique advantage in various downstream NLP tasks, with applications in text classification [48], machine translation [49], etc. Existing researches focusing on prompt learning consists of three main components: a pre-traind language model, a prompt template and a verbalizer, in which research related to verbalizer is orthogonal to our study. Previous studies related to PLM have broadly focused on PLM architectures, including BERT [5], RoBERTa [6], GPT-3, Bart [50], and others. Recently, several PLM related studies have focused on the software engineering domain, with CodeBERT [13] and GraphCodeBERT [12] being typical works. As a generative PLM, CodeT5 [11] benefits a broad set of source code related tasks. A prompt template is used as a container to wrap the input text sequence into a prompt, which is then fed into a PLM to motivate the PLM to recall the rich knowledge associated with the input information. Templates can be constructed in a handcrafted manner [51, 52] and achieve remarkable performance on a variety of downstream NLP tasks. To avoid the onerous effort of manually constructing a prompt template, some researchers seek to construct prompt templates automatically. The automatically generated templates are of two types: discrete templates and continuous templates. The MINE approach [53] is a mining-based discrete approach to automatically find templates given a set of training inputs x and outputs y. Jiang et al. [53] leverage round-trip

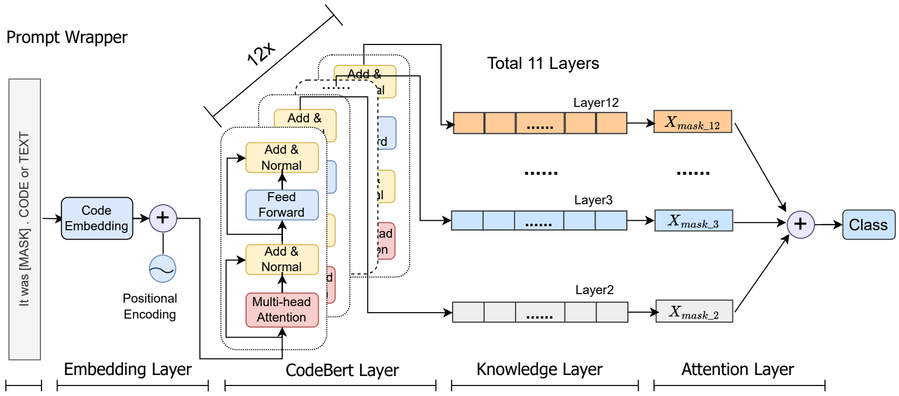

Fig. 1 The architecture of CodePrompt

translation of the prompt into another language then back to generate new templates. The MINE approach is a mining-based approach to automatically find templates given a set of training inputs x and outputs y. Prefix Tuning [54] is a method that can be applied to continuous templates. The technique involves adding a sequence of task-specific vectors as prefixes to the input while keeping the parameters of the pre-trained language model (PLM) frozen.

2.4 Novelty of Our Study

The proposed method represents a departure from existing approaches that rely solely on either the '[CLS]' location of the last layer of output information or multiple layers of BERT-based models to extract linguistic features [17, 16]. Instead, CodePrompt leverages knowledge features induced from a large language model by prompt information to successfully complete source code-related tasks, while also reducing computational consumption during execution. To the best of our knowledge, this approach has not been explored in the field of software engineering. Specifically, CodePrompt leverages prompt learning to encourage a pre-trained language model to output knowledge information associated with an input sequence. To enable the extraction of more implicit feature information, this is achieved by directly combining relevant multiple layers of knowledge from the output of a PLM, computing attention values for these layers, and classifying the aggregated value into a specific category without the need for extra feature-extracting layers.

3 CodePrompt

Our proposed approach, named CodePrompt, leverages multiple aspects of knowledge recalled by a PLM to facilitate source code-related classification tasks. The knowledge contained in each layer of CodeBERT output has its own higherarchy [16] and is regards as an aspect of knowledge. The model architecture of our CodePrompt approach is shown in Fig. 1, which consists of a prompt wrapper, an embedding layer, a CoderBERT layer, a knowledge layer, and an attention layer.

3.1 Prompt Wrapper

The prompt wrapper is designed to wrap an input sequence into a prompt template as a prompt, which is then inputted into a PLM. A effective prompt can induce the PLM to output relevant knowledge related to the input sequence.

In prompt learning, an input sequence $\mathbf{x} = \{w_1, w_2, ...w_n\}$ need to first be wrapped into a prompt template

$$\mathbf{x}_p = \texttt{[CLS]} It \quad was \quad \texttt{[MASK].} \quad \mathbf{x} \tag{1}$$

as a prompt, then fed into a PLM.

3.2 Embedding Layer

This layer is aimed to capture the relationship between tokens which maps target input from a textual form to a vector representation on a low-dimentional dense space. To the input a prompt to the model, the wrapped prompt text sequence is first tokenized by word-piece algorithm then we obtain the sequence $\mathbf{x} = (x_1, x_2, ..., x_n)$, where n is the length of the input sequence, x_n is the n-th tokenized sub-word. Since the length of the input sequence varies for different inputs, we need to pad them to a uniform length to facilitate subsequent processing of the CodeBERT model. For the set maximum length of input sequence N, if the length of sequences is less than N, we pad 0 to the end of these sequences to make their length equal to N. For sequences whose length is greater than N, we directly truncate redundant text sequence at the end. Therefore, the output of the embedding layer is $\mathcal{X} = (X_1, X_2, ...X_N)$. An input text is not merely a combination of tokens, it has important order information. To make the model to leverage the order information, absolute positional encoding (APE) is added in $\mathcal{X}$, the final output to feed into the next layer is

$$\mathcal{X} = \mathcal{X} + APE(\mathcal{X}), \mathcal{X} \in \mathbb{R}^{N \times d_{model}} \tag{2}$$

, where d_{model} is the size of the embedding dimension.

3.3 CodeBERT Layer

In this layer, unlike other works, we leverage the knowledge information output from a masked language model. The embedding vectors from one embedding layer are fed into the layer to motivate the layer to output the knowledge information stored during the pre-training phase. The motivated knowledge information is leveraged as knolwedge features of input sequences. CodeBERT is a bi-modal pre-trained language model based on transformers for both programming languages (PL) and natural langauges (NL) [10]. CodeBERT is pre-trained on a large general-purpose corpus by two tasks: Masked Language Model (MLM) and Replaced Token Detection (RTD). Specifically, the MLM task targets bimodal data by simultaneously feeding the code with the corresponding comments and randomly selecting positions for masking, then replacing the token with a special '[MASK]' token, the goal of the MLM task is to predict the original token. The RTD task targets unimodal data with separate codes and comments, randomly replaces the token, and aims to learn whether the token is the original word using a discriminator [10]. However, in a large language model,such as CodeBERT, the output of '[MASK]' location is representing not only the original token but also the relavent knowledge about input information.

3.4 Knowledge Layer

Previous studies leverage only the vector of one or more layers at '[CLS]' location [17]. According to Choi et al. [55], the output of '[CLS]' location is not the best choice for the representation of an input sequence, which are not stable for downstream tasks. Jawahar et al. [16] proposed that each layer of the output of a BERT-like model has different semantic features that can be combined to further extract higher-level features that better represent the input. Unlike these studies, we leverage knowledge information from the output of CodeBERT. As described in the prior subsection, the output of '[MASK]' location includes knowledge information related to the input, in which each layer has different aspect of knowledge and different importance related to the input. The output of the CodeBERT model has 13 layers, where one embedding layer and 12 layers for the encoders. The lowest-level embedding has little knowledge from the whole input, and the remaining layers have different importance. Through pilot experiments, the outputs from layer 2 through layer 12 have been chosen as knowledge sources.

$$\mathcal{X}_{know} = \begin{bmatrix} \mathbf{x}_{mask\_2} \\ \mathbf{x}_{mask\_3} \\ \cdots \\ \mathbf{x}_{mask\_12} \end{bmatrix}, \mathcal{X}_{know} \in \mathbb{R}^{11 \times d_{model}} \tag{3}$$

, where d_{model} is the size of the embedding dimension.

3.5 Attention Layer

Several knowledge features were obtained in the previous subsection, not all representational information contributes equally to the input, and each layer of knowledge features has different weights for the entire representation of the input. Some source code related tasks focus more on lower-level knowledge features, while others focus more on higher-level features. Therefore, in CodePrompt, the attention mechanism is used to compute different weights for each knowledge feature. Specifically, we first compute the tanh value of $\mathcal{X}_{know}$ as

$$\mathbf{u}_i = tanh(\mathcal{X}_{know}) \tag{4}$$

, then the similarity between $\mathbf{u}_i$ and the context vector $\mathbf{u}_w$ can be calculated and transformed into a probability distribution by Softmax.

$$\alpha_i = \frac{\exp\left(\mathbf{u}_i^T \mathbf{u}_w\right)}{\sum_i \mathbf{u}_i^T \mathbf{u}_w} \tag{5}$$

α_i can be treated as the importance of the input for each leavel of knowledge feature, therefore using α_i as a global weighted summation over $\mathcal{X}_{know}$ can generate the input vector $\mathbf{x}_{out}$,

$$\mathbf{x}_{out} = \sum_i \alpha_i \mathbf{x}_i \tag{6}$$

Finally, for the $\mathbf{x}_{out}$, it can be classified by a layer of fully connected feed-forward network.

$$\mathbf{p}(y|\mathbf{x}_{out}) = \mathbf{w}(gelu(\mathbf{x}_{out})) + b \tag{7}$$

The final class label y of the input is

$$\hat{y} = \arg\max \mathbf{p}(y|\mathbf{x}_{out}) \tag{8}$$

4 Experimental Study

In this section, we design four source code-related tasks aimed at answering the following questions (RQs).

RQ1: Can our proposed approach achieve state-of-the-art results for source code-related tasks without the need for additional feature extraction layers, relative to baselines?

RQ2: Can the performance of CodePrompt be enhanced by solely utilizing either the attention mechanism or a prompt template?

RQ3: How does the attention mechanism work in the four source code-related classification tasks?

RQ4: Does the proposed model perform equally well on both programming language-based and natural language-based tasks?

RQ5: What is the potential time-saving that could be obtained by removing additional neural network layers?

RQ1 aims to validate the performance of CodePrompt on four source code-related classification tasks. To achieve this goal, we conducted extensive experiments and compared CodePrompt with current state-of-the-art baselines. In RQ2, we investigated the effectiveness of different components in the

CodeBERT-based approach on performance. To answer these questions, we performed comprehensive ablation studies on four datasets. RQ3 examines the differences in focusing on different layers of knowledge across four different tasks. RQ4 addresses the question of whether knowledge features are equally effective for both natural language-based and programming language-based tasks. Lastly, RQ5 investigates the potential time-saving that could be obtained by removing additional neural network layers in a CodeBERT-based pipeline.

4.1 Tasks and Datasets

To validate our proposed approach, we conducted extensive experiments on four downstream tasks related to source code. The first two tasks (code language classification and code smell classification) are related to programming language processing, while the other two (code comment classification and technical debt classification) are related to natural language processing. Each task has a dedicated dataset, which is described in detail in the corresponding task description.

4.1.1 Code Language Classification

In this task, the target data are source code snippets. We leverage the publicly shared dataset in SC++ [3], which collects 21 programming languages popular in the Stack Overflow community, based on the 2017 Stack Overflow developer survey [2]. As is inevitable, there is some invalid source code in the dataset, which we removed using tools from DeepSCC [31], keeping 19 programming languages. In other words, this task is a multi-class source code task that predicts the programming language type of each code snippet.

4.1.2 Code Smell Classification

Code smell classification is a binary classification task where a model needs to decide whether it has code smell or not, and its target is the source of a variety of code snippets. Fakhoury et al. [56] build a corpus which selects 4205 lines of source code from 13 java open-source systems to avoid domain-specific dependencies of the results. The corpus has over 1700 labeled code snippets, which is in line with the taxonomy of linguistic smells presented in the paper [57]. The corpus is a typical dataset for code smell classification in previous studies [17], and we adopt it for this task.

4.1.3 Code Comment Classification

Code comments are crucial software components that contain important information concerning software design, code implementation, and other technical

[2] https://insights.stackoverflow.com/survey/2017#technology

Table 1 Corpus statistics for four source code-related tasks

Task	Class Num	Train	Test	Avg	Mode	Median	<32	<64	<128	<256	<300
Code Smell	2	1,399	350	41.97%	8	20.0	68.19%	85.56%	93.14%	97.63%	98.43%
Code Language	19	179,556	44,889	58.58%	2	31.0	51.04%	73.76%	89.78%	96.94%	97.75%
Technical Debt	2	31,708	6,652	9.27%	3	6.0	95.98%	99.23%	99.87%	99.98%	99.99%
Code Comment	16	8,985	2,247	15.82%	2	7.0	88.46%	93.88%	97.12%	99.97%	99.99%

details. We adopt the corpus shared by Pascarella and Bacchelli [58], which has over 11, 000 code reviews and 16 classes from six java open source software projects. This task is a multi-class natural language classification task that aims to categorize each comment into a specific class.

4.1.4 Technical Debt Classification

Techincal debt classification is a natural language classification task based on annotated information indicating whether there is technical debt or not. Our goal is to detect technical debt annotated by programmers, i.e., self-admitted techical debt (SATD). The dataset presented by Maldonado et al. [59] consists of approximately 10, 000 code comments collected from 10 open source projects, which are classified into five types of SATD, namely, design debt, requirement debt, defect debt, documentation debt, or test debt. All our SATD experiments are performed on the corpus.

For brevity, in the following sections, code language classification is abbreviated as code language, code smell classification as code smell, code comment classification as code comment, and technical debt classification as technical debt.

For a fair comparison with current state-of-the-art results, we adopt stratified sampling to divide the corpus into a training set and a test set in an 80:20 percent ratio for each task. These statistics consist of (1) the number of training and test sets, (2) the mean, mode, and median of the code/comment length, and (3) the percentage of samples with sizes <32, <64, <128, <256, and <300. All statistical information for the four datasets is detailed in Table 1.

4.2 Evaluation Metrics

Accuracy, Precision, Recall, and F1-Score are chosen as evaluation metrics for the binary classification task. These evaluation metrics are calculated as follows.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

$$Precision = \frac{TP}{TP + FP} \tag{10}$$

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

$$F1\text{-}Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{12}$$

where TP means that a positive sample is predicted as a positive class, TN means that a negative sample is predicted as a negative class, FP means that a negative sample is assigned a positive label, and FN means that a positive sample is predicted as a negative class.

For all multi-classification tasks, we leverage the macro approach to compute evaluation metrics. Specifically, we tally TP, FP, FN and TN for each class and then compute Precision, Recall and F1-Score, respectively. Finally, we obtain the mean value of each metric for all classes to obtain Macro-Precision, Macro-Recall, and Macro-F1-Score.

For simplicity, Accuracy is abbreviated as ACC, Precision as P, Recall as R, and F1-Score as F1 in the following tables.

4.3 Baselines

Previous studies have extensively investigated source code-related tasks [60], utilizing a range of AI-based tools from machine learning to deep learning. In this paper, we compare our CodePrompt model with eight state-of-the-art baselines for four source code-related classification tasks. These baselines can be classified into two categories from the perspective of AI development: machine learning-based approaches (traditional machine learning) and neural network-based approaches. The neural network-based approaches can be further categorized into two sub-classes: classical neural network-based approaches and pre-trained language model-based methods. The following is a brief introduction to all the baselines we compare with: Random Forest, XGBoost, TextCNN, AttBLSTM, BERT, RoBERTa, CodeBERT, and EL-CodeBERT.

4.3.1 Machine Learning based Methods

Random Forest is an ensemble machine learning algorithm based on decision trees and bagging proposed by Breiman [2], and the baseline experiments were implemented with the library scikit-learn[3].
XGBoost is sugguest by Chen and Guestrin [61], which is a scalable end-to-end tree boosting system and is used widely by data scientists to achieve state-of-the-art results on many machine learning challenges. It is sparsity-aware algorithm for sparse data and weighted quantile sketch for approximate tree learning. We adopt the offiical implementation from the original author on github[4].

4.3.2 Neural Networks based Methods

Classical Neural Netowrk based Approaches

[3] https://github.com/scikit- learn/scikit-learn
[4] https://github.com/dmlc/xgboost

TextCNN Kim [62] proposed TextCNN, a sophisticated approach that leverages convolutional neural networks for natural language processing. In this study, we adopt a classical implementation of TextCNN as a baseline[5].

AttBLSTM Zhou et al. [63] proposed AttBLSTM, a combined approach that leverages attention mechanism and bidirectional long short-term memory network (BiLSTM) to capture important semantic features of textual sequences. In this study, we utilize a baseline implementation of AttBLSTM based on source code available on Github[6].

Pre-trained Model based Methods

BERT [5] is a deep bidirectional transformer-based language model for language understanding, pre-trained on both next sentence prediction and masked language modeling tasks using self-supervised methods with large-scale corpora. In this study, we utilize a baseline implementation of BERT based on the BERT-base model[7].

RoBERTa [6] is an enhanced BERT-based model pre-trained on a much larger corpus than the original BERT model[8] using only masked language modeling approach. In this study, we utilize a baseline implementation of sequence classification based on the RoBERTa-base model, which is the official implementation provided by the authors[9].

CodeBERT [10] is a transformer-based model pre-trained on a corpus consisting of both programming language and natural language, using both masked language modeling and replaced token detection tasks. In this study, we utilize a baseline implementation of CodeBERT based on the CodeBERT-base model.

EL-CodeBERT [17] is a two-stage model that builds upon CodeBERT, attention mechanism, and BiLSTM. The model utilizes BiLSTM to extract multiple layers of semantic features and then employs attention mechanism to aggregate the final features, resulting in promising performance compared to other baselines. To ensure consistency with the baseline implementation, we adopt the official code and training method[10].

4.4 Experimental Settings

Our proposed approach utilizes Openprompt [64], an open-source framework for prompt learning, along with the CodeBERT-base model [10]. The prompt templates used for all datasets are listed in Table 2. To ensure consistent and fair comparison, we conducted all experiments using an Nvidia RTX3090 GPU, Linux operating system (Ubuntu 22.04), and 64GB of system memory for both the baselines and our proposed method.

[5] https://github.com/NTDXYG/Text-Classify-based-pytorch/blob/master/model/TextCNN.py

[6] https://github.com/NTDXYG/Text-Classify-based-pytorch/blob/master/model/TextRNN_Attention.py

[7] https://huggingface.co/transformers/v3.0.2/model_doc/bert.html#bertforsequenceclassification

[8] https://huggingface.co/roberta-base

[9] https://huggingface.co/docs/transformers/model_doc/roberta#transformers.RobertaForSequenceClassification

[10] https://github.com/NTDXYG/ELCodeBert

Table 2 Prompt templates for four tasks

Task	Prompt Template
Code Language	"Just [MASK] ! **x**"
Code Smell	"It was [MASK] . **x**"
Code Comment	"Just [MASK] ! **x**"
Technical Debt	"**x** In summary , it was [MASK] ."

Table 3 Results on code language classification task

Method	ACC(%)	P(%)	R(%)	F1(%)
Random Forest	78.728	79.362	78.825	78.874
XGBoost	78.803	79.925	78.891	79.217
TextCNN*	82.662	83.561	82.706	82.964
AttBLSTM*	79.035	79.801	79.107	79.272
BERT	86.865	87.129	86.938	86.985
RoBERTa	87.202	87.424	87.276	87.135
CodeBERT	87.418	88.042	87.450	87.614
EL-CodeBERT(2022)	87.959	88.177	88.023	88.077
CodePrompt	**88.024**	**88.232**	**88.091**	**88.149**

5 Results and Analysis

This section presents the evaluation and analysis of our proposed approach, CodePrompt. We will commence by comparing our results with the baselines on four source code-related tasks to demonstrate the performance of our approach. Following that, we will experimentally validate the effectiveness of each component in our approach. We will then conduct a series of attention-related experiments to demonstrate how different tasks focus on different knowledge layers. Next, we will analyze the varying effects of knowledge features on programming language-based and natural language-based tasks. Finally, we will perform a set of experiments to exhibit the time-saving potential of CodePrompt.

5.1 Result Analysis for RQ1

In Table 3, 4, 5, and 6, the first two rows correspond to machine learning approaches, the next two rows marked with an asterisk (*) are classical neural network approaches without pre-trained language models, and the next four rows are pre-trained model-based approaches. The number in parentheses following a method name denotes the year the method was proposed.

Code Language Classification. Table 3 presents the results of the comparison between CodePrompt and all baselines. We observe that pre-trained

Table 4 Results on code smell classification task

Method	ACC (%)	P(%)	R(%)	F1(%)
Random Forest	78.286	78.880	77.548	77.756
XGBoost	75.714	75.667	75.305	75.409
TextCNN*	80.000	80.016	79.641	79.761
AttBLSTM*	78.857	78.810	78.537	78.631
BERT	79.714	79.580	79.653	79.610
RoBERTa	81.143	81.014	81.067	81.038
CodeBERT	85.429	85.516	85.128	85.264
EL-CodeBERT(2022)	**86.000**	85.990	**85.795**	**85.874**
CodePrompt	**86.000**	**86.167**	85.657	85.824

model-based methods achieve better performance than machine learning and classical neural networks, with clear advantages.

Our CodePrompt approach outperformed all baselines on all four evaluation metrics, achieving accuracy, precision, recall, and F1-score values of 88.024%, 88.232%, 88.091%, and 88.149%, respectively. These results provide strong evidence that the combination of knowledge features and the attention mechanism can significantly enhance the performance of source code-related classification tasks. Notably, the use of knowledge features inspired by prompts proved to be effective in better capturing the features of the input information.

Code Smell Classification. The results of comparison between CodePromt and all baselines are listed in Table 4. Among all the methods evaluated in this task, the pre-trained model-based approach achieved the best performance. However, our CodePrompt method achieved the highest accuracy and precision values of 86.000% and 86.167%, respectively, outperforming all other methods on these two metrics. Furthermore, CodePrompt achieved comparable results on the remaining two metrics, with recall and F1-score values of 85.657% and 85.824%, respectively. These results demonstrate that CodePrompt can achieve comparable performance to EL-CodeBERT with lower computational cost. This may be attributed to the superior feature extraction performance of the CodePrompt method on programming language processing tasks.

Code Comment Classification. Table 5 presents the results of the comparison between CodePrompt and all baselines. The table shows that CodePrompt achieved the highest accuracy value of 95.416%, outperforming all other methods on this metric. In addition, CodePrompt achieved a better precision value of 89.654% compared to EL-CodeBERT, and achieved comparable recall and F1-score values of 87.036% and 87.930%, respectively. These results demonstrate that CodePrompt has notable performance on code comment classification, suggesting its promising capability of feature extraction on natural language-based content. Notably, all of these results were achieved with lower computational consumption than EL-CodeBERT.

Table 5 Results on code comment classification task

Method	ACC(%)	P(%)	R(%)	F1(%)
Random Forest	90.921	83.783	75.104	74.618
XGBoost	90.565	77.561	68.661	71.645
TextCNN*	91.945	87.541	78.596	80.977
AttBLSTM*	92.345	85.578	78.268	80.596
BERT	94.482	87.149	83.935	85.275
RoBERTa	94.393	**90.525**	86.121	86.875
CodeBERT	94.838	87.916	86.301	86.820
EL-CodeBERT(2022)	95.238	89.395	**87.280**	**87.977**
CodePrompt	**95.416**	89.654	87.036	87.930

Table 6 Results on technical debt classification task

Method	ACC(%)	P(%)	R(%)	F1(%)
Random Forest	97.278	**95.080**	87.019	90.564
XGBoost	97.294	93.901	88.516	90.991
TextCNN*	96.978	93.015	87.258	89.881
AttBLSTM*	97.114	93.979	87.177	90.227
BERT	96.783	91.229	88.004	89.534
RoBERTa	97.595	93.352	91.110	92.279
CodeBERT	97.835	94.197	91.991	93.059
EL-CodeBERT(2022)	97.850	94.024	**92.310**	93.146
CodePrompt	**97.895**	94.330	92.257	**93.263**

Technical Debt Classification. Table 6 presents the results of the comparison between CodePrompt and all baselines. The table shows that all approaches, from machine learning to pre-trained model-based methods, achieved similar results in terms of accuracy. However, our CodePrompt method achieved the highest accuracy and F1-score values of 97.895% and 93.263%, respectively. Moreover, CodePrompt achieved the second-best performance on metric Precision and Recall, with a Precision value of 94.330%, which outperformed EL-CodeBERT's Precision value of 94.024%. These results demonstrate the excellent performance of CodePrompt on code comment classification tasks, which may be attributed to its superior feature extraction capability on natural language content.

Summary for RQ1: CodePrompt outperforms all previous state-of-the-art baselines in terms of accuracy for all four source code-related tasks. The approach exhibits clear advantages on tasks with lower metric values, such as code language classification and code smell classification. These results demonstrate the superior performance of CodePrompt with lower computational cost compared to previous state-of-the-art baselines. The main reason for this is

Table 7 Results of the ablation study. "w/o attention" refers to the absence of the attention mechanism. The task marked with an asterisk (*) is a programming language task, while others are natural language tasks. "w/o Attention&Prompt" denotes the absence of both the attention mechanism and prompt templates, equivalent to a classifier based on fully connected feed-forward networks and CodeBERT.

Task	Method	ACC (%)	P(%)	R(%)	F1(%)
Code Language*	CodePrompt	**88.024**	**88.232**	**88.091**	**88.149**
	w/o Attention	87.828	88.010	87.926	87.958
	w/o Attention&Prompt	87.418	88.042	87.450	87.614
Code Smell*	CodePrompt	**86.000**	**86.167**	**85.657**	**85.824**
	w/o Attention	82.286	82.407	81.900	82.051
	w/o Attention&Prompt	85.429	85.516	85.128	85.264
Code Commet	CodePrompt	**95.416**	89.654	**87.036**	**87.930**
	w/o Attention	95.016	**92.360**	86.328	87.797
	w/o Attention&Prompt	94.838	87.916	86.301	86.820
Techical Debt	CodePrompt	**97.895**	**94.330**	92.257	**93.263**
	w/o Attention	97.745	92.677	**93.337**	93.004
	w/o Attention&Prompt	97.835	94.197	91.991	93.059

the powerful capability of CodePrompt's feature extraction on both programming language and natural language, without the need for additional neural network layers to enhance feature extraction.

5.2 Result Analysis for RQ2

RQ2 aims to investigate the contribution of the attention and prompt components in our proposed approach. To demonstrate their contribution, we conducted extensive ablation experiments on four source code-related tasks, keeping the same experimental setup except for replacing the attention mechanism with a fully connected network or removing the prompt templates. Table 7 presents a comparison of the experimental results. From the table, we observed that replacing the attention mechanism with a fully connected neural network results in worse performance compared to using the attention mechanism on all four tasks, particularly on programming language tasks, such as code classification and code smell classification. For the code comment classification and technical debt classification tasks, some individual evaluation metrics show slight improvement, except for accuracy. After removing both the attention and prompt components, the model degenerates into a classifier based on fully connected feed-forward networks and CodeBERT, and its performance is worse on all metrics than CodePrompt.

Comparing the last two rows in each task, we observed that utilizing the prompt template alone (denoted as "w/o Attention") does not improve performance in code smell classification (Accuracy value drops from 85.429% to 82.386%), and its contribution does not provide an explicit advantage in the other tasks. However, combining the prompt template with the attention mechanism can significantly enhance performance on all tasks. The results of our proposed CodePrompt approach strongly demonstrate its effectiveness.

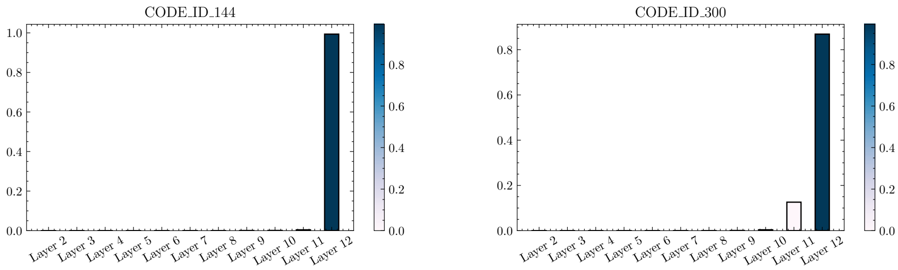

Fig. 2 Attention values at each layer for code language classification

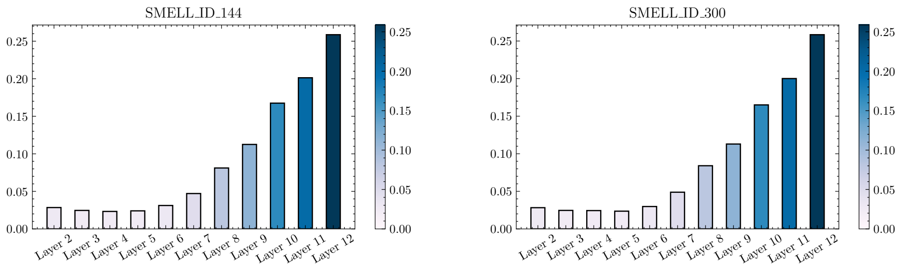

Fig. 3 Attention values at each layer for code smell classification

Comparing code language classification with code smell classification, we observed that the attention mechanism has a greater advantage over fully connected feed-forward networks (denoted as "w/o Attention&Prompt") in code smell classification. The reason for this difference is that code smell requires the detection of semantic differences between code snippets of the same type, while code language classification involves detecting linguistic features of different programming languages. From the results, it appears that semantic features can be detected more easily than linguistic features.

Summary for RQ2: Our proposed CodePrompt approach, which incorporates prompt templates and the attention mechanism to avoid additional computation costs associated with additional neural network layers, significantly improves performance on the four source code-related tasks compared to fully connected feed-forward networks. Both the attention and prompt components are valid and indispensable for achieving the best performance.

5.3 Result Analysis for RQ3

RQ3 aims to investigate the layers with decisive attention values that each task focuses on. We selected two examples from each task to examine the attention values of each layer. The details of each sample are lists in Table 9. The results are presented in Figures 2, 3, 4, and 5.

Code Language Classification. As shown in Figure 2, the concentration of attention in code language classification tasks is mainly on the last two

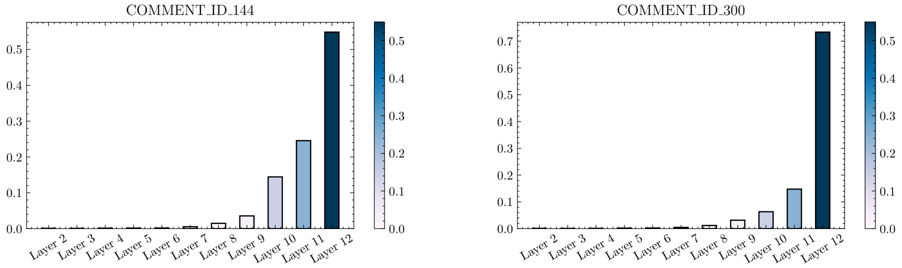

Fig. 4 Attention values at each layer for code comment classification.

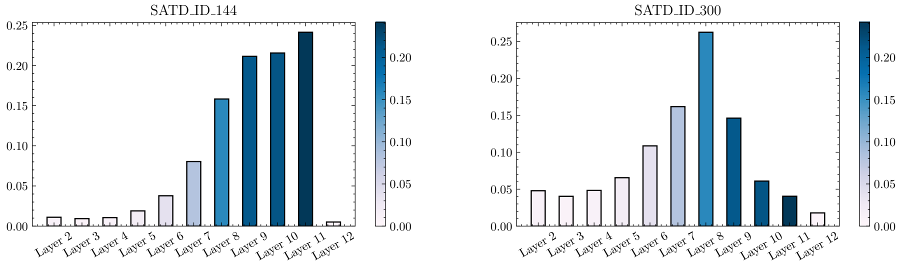

Fig. 5 Attention values at each layer for technical debt classification

layers of knowledge, particularly the final layer. For instance, in the case of id 144, the attention value focused on the last layer is 99.37%, while that on layer 11 is only 0.42%. Similarly, for id 300, attention on the last layers is 86.90%, whereas that on layer 11 is 12.60%. According to [65], the last layer represents the highest level of knowledge regarding the input. As the code language classification task aims to distinguish the type of programming languages, the highest level of knowledge contains sufficient identifying features to accomplish this task.

Code Smell Classification. As shown in Figure 3, code smell classification tasks focus on each layer of knowledge derived from a pre-trained model, but there is a higher concentration on the last few layers. For example, in the case of id 144, the attention values are 25.87% on the final layer and 20.13% on layer 11. The objective of code smell classification is to detect coding style and semantic information, and it relies on the features of knowledge that range from low to high levels.

Code Comment Classification. As illustrated in Figure 4, this task primarily focuses on the last four layers of knowledge, with a stronger emphasis on the last three layers. For instance, for id 144, the attention values for layer 12, 11, and 10 are 54.84%, 24.55%, and 14.44%, respectively. The last layer, which contains the highest-level knowledge, is particularly crucial for this task. Code comment classification is a natural language processing task that places a significant emphasis on differences in semantic information.

Technical Debt Classification. As shown in Figure 5, technical debt classification focuses more on the middle layers of knowledge rather than the last

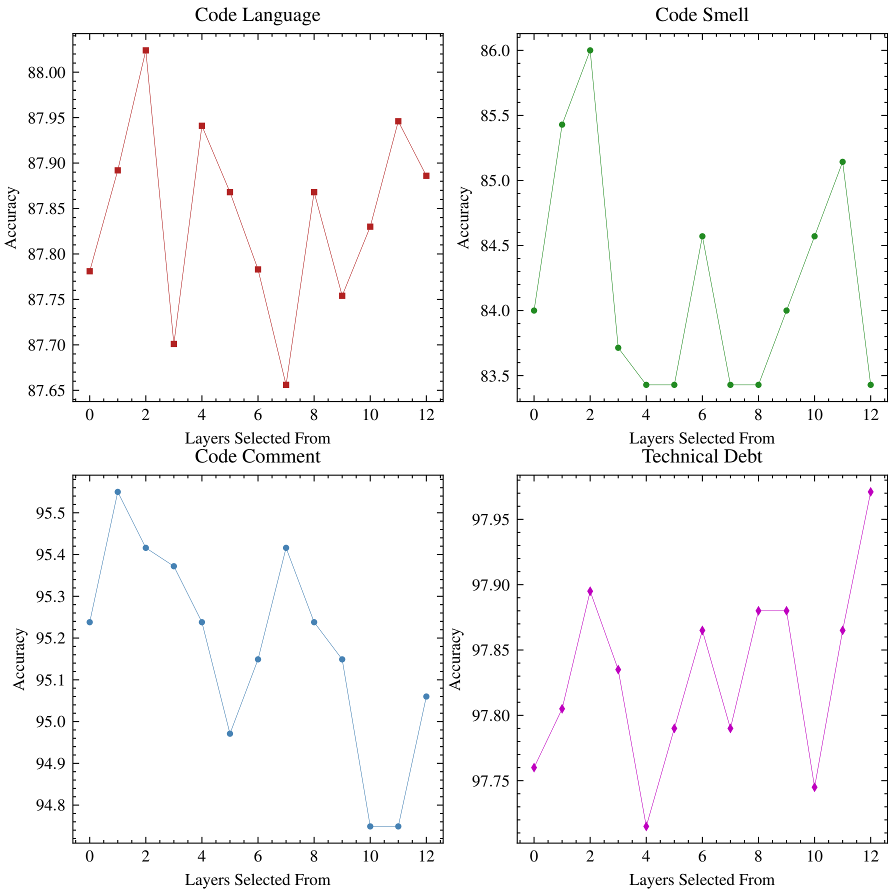

Fig. 6 Results for the metric Accuracy

layers. For instance, in the case of id 144, it concentrates more on layer 7, 8, 9, 10, and 11, while for the last layer, the attention value is only 0.51%. A similar trend can be observed for id 300. This suggests that the highest layer of knowledge contributes less to technical debt classification than other layers. This task is a binary classification task, similar to emotional classification, and does not involve concrete semantic information.

Summary for RQ3: Various source code-related tasks demonstrate different degrees of attention on each layer of output knowledge from the CodeBERT model. This indicates that different levels of knowledge hold distinct meanings for different tasks, highlighting the effectiveness of the attention mechanism for source code-related tasks.

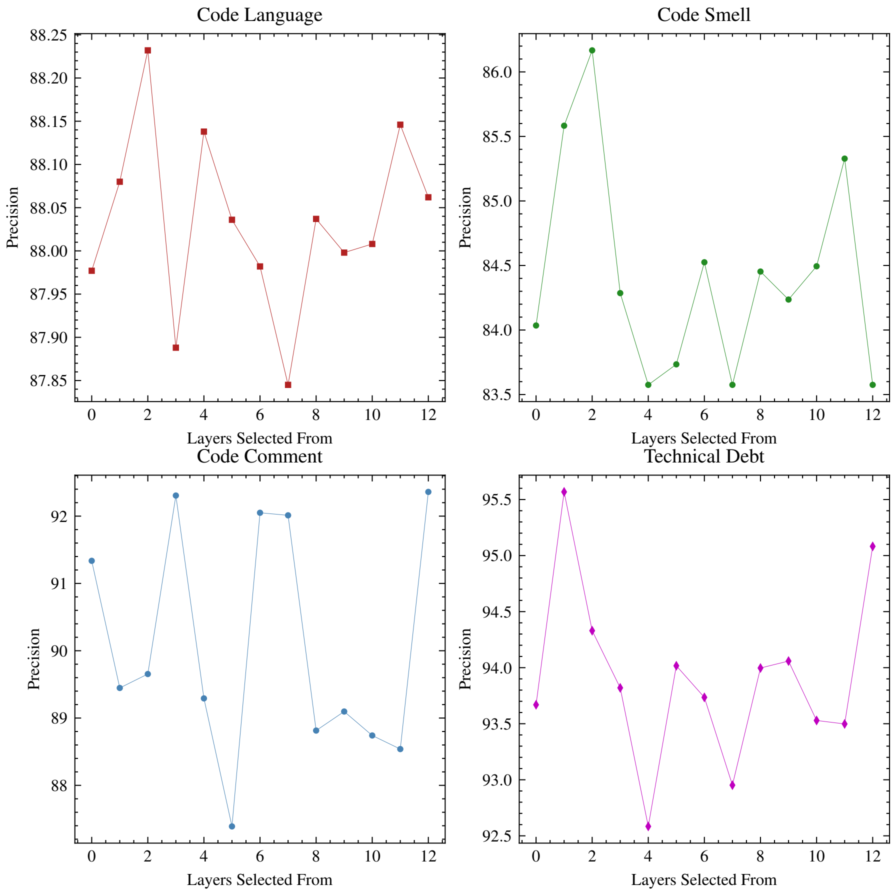

Fig. 7 Results for the metric Precision

5.4 Result Analysis for RQ4

The hidden state output of the BERT-based model consists of 13 layers (from layer 0 to layer 12), where layer 0 represents the embedding vector of the input information, and the other layers contain hierarchical linguistic features [16]. Our experiments aim to investigate the impact of various layers of knowledge features on source code-related classification tasks. To this end, we collected 13 sets of knowledge features for each task, with the first set ranging from layer 0 through layer 12, the second set from layer 1 through layer 12, and so on.

The four tasks can be grouped into two categories: programming language-based tasks (code language classification and code smell classification) and natural language-based tasks (code comment classification and technical debt classification). The experiment results for four metrics, namely Accuracy, Precision, Recall, and F1-score, are presented in Figure 6 through Figure 9.

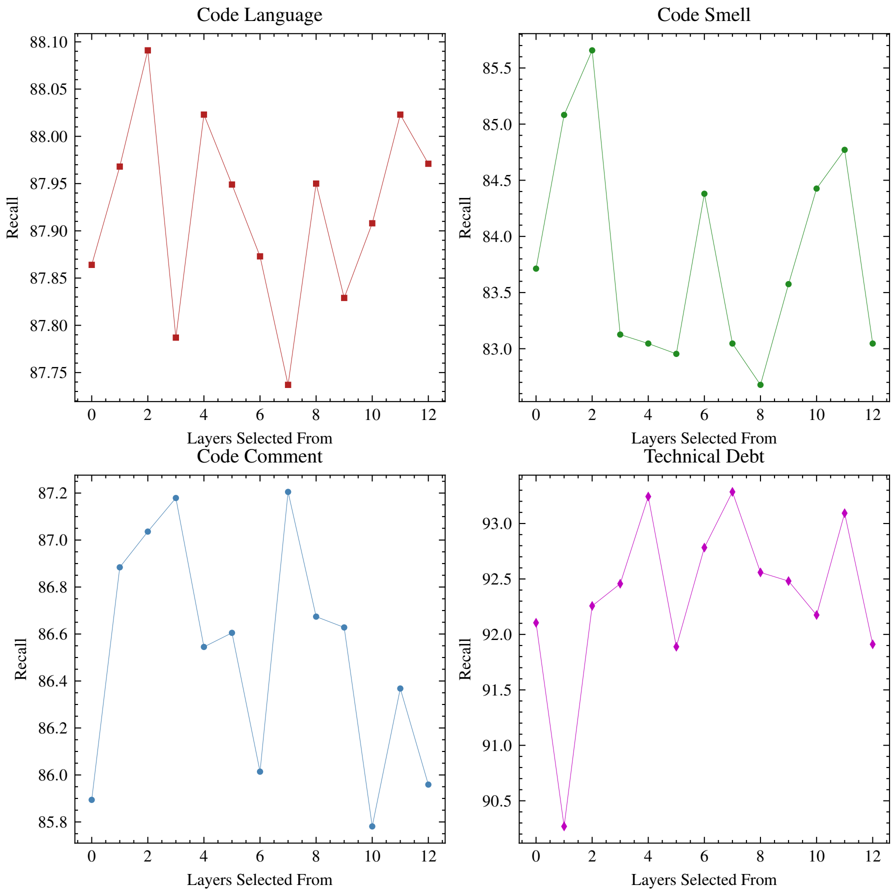

Fig. 8 Results for the metric Recall

From these figures, it is observed that the CodePrompt approach consistently achieves the best results on the third set of knowledge features (from layer 2 through layer 12) across all evaluation metrics for programming language-based tasks. These results demonstrate that the attention mechanism on knowledge features from layer 2 through layer 12 is better able to represent features on the programming language-based tasks. For natural language processing tasks, our CodePrompt approach on this set of features (from layer 2 through layer 12) still obtained better results than the baselines. To ensure uniformity across tasks, we selected this set of knowledge features for all four source code-related tasks.

Summary for RQ4: Our CodePrompt approach achieves greater performance on both programming language-related tasks and natural language-based tasks, with higher stability particularly for programming language-based source code-related tasks.

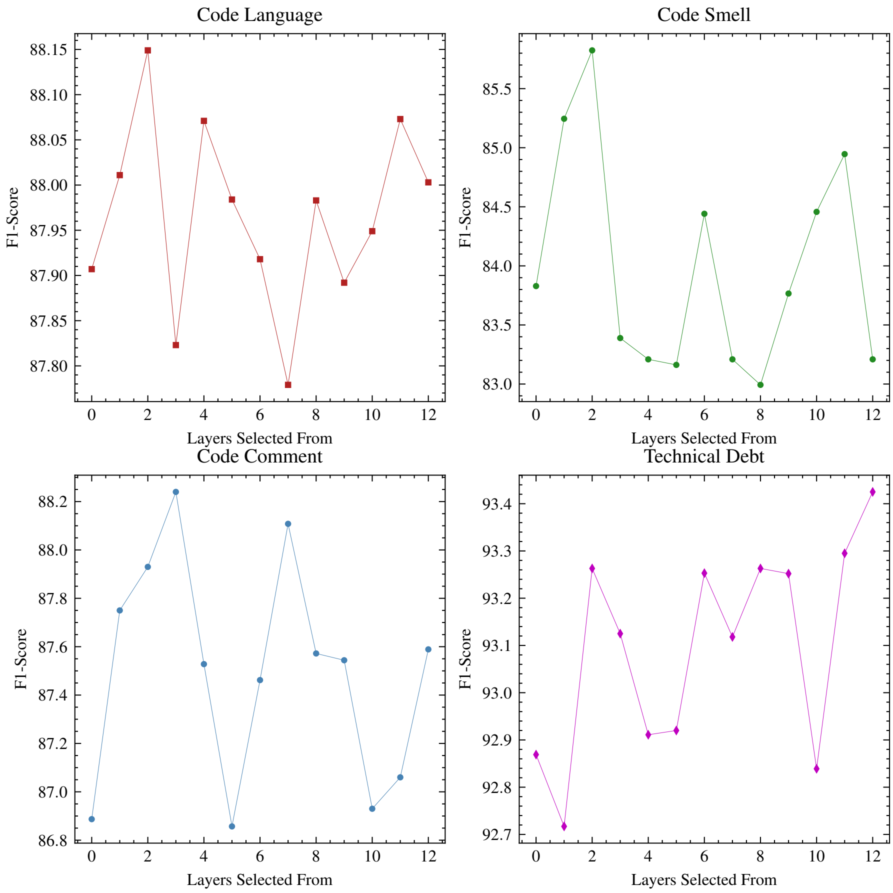

Fig. 9 Results for the metric F1-score

Table 8 Time consumption of an additional LSTM layer in CodeBERT based pipeline.

Task	Max Length	Repeat Times	Total(ms)	LSTM(ms)	Percentage(%)
Code Language	256	10*1000	**6430.27**	**719.15**	**11.18**

5.5 Result Analysis for RQ5

The time consumption experiment aimed to demonstrate the impact of additional neural network layers on a CodeBERT-based classification pipeline. Previous studies [17] have utilized additional BiLSTM layers to extract more effective features and achieve notable performance with the CodeBERT-based pipeline. Using the same hardware and software settings, including the GPU, CPU, memory, and software versions, we selected an example from the dataset of code language classification and conducted ten groups of experiments, each consisting of 1,000 inference repetitions on the example. The resulting average

time consumed by the ten groups is reported in Table 8, where "Total" denotes the time consumption of the entire pipeline, and "LSTM" represents the time cost of the BiLSTM layer in the pipeline. The results show that the BiLSTM layer consumes 11.18% of the time during the entire pipeline.
Summary for RQ5: The experiments suggest that, with the aid of powerful feature extraction capabilities, removing an additional neural network layer can significantly reduce computation time and effectively eliminate unnecessary computation costs.

6 Threats to Validity

In this section, we will focus on discussing potential threats to the validity of our empirical study.

6.1 Interanl Threats

The internal threats to the validity of our research are primarily related to the experimental environment. The first threat arises from the choice of hardware and software platforms, which can impact the stability of our proposed method's implementation. To address this concern, we maintained a stable experimental setup by using consistent hardware and selecting mature versions of software packages such as Pytorch and the Linux operating system. The second threat stems from the implementation of baselines, which we mitigated by selecting source code from publicly available, mature libraries. Lastly, the third threat concerns randomness in the initialization of deep learning models. To ensure the reproducibility of our studies, we set fixed random seeds for all experiments.

6.2 External Threats

The main external threat lies in the choice of datasets used for the four downstream tasks related to source code. To mitigate this threat, we have opted to use publicly available corpora. Specifically, for code language classification, we draw upon the dataset provided by Alrashedy et al. [3]. For code smell classification, we utilize the dataset from Fakhoury et al. [56]. For code comment classification, we rely on the dataset prepared by Pascarella et al. [58]. Lastly, for technical debt classification, we employ the dataset introduced by Maldonado et al. [59, 56].

6.3 Construct Threats

The primary constructive threat we address in this study is the selection of appropriate evaluation metrics for assessing performance on source code-related

Table 9 The text or code lists of attention values. ID denotes the id of a selected example.

Task	ID	Text or Code Snippet
Code Language	144	$\langle\% = Html.RadioButton(''ticketStatus'', ''Open'', true)\%\rangle$
	300	$ClassDocumentVOPrintJobVOjob; PrintRunVOrun; Stringid; gettersandsetters..$
Code Smell	144	$finalint \langle w\rangle\ getAttributes \langle /w\rangle\ (intid)\{\langle w\rangle\ ensureId \langle /w\rangle\ (id) \ldots\}$
	300	$publicvoid \langle w\rangle\ ToSource \langle /w\rangle\ (\langle w\rangle\ StringBuilder \langle /w\rangle\ sb)\{Stringdot = \langle strliteral\rangle \langle str - literal\rangle \ldots\}$;
Code Comment	144	Contributors: * IBM Corporation - initial API and implementation * Bjorn Freeman-Benson
	300	// verify we are in main view and url is correct
Technial Debt	144	//ChangeFactoryImpl
	300	ODO: What does the output directory have to do with the class path? Project p = ... ;

tasks. To ensure a fair and comprehensive comparison, we have selected four widely-used metrics (accuracy, precision, recall, and F1-Score) that have been extensively employed in previous studies, such as the work by Alrashedy et al. [3].

7 Conclusion

This paper proposed a novel approach that leverages knowledge features induced by prompts from a pre-trained model to enhance classification tasks related to source code. Our approach reduced computational costs and eliminated the need for additional neural network layers by aggregating multiple layers of induced knowledge contained in the input as input representation. We improved accuracy by enhancing the input representation through combining important knowledge layers that relate to each task. Our CodePrompt approach achieved new state-of-the-art results on the accuracy metric and comparable results on other metrics, as demonstrated by extensive experiments. We conducted time consumption experiments to demonstrate the computational cost-saving capability of CodePrompt and ablation experiments to validate the effectiveness of the proposed multi-layer attention-based component. Furthermore, we conducted attention observation experiments to demonstrate the role of different layers in each task.

Acknowledgements The work was supported by 242 National Information Security Projects (no.2020A065).

Conflict of interest

The authors declared that they have no conflict of interest.

Data Availability Statements

The data that support the findings of this study are openly available in the github repository at https://github.com/BIT-ENGD/codeprompt.

References

1. Khasnabish JN, Sodhi M, Deshmukh J, Srinivasaraghavan G (2014) Detecting Programming Language from Source Code Using Bayesian Learning Techniques. In: Perner P (ed) Machine Learning and Data Mining in Pattern Recognition, Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 513–522
2. Breiman L (2001) Random Forests. Machine Learning 45(1):5–32
3. Alrashedy K, Dharmaretnam D, German DM, Srinivasan V, Aaron Gulliver T (2020) SCC++: Predicting the programming language of questions and snippets of Stack Overflow. Journal of Systems and Software 162(C)
4. Gilda S (2017) Source code classification using Neural Networks. In: 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp 1–6
5. Devlin J, Chang MW, Lee K, Toutanova K (2019) BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Association for Computational Linguistics, Minneapolis, Minnesota, pp 4171–4186
6. Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V (2019) RoBERTa: A Robustly Optimized BERT Pretraining Approach. `1907.11692`
7. Radford A, Narasimhan K, Salimans T, Sutskever I (2018) Improving Language Understanding by Generative Pre-Training. OPENAI blog
8. Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ (2020) Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. Journal of Machine Learning Research 21(140):1–67
9. Qiu X, Sun T, Xu Y, Shao Y, Dai N, Huang X (2020) Pre-trained models for natural language processing: A survey. Science China Technological Sciences 63(10):1872–1897
10. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) CodeBERT: A Pre-Trained Model for Programming and Natural Languages. `2002.08155`
11. Wang Y, Wang W, Joty S, Hoi SCH (2021) CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. `2109.00859`
12. Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng SK, Clement C, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M (2021) GraphCodeBERT: Pre-training Code Representations with Data Flow. `2009.08366`
13. Kwon S, Jang JI, Lee S, Ryu D, Baik J (2023) CodeBERT Based Software Defect Prediction for Edge-Cloud Systems. In: Agapito G, Bernasconi A, Cappiello C, Khattak HA, Ko I, Loseto G, Mrissa M, Nanni L, Pinoli P, Ragone A, Ruta M, Scioscia F, Srivastava A (eds) Current Trends in

Web Engineering, Springer Nature Switzerland, Cham, Communications in Computer and Information Science, pp 11–21
14. Kanade A, Maniatis P, Balakrishnan G, Shi K (2020) Learning and Evaluating Contextual Embedding of Source Code. In: Proceedings of the 37th International Conference on Machine Learning, PMLR, pp 5110–5121
15. Goldberg Y (2019) Assessing BERT's Syntactic Abilities. 1901.05287
16. Jawahar G, Sagot B, Seddah D (2019) What Does BERT Learn about the Structure of Language? In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Florence, Italy, pp 3651–3657
17. Liu K, Yang G, Chen X, Zhou Y (2022) EL-CodeBert: Better Exploiting CodeBert to Support Source Code-Related Classification Tasks. In: Proceedings of the 13th Asia-Pacific Symposium on Internetware, Association for Computing Machinery, New York, NY, USA, Internetware '22, pp 147–155
18. Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S, Herbert-Voss A, Krueger G, Henighan T, Child R, Ramesh A, Ziegler D, Wu J, Winter C, Hesse C, Chen M, Sigler E, Litwin M, Gray S, Chess B, Clark J, Berner C, McCandlish S, Radford A, Sutskever I, Amodei D (2020) Language Models are Few-Shot Learners. In: Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 33, pp 1877–1901
19. Liu P, Yuan W, Fu J, Jiang Z, Hayashi H, Neubig G (2021) Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. 2107.13586
20. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA, NIPS'17, pp 6000–6010
21. Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A Survey of Machine Learning for Big Code and Naturalness. ACM Computing Surveys 51(4):81:1–81:37
22. Nguyen AT, Nguyen TD, Phan HD, Nguyen TN (2018) A deep neural network language model with contexts for source code. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 323–334
23. Harer J, Kim LY, Russell RL, Ozdemir O, Kosta L, Rangamani A, Hamilton LH, Centeno GI, Key JR, Ellingwood PM, McConley M, Opper JM, Chin P, Lazovich T (2018) Automated software vulnerability detection with machine learning. ArXiv
24. Le Q, Mikolov T (2014) Distributed representations of sentences and documents. In: Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32, JMLR.org, Beijing, China, ICML'14, pp II–1188–II–1196
25. DeFreez D, Thakur AV, Rubio-González C (2018) Path-based function embedding and its application to error-handling specification mining. In:

Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2018, pp 423–433
26. Alon U, Zilberstein M, Levy O, Yahav E (2019) Code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages 3(POPL):1–29
27. Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019) A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp 783–794
28. Hu X, Li G, Xia X, Lo D, Lu S, Jin Z (2018) Summarizing source code with transferred API knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, International Joint Conferences on Artificial Intelligence Organization, pp 2269–2275
29. Yang G, Zhou Y, Chen X, Yu C (2021) Fine-grained Pseudo-code Generation Method via Code Feature Extraction and Transformer. In: 2021 28th Asia-Pacific Software Engineering Conference (APSEC), pp 213–222
30. Jain P, Jain A, Zhang T, Abbeel P, Gonzalez J, Stoica I (2021) Contrastive Code Representation Learning. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, pp 5954–5971
31. Yang G (2021) DeepSCC: Source Code Classification Based on Fine-Tuned RoBERTa (S). In: The 33rd International Conference on Software Engineering and Knowledge Engineering, pp 499–502
32. Shinyama Y, Arahori Y, Gondow K (2018) Analyzing Code Comments to Boost Program Comprehension. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC), IEEE, Nara, Japan, pp 325–334
33. Rani P, Panichella S, Leuenberger M, Di Sorbo A, Nierstrasz O (2021) How to identify class comment types? A multi-language approach for class comment classification. Journal of Systems and Software 181:111047
34. Chen Q, Xia X, Hu H, Lo D, Li S (2021) Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction. ACM Transactions on Software Engineering and Methodology 30(2):25:1–25:29
35. Potdar A, Shihab E (2014) An Exploratory Study on Self-Admitted Technical Debt. In: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, Victoria, BC, Canada, pp 91–100
36. Brown N, Cai Y, Guo Y, Kazman R, Kim M, Kruchten P, Lim E, MacCormack A, Nord R, Ozkaya I, Sangwan R, Seaman C, Sullivan K, Zazworka N (2010) Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, Association for Computing Machinery, New York, NY, USA, FoSER '10, pp 47–52

37. Wehaibi S, Shihab E, Guerrouj L (2016) Examining the Impact of Self-Admitted Technical Debt on Software Quality. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol 1, pp 179–188
38. Huang Q, Shihab E, Xia X, Lo D, Li S (2018) Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering 23(1):418–451
39. Ren X, Xing Z, Xia X, Lo D, Wang X, Grundy J (2019) Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability. ACM Transactions on Software Engineering and Methodology 28(3):15:1–15:45
40. Wang X, Liu J, Li L, Chen X, Liu X, Wu H (2021) Detecting and explaining self-admitted technical debts with attention-based neural networks. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, ASE '20, pp 871–882
41. Fowler M (2018) Refactoring. Addison-Wesley Professional
42. Arcelli Fontana F, Zanoni M (2017) Code smell severity classification using machine learning techniques. Knowledge-Based Systems 128(C):43–58
43. Arcelli Fontana F, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering 21(3):1143–1191
44. Das AK, Yadav S, Dhal S (2019) Detecting Code Smells using Deep Learning. In: TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON), pp 2081–2086
45. Liu H, Jin J, Xu Z, Zou Y, Bu Y, Zhang L (2021) Deep Learning Based Code Smell Detection. IEEE Transactions on Software Engineering 47(9):1811–1837
46. Sharma T, Efstathiou V, Louridas P, Spinellis D (2021) Code smell detection by deep direct-learning and transfer-learning. Journal of Systems and Software 176:110936
47. Li Y, Zhang X (2022) Multi-Label Code Smell Detection with Hybrid Model based on Deep Learning. In: The 34th International Conference on Software Engineering and Knowledge Engineering, pp 42–47
48. Sun C, Qiu X, Xu Y, Huang X (2019) How to Fine-Tune BERT for Text Classification? In: Sun M, Huang X, Ji H, Liu Z, Liu Y (eds) Chinese Computational Linguistics, Springer International Publishing, Cham, Lecture Notes in Computer Science, pp 194–206
49. Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I (2019) Language Models are Unsupervised Multitask Learners
50. Lewis M, Liu Y, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L (2020) BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Online, pp 7871–7880

51. Perez E, Kiela D, Cho K (2021) True Few-Shot Learning with Language Models. In: Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 34, pp 11054–11070
52. Schick T, Schütze H (2021) Few-Shot Text Generation with Pattern-Exploiting Training. 2012.11926
53. Jiang Z, Xu FF, Araki J, Neubig G (2020) How Can We Know What Language Models Know? Transactions of the Association for Computational Linguistics 8:423–438
54. Li XL, Liang P (2021) Prefix-Tuning: Optimizing Continuous Prompts for Generation. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), Association for Computational Linguistics, Online, pp 4582–4597
55. Choi H, Kim J, Joe S, Gwon Y (2021) Evaluation of BERT and ALBERT Sentence Embedding Performance on Downstream NLP Tasks. In: 2020 25th International Conference on Pattern Recognition (ICPR), pp 5482–5487
56. Fakhoury S, Arnaoudova V, Noiseux C, Khomh F, Antoniol G (2018) Keep it simple: Is deep learning good for linguistic smell detection? In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 602–611
57. Arnaoudova V, Di Penta M, Antoniol G (2016) Linguistic antipatterns: What they are and how developers perceive them. Empirical Software Engineering 21(1):104–158
58. Pascarella L, Bacchelli A (2017) Classifying code comments in Java open-source software systems. In: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, Buenos Aires, Argentina, MSR '17, pp 227–237
59. Maldonado EdS, Shihab E, Tsantalis N (2017) Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. IEEE Transactions on Software Engineering 43(11):1044–1062
60. Sharma T, Kechagia M, Georgiou S, Tiwari R, Vats I, Moazen H, Sarro F (2022) A Survey on Machine Learning Techniques for Source Code Analysis. 2110.09610
61. Chen T, Guestrin C (2016) XGBoost: A Scalable Tree Boosting System. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Association for Computing Machinery, New York, NY, USA, KDD '16, pp 785–794
62. Kim Y (2014) Convolutional Neural Networks for Sentence Classification. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Doha, Qatar, pp 1746–1751
63. Zhou P, Shi W, Tian J, Qi Z, Li B, Hao H, Xu B (2016) Attention-Based Bidirectional Long Short-Term Memory Networks for Relation Classification. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), Association for

Computational Linguistics, Berlin, Germany, pp 207–212
64. Ding N, Hu S, Zhao W, Chen Y, Liu Z, Zheng HT, Sun M (2021) OpenPrompt: An open-source framework for prompt-learning. arXiv preprint arXiv:211101998 2111.01998
65. Conneau A, Kruszewski G, Lample G, Barrault L, Baroni M (2018) What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Melbourne, Australia, pp 2126–2136