Do Pretrained Language Models Indeed Understand Software Engineering Tasks?

Yao Li, Tao Zhang, *Senior Member, IEEE*, Xiapu Luo, Haipeng Cai, *Senior Member, IEEE*, Sen Fang, and Dawei Yuan

***Abstract*—Artificial intelligence (AI) for software engineering (SE) tasks has recently achieved promising performance. In this article, we investigate to what extent the pre-trained language model truly understands those SE tasks such as code search, code summarization, etc. We conduct a comprehensive empirical study on a board set of AI for SE (AI4SE) tasks by feeding them with variant inputs: 1) with various masking rates and 2) with sufficient input subset method. Then, the trained models are evaluated on different SE tasks, including code search, code summarization, and duplicate bug report detection. Our experimental results show that pre-trained language models are insensitive to the given input, thus they achieve similar performance in these three SE tasks. We refer to this phenomenon as *overinterpretation*, where a model confidently makes a decision without salient features, or where a model finds some irrelevant relationships between the final decision and the dataset. Our study investigates two approaches to mitigate the overinterpretation phenomenon: whole word mask strategy and ensembling. To the best of our knowledge, we are the *first* to reveal this overinterpretation phenomenon to the AI4SE community, which is an important reminder for researchers to design the input for the models and calls for necessary future work in understanding and implementing AI4SE tasks.**

***Index Terms*—Overinterpretation, deep learning, pre-trained language model, software engineering.**

I. Introduction

GIVEN the great potential of artificial intelligence, applying AI for software engineering gains encouraging results in software quality [1], [2], software development [3], [4], and software project management [5], [6]. Despite early successes, AI4SE suffers fundamental explainability problems for its performance [7]. The major reason is that the inside of the neural model is still as mysterious as a black box for researchers. To reveal the nature of AI, recent studies explore controlled empirical studies by specifically targeting on one task. For example, Qu et al. [8] conduct an extensive empirical study to evaluate network embedding algorithms in bug prediction. Different from all previous empirical studies in AI4SE, our empirical study focuses on the impact of input variations on pre-trained language models (PLMs) applied to AI4SE tasks. Specifically, we use a masking strategy or a sufficient subset of inputs (SIS) algorithm to control the inputs of the model for observing the model performance. Therefore, the design of our empirical study is under the hyperthesis that different keywords (unmasked) lead the trained model to achieve different levels of performance. Surprisingly, our experimental results show that by varying masking rate from 15% and 80%, the neural models archive similar results. For example, with 80% masking rate, Bidirectional Encoder Representations from Transformers (BERT) [9] still learns good pre-trained representations and keep more than 90% of the performance on downstream tasks. We call this phenomenon “overinterpretation”.

Overinterpretation is a type of deep learning (DL) model failure, where a model confidently makes a decision without salient features (e.g., keywords), or where a model makes a prediction by utilizing some irrelevant relationships between the final decision and the dataset (i.e., classifying images by background pixels). However, overinterpretation can easily be misleading. Because it looks like the model can even work under bad conditions. For example, the model can classify images in which only 10% of the pixels are retained, and the highly sparse, unmodified subsets of pixels in images suffice for image classifiers to make the same predictions as on the full images [10]. Meanwhile, we have found this phenomenon in SE tasks. Fig. 1 depicts examples of masking 15%, 40%, and 80%, as well their downstream task performance. With 80% masking rate, BERT still learns good representations and keeps more than 90% of the performance on downstream tasks. Even in just 40% masking rate, we can no longer understand the meaning of the sentence, but the model still makes accurate judgments. Fig. 2 presents an example of SE task, i.e., code search. The query displayed in Fig. 2 is partially obscured, with only a few letters visible, namely “L”, “the”, and “ON”. These individual characters do not contain key information that would be readily interpretable by a human, making it challenging to obtain useful information through their combination. However, remarkably,

Manuscript received 8 October 2022; revised 17 July 2023; accepted 23 August 2023. Date of publication 28 August 2023; date of current version 17 October 2023. This work was supported in part by Macao Science and Technology Development Fund (FDCT), Macau SAR under Grands 0047/2020/A1 and 0014/2022/A, and in part by Hong Kong RGC Project (No. PolyU15224121) and HKPolyU Grant (ZVG0). Recommended for acceptance by M. Nagappan. *(Corresponding author: Tao Zhang.)*

Yao Li, Tao Zhang, Sen Fang, and Dawei Yuan are with the School of Computer Science and Engineering, Macau University of Science and Technology, Macao 999078, China (e-mail: 2109853gia30001@student.must.edu.mo; tazhang@must.edu.mo; fangsen1996@gmail.com; wu.xiguanghua2014@gmail.com).

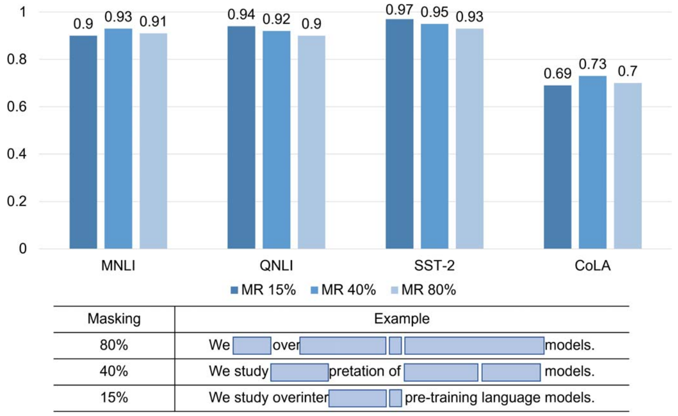
Xiapu Luo is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong 999077, China (e-mail: csxluo@comp.polyu.edu.hk).

Haipeng Cai is with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99163, USA (e-mail: haipeng.cai@wsu.edu).

Digital Object Identifier 10.1109/TSE.2023.3308952

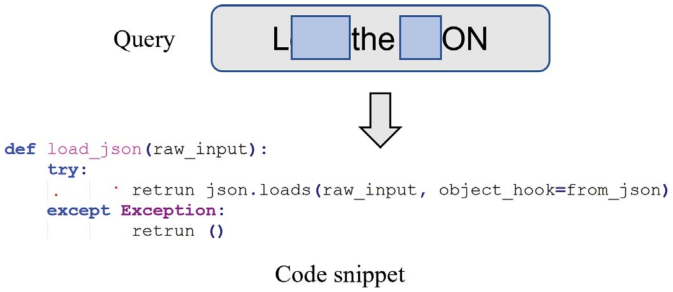
TABLE I
The Overview of RQs and Findings

RQ	Task & Methodology	Finding
Do software engineering tasks (code search, code summarization, and duplicate bug report detection) suffer from overinterpretation?	Code search, code summarization, duplicate bug report detection: Masking & SIS	Software engineering tasks suffer from overinterpretation.
Does overinterpretation depend on software engineering tasks and how prevalent is overinterpretation in PLMs?	GPT, BERT, XLNet: Masking & SIS	Overinterpretation is not dependent on software engineering tasks and is also prevalent in pre-trained language models.
What is the impact of overinterpretation? What are the challenges in mitigating overinterpretation in general and how to mitigate overinterpretation?	Whole word masking & Ensembling	The main challenge is that overinterpretation is more difficult to detect. Whole word masking and ensembling can mitigate overinterpretation.



Masking	Example
80%	We ▭ over▭ ▭ ▭ models.
40%	We study ▭ pretation of ▭ ▭ models.
15%	We study overinter▭ ▭ pre-training language models.

Fig. 1. Performance of PLMs under different masking rates. “MR” means masking rate. Different mask rate leads to similar performance in the four considered metrics.



```
def load_json(raw_input):
    try:
            retrun json.loads(raw_input, object_hook=from_json)
    except Exception:
            retrun ()
```

Fig. 2. A code search example after the query is masked. Blue block indicates masking.

the model is still able to find the corresponding code snippet successfully. This example vividly demonstrates the model’s tendency to overinterpret and make accurate predictions even when presented with limited or seemingly meaningless input. Despite the masked characters lacking any meaningful interpretation from a human standpoint, the model leverages its underlying statistical learning capabilities to derive meaningful patterns and associations. As a result, the model is capable of producing comparable results to those obtained with the complete query. This highlights the model’s ability to go beyond human understanding and uncover latent patterns that might not be apparent to us. However, it also raises concerns about the potential for overreliance on statistical associations and the possibility of making interpretations that may not align with human reasoning. Given that the DL models in the above examples which have such remarkable success in SE tasks, it is natural to ask why they perform so well, after the inputs have been masked, what kinds of features these models are learning, and whether they can understand features, i.e., *can they indeed understand SE tasks?*

To answer these questions, the key point is the input features. The features used by the model are derived from the dataset. However, dataset has implicit biases and unique statistical signals that are often introduced during the dataset generation/solidification/labeling process [11]. These biases and statistical signals often allow DL models to achieve high accuracy in test data by learning highly specific features unique to that dataset rather than generalizable features or key features under human understanding. For example, Ribeiro et al. [12] describe an example of a classifier that classifies wolves and huskies. They find that the classier predicts “Wolf” if there is snow (or light background at the bottom), and “Husky” otherwise, regardless of animal color, position, pose, etc. Biases in the dataset and unique statistical signals are learned by the model and cause overinterpretation [11].

In this article, we conduct the first comprehensive empirical study on the overinterpretation of PLMs applied in the SE tasks. Our study contains two parts, task-oriented and model-oriented. In the task-oriented part, we study three SE tasks, code search [13], [14], code summarization [15], [16], and duplicate bug report detection [17], [18]. These tasks are not only widely used in software development and maintenance, but also encompass language processing techniques such as natural language (NL) to programming language (PL) translation, PL to NL translation, and NL classification. In the model-oriented part, we study three famous PLMs, Generative Pre-Training (GPT) [19], BERT [9], and XLNet [20]. These PLMs are widely used in various tasks. The purpose of this study is to provide a systemic and generalized understanding of the overinterpretation of PLMs, which could improve model architecture and solve potential issues such as misclassification, low generalization, etc. To experiment with our method, we adopt the method of Wettig et al. [21] to re-pre-train the model. All models are trained from scratch. We set three research questions (RQ) to verify the overinterpretation. The RQs and findings in the article are shown in Table I. We describe them in details as following subsections.

A. SE Tasks Analysis

To better investigate the models in the SE tasks, we analyze three SE tasks, code search, code summarization, and duplicate bug report detection. With the support of representative studies, our analysis is driven by the following research question:

RQ 1:

Do software engineering tasks (code search, code summarization, and duplicate bug report detection) suffer from overinterpretation?

1) Code Search: The basic principle of code search is to accept full queries and find the correct code snippets. However, our analysis results show that PLMs [13], [14] can still find the corresponding code snippets after entering a masked query (even masking 80% of the query) or SIS (a sparse set of unrelated characters). The masked query and SIS retain only meaningless letters or non-essential words. However, models make accurate decisions based on these confusing letters. For details, please refer to Section V-A.

2) Code Summarization: Despite multiple publications proposing new code summarization methods [22], [23], [24], we do not find an analysis of overinterpretation. Therefore, we analyze several approaches [15], [16] to investigate whether they overinterpret the dataset.

The investigation results show that, despite the lack of input code (masking strategy and SIS), these studies [15], [16] are able to accurately generate the corresponding summaries. However, the input that remains is unrelated and confusing, and cannot be understood from a human perspective. For more details, please refer to Section V-B.

3) Duplicate Bug Report Detection: Capturing and tagging duplicate bug reports is crucial to avoid the assignment of the same bug to different developers. We design a variety of experiments to study and analyze some studies [17], [18].

We use three different masking strategies (15% masking rate, 40% masking rate, 80% masking rate) and SIS to train these two models [17], [18] separately. When lacks most of the content of the description tag, the model can still detect the duplicate report. The retained descriptions are unreadable and meaningless, much less containing salient features. For the details, please refer to Section V-C.

B. PLMs Analysis

To demonstrate that overinterpretation is not task-dependent and is prevalent in pre-trained language models, we choose three representative PLMs (GPT, BERT, and XLNet) for evaluation.

RQ 2:

Does overinterpretation depend on software engineering tasks and how prevalent is overinterpretation in PLMs?

We find that PLMs [9], [19], [20] can still make accurate decisions under conditions where most of the data and salient features are missing (by using masking strategies and SIS). Overinterpretation not only depends on SE-related tasks but is also prevalent in PLMs. For the details, please refer to Section VI.

C. Impact Analysis and Mitigation

Overinterpretation is a potential pitfall. It suggests that the pre-trained language model learns not the salient features in the dataset, such as some keywords in the text. Instead, it learns statistical signals that are unique to the data. Thus, in this article, we are interested in exploring what the hindrances to alleviate this flaw are and how to mitigate overinterpretation.

RQ 3:

What is the impact of overinterpretation? What are the challenges in mitigating overinterpretation in general and how to mitigate overinterpretation?

There are three main challenges. First, overinterpretation is not well understood and studied at present. Meanwhile, overinterpretation can be misleading. Second, overinterpretation is not easily detected. Overinterpretation may come from real statistical signals in the distribution of the underlying dataset. Finally, the pre-trained language model is a black-box model. Moreover, we find two ways to mitigate overinterpretation through experiments, whole word mask and ensembling. These two methods can enrich the dataset used by the model. For the details, please refer to Section VII.

D. Contributions

In summary, this article makes the following contributions:

- We perform the *first* comprehensive study on the overinterpretation of pre-trained language models in SE. We demonstrate that PLMs in SE suffers from overinterpretation.
- We design two schemes to reveal overinterpretation. One is a different masking rate scheme and the other is a sufficient input subset scheme.
- We find two ways to mitigate overinterpretation, whole word mask strategy and ensembling. These two methods can enrich model learning to mitigate overinterpretation.

The rest of this article is organized as follows. In Section II, we give an overview of PLMs, AI4SE tasks, and overinterpretation. Section III describes the study methodology and Section IV describes datasets and experimental setup. Sections V and VI present the analysis of the overinterpretation in SE tasks and PLMs. Section VII introduces the impact of overinterpretation and mitigation measures. Section VIII describes the threats to validity. We survey related work in Section IX and conclude this article in Section X.

II. Background

A. PLMs

Pre-training has always been an effective strategy to learn the parameters of deep neural networks, which are then fine-tuning on downstream tasks. As early as 2006, the breakthrough of deep learning came with greedy layer-wise unsupervised pre-training followed by supervised fine-tuning [25]. In natural language processing (NLP), PLMs on large corpus have been proven to be beneficial for the downstream NLP tasks. With the development of computers, PLMs change a lot, from shallow word embedding to deep neural networks.

Language modeling (LM) objectives for pre-training mainly fall into two categories: (1) autoregressive language modeling, where the model is trained to predict the next token based on the previous context:

$$L(C) = \mathbb{E}_{x\in C}\left[\sum_{x_i\in x} \log p\left(x_i | x_1, x_2, \ldots, x_{i-1}\right)\right] \tag{1}$$

where C is a pre-training corpus and x is a sampled sequence from C. (2) De-noising auto-encoding, where the model is trained to restore a corrupted input sequence. In particular, masked language models (MLMs) [26], [27] mask a subset of input tokens and predict them based on the remaining context:

$$L(C) = \mathbb{E}_{x\in C}\mathbb{E}_{\mathcal{M}\subset x,\ |\mathcal{M}|=m|x|}\left[\sum_{x_i\in\mathcal{M}} \log p\left(x_i \middle| \tilde{x}\right)\right] \tag{2}$$

where a mask m (masking rate, typically 15%) percentage of tokens from the original sentence x and predicts the masked tokens M given the corrupted context x (the masked version of x).

Different masking strategies have been proposed to sample M: Devlin et al. [9] randomly choose from the input tokens with a uniform distribution; Joshi et al. [28] sample contiguous spans of text; Levine et al. [29] sample words and spans with high Pointwise Mutual Information (PMI). These advanced sampling strategies prevent models from exploiting shallow local cues from uniform masking and lead to efficient pre-training. MLMs can encode bidirectional context while autoregressive language models can only "look at the past", and thus MLMs are shown to be more effective at learning contextualized representations for downstream tasks [9]. On the other hand, MLMs suffer a significant computational cost because it only learns from 15% of the tokens per sequence, whereas autoregressive LMs predict every token in a sequence.

B. AI4SE Tasks

Human life depends on reliable software; therefore, the software production process (i.e., software design [30], development [4], and maintenance [31]) becomes one of the most important factors to ensure the quality of software. With the increase in the complexity of software, how to improve the performance and efficiency of software production has become a challenge for software developers and researchers. To address this challenge, researchers have used information retrieval and DL technologies to implement a series of automated tools. These tools can solve SE tasks, such as code search, code summarization, and duplicate bug report detection.

Code search [13], [32], [33], [34], [35] is frequently used by developers to conveniently find relevant code snippets. McMillan et al. [36] propose a code search engine that combines keyword matching with PageRank to return a chain of functions. Lv et al. [35] propose CodeHow, a code search tool that incorporates an extended Boolean model and API matching. Ponzanelli et al. [37] propose an approach that automatically retrieves pertinent discussions from Stack Overflow given a context in the integrated development environment (IDE). Code summarization automatically generates high-quality text to help developers understand the program. Sridhara et al. [22] generate a code summary for the Java method from its method call and signature using the NLP techniques. Software Word Usage Model has been built for software analysis using NLP by Pollock et al. [23]. Fowkes et al. [24] design an unsupervised, extractive source code summarization system using an autofolding method. Programmers fix bugs based on bug reports. BM25F [38] calculates the similarity between two bug reports based on common words shared between the bug reports. REP [39] extends BM25F by also considering bug report attribute information (e.g., product, priority). Deshmukh et al. [40] propose a deep learning-based approach (i.e., DLDBR), which mainly relies on the textual feature to detect duplicate bug reports. Our study is different from the above work. We study some tasks in program comprehension to demonstrate the overinterpretation in them.

More and more AI-based schemes have been proposed to solve traditional SE tasks [4], [30], [41]. However, previous studies analyze the impact of different programming languages and antipatterns on program understanding [41], [42], [43], as well as investigated the positive impact of dynamic analysis on program comprehension [44], [45]. Moreover, researchers have studied sub-tasks of program understanding to help improve efficiency, such as code search [46], code summarization [47], and duplicate bug report detection [48], etc.

C. Overinterpretation

Overinterpretation is a serious issue in black-box models. We define model overinterpretation to occur when a model finds strong class evidence in input that contains no semantically salient features. Overinterpretation is related to overfitting, but overfitting can be diagnosed via reduced test accuracy [10]. Overinterpretation may stem from real statistical signals in the distribution of the underlying data set that happen to be generated by specific properties of the data source. Thus, overinterpretation is harder to be diagnosed as it admits decisions that are made by statistically valid criteria, and models that use such criteria can excel at benchmarks. Meanwhile, PLMs always require a large dataset to train and fine-tune. Datasets always contain implicit biases and unique statistical signals. These biases and statistical signals often allow DL models to achieve high accuracy in test data by learning highly specific features unique to that dataset rather than generalizable features

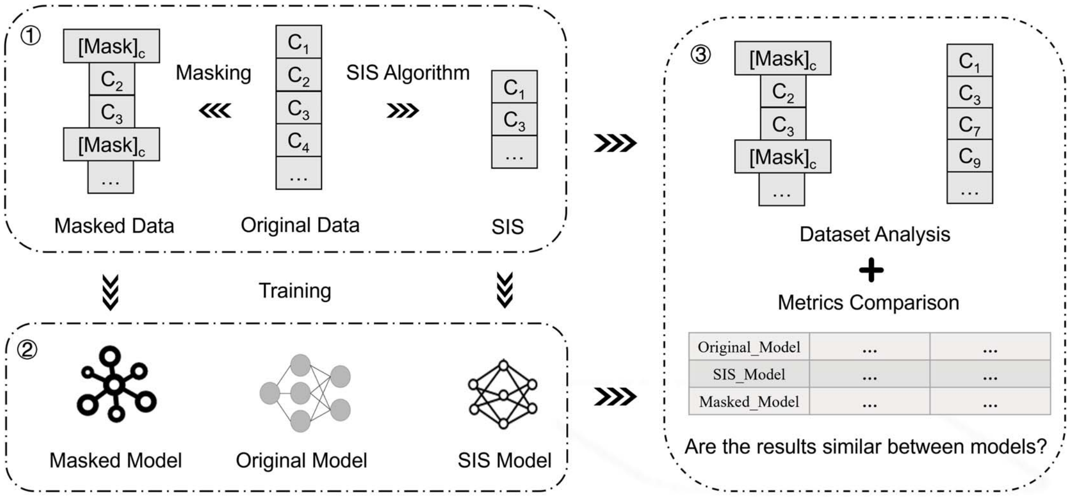


Fig. 3. Flow of analysis in this study. Step ①, we process the raw data using the masking strategy and the SIS algorithm. Step ②, the models are trained using the processed data and the original data, respectively. Step ③, the three types of models and the features they use are compared and analyzed.

or key features under human understanding [11]. However, these non-distinctive features are beyond human comprehension. They may be a series of unrelated characters or sparse pixels, and are not the features that we consider to be capable of making critical decisions. Although the outward appearance of this phenomenon can be surprising, as the model still works under bad conditions. But the model learns not problem-based features but dataset-based features. This can make the model less generalizable. Understanding overinterpretation is significant for improving model quality. Moreover, it can also provide guidance for designing the architecture of models.

III. Methodology

A. Overview

Revealing overinterpretation requires a systematic way to identify which features are used by a model to reach its decision. In this study, to comprehensively examine whether pre-trained language models are overinterpreted, we propose two evaluation methods. One is to use multiple different masking rates to train the model. The other is to train the model using SIS. As shown in Fig. 3, our study consists of three main steps: 1) *we use two methods to process the dataset. One is the masking rate strategy which masks the dataset with different degrees, and the other is to extract sufficient input subsets using the SIS algorithm;* 2) *we train the PLM using the masked dataset and SIS separately;* 3) *we use multiple evaluation metrics to comprehensively evaluate and compare the models, and analyze the corresponding datasets.* To experiment with our method, all models are trained from scratch. we re-pre-train models using SIS and MR methods and fine-tune them on the downstream tasks. We adopt the method used by Wettig et al. [21]. Downstream task development performance of large models trained with the efficient pre-training recipe, under different masking rates. To ensure the integrity of our model and minimize any potential confounding factors, we employed the exact same parameters as the original model. In addition, we extensively refer to relevant articles and open-source codes of the models used in this article to ensure a robust and reliable training process. It is important to note that all the models we use in our research are open-sourced with their respective source codes and articles. The data underlying this article are available at our repository.

B. Different Masking Rate Strategies

The emergence of pre-trained models has made the training of large models easy. However, a masking strategy is introduced to perform sufficient representation learning during pre-training. It increases the learning difficulty of the model to some extent even though the model is more generalizable.

In this study, we set three masking rates, 15% masking rate, 40% masking rate, and 80% masking rate. In addition to using the 15% masking rate to train the model, we also use the 40% and 80% masking rates to train the model. A series of experiments are conducted to compare the differences between the classifiers trained with each masking rate. As the masking rate increases, the more corpus is masked, the more difficult it becomes for humans to understand its meaning. The performance of the PLM does not change much when the masking rate is increased from 15% to 80% (the evaluation metric stays within 10% for both improvement and decrease). This fact indicates that the large reduction in input data has little impact on the effectiveness of the PLM. Meanwhile, as the masking rate increases, the utterance contains less information and becomes more difficult to understand. When the masking rate reaches 40%, humans can no longer understand the meaning of the original sentence correctly. Thus, PLMs can learn information from unrelated or even meaningless words or letters to make the final decision. This fact proves that pre-trained language models overinterpret the data.

C. Sufficient Input Subset

The idea of SIS has been proposed to help humans interpret the decisions of black-box models [49]. An SIS subset is a minimal subset of features that suffices to yield a class probability above a certain threshold with all other features masked. One simple explanation about why a particular black-box decision is reached may be obtained via a sparse subset of the input features whose values form the basis for the model's decision — a rationale.

The SIS rationalizes why reaching a particular black-box decision only applies to input instances x that satisfy the decision criterion $f(x) > m$. For such an input x, we aim to identify an SIS-collection of disjoint feature subsets that satisfy the following criteria:

- $f(X_{S_n}) \geq \tau \ for \ each \ n = 1, ..., N$
- There exists no feature subset $S^{'} \subset S_n$ for some $n = 1, ..., N$ such that $f(X_{S'}) \geq \tau$
- $f(X_R) < \tau \ for \ R = [p] \setminus \bigcup_{n=1}^{N} S_N$ (the remaining features outside of the SIS-collection)

Criterion (1) ensures that for any SIS (S_n), in the absence of any other features, the features in that subset alone are sufficient to justify the decision. Criterion (2) ensures that each SIS that reaches a decision contains a minimum number of features. Criterion (3) no longer reaches the same decision on the input after the entire SIS set is masked.

Thus, we perform the SIS algorithm to extract the subset of corpus and then train the corresponding model. By comparing the selected evaluation metrics, we consider the extracted SIS to be valid if the model trained using SIS is similar to the model trained using the full dataset. Then, we analyze the SIS to determine whether humans can understand the meaning of the SIS. If the SIS is not meaningful, it indicates that the model suffers from overinterpretation. For the SIS algorithm used in this article, we refer to the method of [49] by Carter et al. Their source code and parameters can be found on GitHub.

IV. Experimental Setup

To ensure the authenticity of the experiments, all models use the default settings and the datasets are the same as those used in the original work. All the work investigated in the article has public datasets and source code. The source code and dataset of code search tasks are provided by the previous studies [13], [14]. We can find code summarization's source code and dataset in the literatures [15], [16]. Duplicate bug report detection task's code and dataset are in the following studies [17], [18]. The code snippets of PLMs can be found in the following studies [9], [19], [20]. The evaluation indicators in this article are the same as in the original work. All experiments are done on the same machine. We run our experiments on single NVIDIA Geforce 3090Ti GPU, Intel (R) Xeon(R) 2.60GHz 16 CPU. To comprehensively demonstrate the overinterpretation of the PLMs, we design two types of experiments. For the details, please refer to Sections V and VI.

V. Task-Oriented Overinterpretation Analysis

We study some AI4SE tasks and reveal the overinterpretation in these tasks. We choose three tasks, code search [13], [14], code summarization [15], [16], and duplicate bug report detection [17], [18]. These tasks are not only widely used in software development and maintenance, but also encompass language processing techniques such as NL to PL translation, PL to NL translation, and NL classification. For each task, two representative studies are selected and their models are evaluated to detect the presence of overinterpretation. The next sections describe these experiments in detail.

TABLE II
Overall Accuracy of DeepCS Under Different Masking Rates. "MR" Means Masking Rate, and "0%" Indicates the Absence of Any Masking Strategy

Pre-training	Metrics						
MR	R@1	R@5	R@10	P@1	P@5	P@10	MRR
0%	0.51	0.82	0.89	0.49	0.51	0.48	0.61
15%	0.48	0.76	0.85	0.48	0.47	0.46	0.60
40%	0.47	0.75	0.82	0.46	0.44	0.45	0.59
80%	0.42	0.69	0.77	0.41	0.42	0.43	0.53

RQ 1:

Do software engineering tasks (code search, code summarization, and duplicate bug report detection) suffer from overinterpretation?

A. Code Search

Code search is a frequent activity in software development. To implement a program functionality, developers can reuse previously written code snippets by searching through a large-scale codebase. Over the years, many code search tools have been proposed to help developers such as DeepCS [13] and CodeBERT [14], both of which use deep learning models to conduct code search.

DeepCS is a novel code search tool using deep embedding neural networks. Instead of matching textual similarities, DeepCS co-embeds code snippets and natural language descriptions into a high-dimensional vector space, and then performs searches based on the vectors. They empirically evaluate DeepCS on a large-scale codebase collected from GitHub. The experimental results show that the method can effectively retrieve relevant code snippets and outperforms previous techniques. CodeBERT is a bimodal pre-trained model for a programming language and natural language. Authors evaluate CodeBERT on two NL-PL applications by fine-tuning model parameters. Results show that CodeBERT achieves state-of-the-art performance on both natural language code search and code documentation generation.

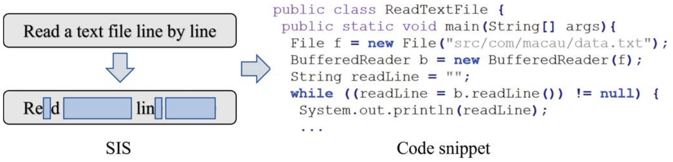
First, we train models using different masking rates, 15% masking rate, 40% masking rate, and 80% masking rate. Four common metrics are used to measure the effectiveness of code search, namely, FRank [33], SuccessRate@k [34], Precision@k [35], and Mean Reciprocal Rank (MRR) [50]. They are widely used metrics in information retrieval and code search literature. Table II shows the performance of DeepCS under different masking rates. DeepCS outperforms other models when training models using the full dataset. But after using the masking rate strategy to train the model, the performance of DeepCS decreases. Moreover, as

TABLE III
OVERALL ACCURACY OF CODEBERT UNDER DIFFERENT MASKING RATES. "MR" MEANS MASKING RATE, AND "0%" INDICATES THE ABSENCE OF ANY MASKING STRATEGY

MR	RUBY	JAVASCRIPT	GO	PYTHON	JAVA	PHP
0%	0.69	0.71	0.84	0.86	0.74	0.70
15%	0.71	0.72	0.83	0.87	0.74	0.72
40%	0.74	0.70	0.85	0.88	0.75	0.71
80%	0.67	0.68	0.80	0.82	0.71	0.69

the masking rate increases, the performance of DeepCS gradually decreases. When the model is trained with 80% masking rate, DeepCS has the lowest performance, but the difference is within 10% compared to the model trained with the complete dataset. This fact demonstrates that DeepCS can learn enough "knowledge" to make accurate judgments with most of the input missing (at least 20% of the data is retained). "Knowledge" refers to the real statistical signals in the dataset. Meanwhile, the high masking rate means that it is difficult for humans to understand masked sentences. We conduct the same experiment with CodeBERT. Table III shows the performance of CodeBERT. Unlike DeepCS, the performance of CodeBERT does not decrease as the masking rate increases, but increases. This result indicates that the performance of CodeBERT improves as the input dataset is reduced. Moreover, CodeBERT can still make accurate judgments when the masking rate is 80%, i.e., only 20% of the input data is retained. This suggests that the above models do not learn the features in the dataset but the statistical signals in the dataset.

Then, we use the SIS algorithm to extract a subset to train DeepCS, and CodeBERT. Results are shown in Fig. 5(a) and 5(b). SIS is a subset of the complete dataset, which is sparse but allows the model to make accurate decisions. To demonstrate SIS in more detail, we show an example of code search, as shown in Fig. 4. The original query is "Read a text file line by line." After applying the SIS algorithm, the extracted SIS is shown, which includes the terms "red" and "lin". It is important to note that the SIS is significantly different from the original query in terms of its composition. For humans, the salient features of the query are typically "read" and "line," and based on these words, humans can make certain inferences and understand the context to search for a relevant code snippet. However, when we consider the SIS, it becomes difficult for humans to comprehend what the SIS represents, and it becomes even more challenging to find the corresponding code snippet based solely on the SIS. In contrast, a pre-trained language model can still perform code search based on the SIS. Despite the lack of human interpretability, the model can leverage the subset of relevant terms in the SIS to accurately retrieve the corresponding code fragment. This demonstrates the model's ability to utilize the subset of salient features, even when they may not be easily understandable or interpretable from a human perspective. This suggests that pre-trained language models do not make decisions based on salient features, but learn statistical signals unique to the dataset to make judgments. In the absence of salient features, DeepCS and CodeBERT can still achieve good performance. Therefore, the tasks related to code search overinterpret the dataset.



```
public class ReadTextFile {
 public static void main(String[] args){
  File f = new File("src/com/macau/data.txt");
  BufferedReader b = new BufferedReader(f);
  String readLine = "";
  while ((readLine = b.readLine()) != null) {
   System.out.println(readLine);
   ...
```

Fig. 4. Example of SIS in the code search. The blue blocks mean the data filtered out by the SIS algorithm.

Finding 1.1: Code search suffers from overinterpretation. PLMs trained in different ways have similar performance, and these models are trained without salient features.

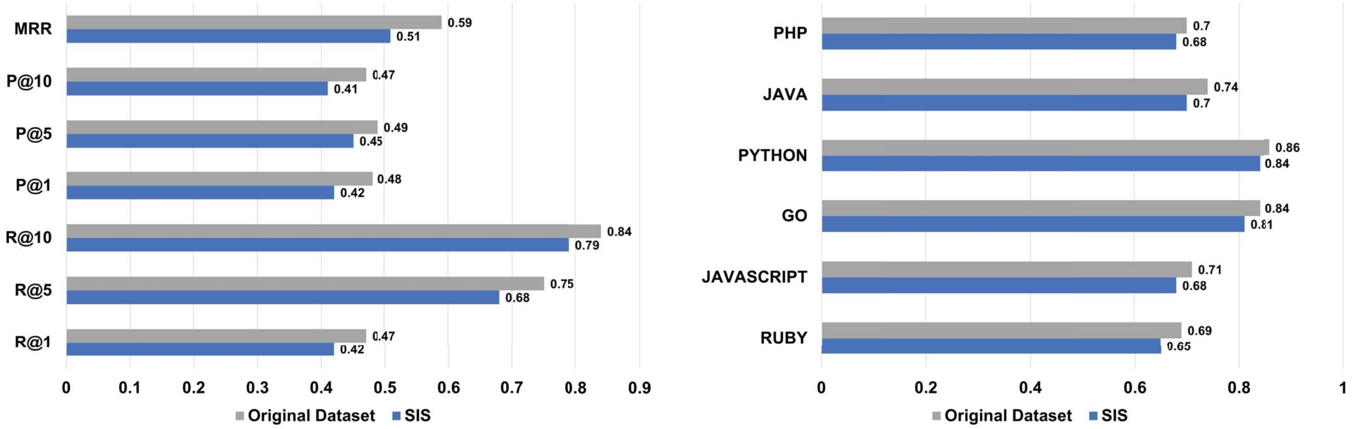
B. Code Summarization

Generating a readable summary that describes the functionality of a program is known as source code summarization which can help developers understand and maintain software. In this task, learning code representation by modeling the pairwise relationship between code tokens to capture their long-range dependencies is crucial. To learn code representation for summarization, researchers have proposed many efficient methods such as [15] and [16].

Ahmad et al. [15] explore the Transformer model that uses a self-attention mechanism and has shown to be effective in capturing long-range dependencies. The authors perform experiments on two well-studied datasets, and the results endorse the effectiveness.

Iyer et al. [16] present, CODE-NN, the first completely data-driven approach for generating high-level summaries of source code. Experiments outperform strong baselines.

We choose the above research studies to investigate whether there is overinterpretation. The experimental results are shown in Table IV. We evaluate the source code summarization performance using three metrics, BLEU [51], METEOR [52], and ROUGE-L [53]. The performance of Transformer-based shows a pattern of increasing and then decreasing as the masking rate increases. In addition, the 80% masking rate outperforms the 15% masking rate. This result shows that pre-trained language models can make accurate decisions based on either the complete dataset or only 20% of the dataset retained. It also proves that not all data are useful for Transformer-based. CODE-NN's performance decreases as the masking rate increases. Although each metric decreases, the difference in metrics is small (within 10%) compared to the model trained on the full dataset. CODE-NN learns approximately for the dataset, regardless of the amount of data contained in the dataset. With the loss of a large amount of data, the CODE-NN can still make accurate judgments, and the only remaining datasets are completely incomprehensible to humans. Fig. 6 shows an example of training code summarization using the SIS. The left part shows the original code snippet and the right part shows the subset extracted by the SIS algorithm. In the context of code summarization, a code fragment consists of several salient features, including class names, function names, and more. These features play a crucial role in indicating the purpose and functionality of the code.



(a) Performance of DeepCS under SIS

(b) Performance of CodeBERT under SIS

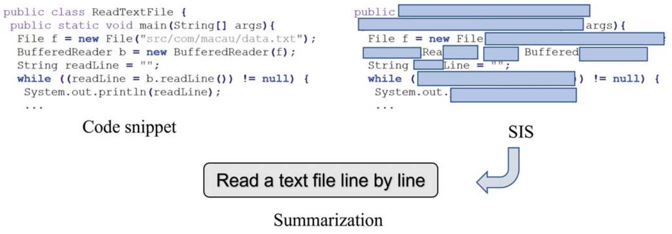
Fig. 5. The performance of SIS in the code search.

TABLE IV
Overall Performance of Code Summarization Tasks Under Different Masking Rates. "MR" Means Masking Rate, and "0%" Indicates the Absence of Any Masking Strategy

Works	Pre-T	Metrics		
	MR	BLEU	METEOR	ROUGE-L
Transformer-based	0%	0.44	0.26	0.54
	15%	0.45	0.25	0.55
	40%	0.47	0.32	0.61
	80%	0.46	0.28	0.57
CODE-NN	0%	0.25	0.17	0.56
	15%	0.22	0.15	0.53
	40%	0.21	0.13	0.51
	80%	0.18	0.10	0.48

However, in the masked code snippet, the function names and class names that are typically indicative of the code's purpose are masked, making it challenging for humans to comprehend the function of the code snippet by solely reading the masked code. Despite the lack of explicit salient features, PLMs are still able to generate accurate code descriptions based on masked code. This suggests that PLMs have learned to capture the underlying patterns and relationships within the code, enabling them to generate meaningful and accurate summaries even when specific salient features are masked or absent. This finding highlights the ability of PLMs to generalize and understand the semantics of the code beyond relying solely on explicit salient features. Humans cannot understand the function of this code snippet by reading the masked code. But PLMs still generate accurate code descriptions. Fig. 7(a) and 7(b) describes the performance of the model after training with SIS. Although the performance of both tasks decreased after training with SIS. But the difference with the model trained using the full dataset is small. The code summarization model can still achieve good results in the absence of salient features. Therefore, there is an overinterpretation of the code summarization task.

Finding 1.2: The model is able to generate accurate summaries despite the absence of salient features. The experimental results show that the model overinterpret the dataset.



```
public class ReadTextFile {
 public static void main(String[] args){
  File f = new File("src/com/macau/data.txt");
  BufferedReader b = new BufferedReader(f);
  String readLine = "";
  while ((readLine = b.readLine()) != null) {
   System.out.println(readLine);
   ...
```

```
public
                          args){
  File f = new File
       Rea            Buffered
  String    Line = "";
  while (                     ) != null) {
   System.out.
   ...
```

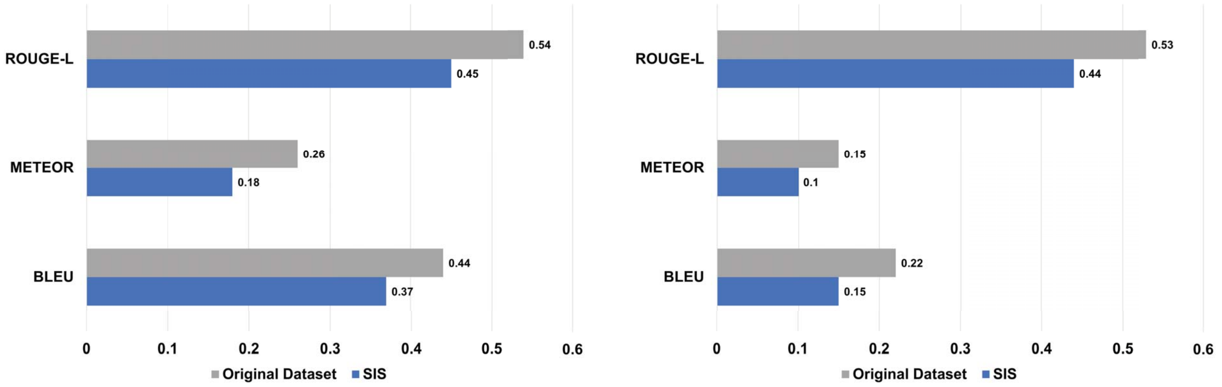
Fig. 6. Example of SIS in the code summarization. The blue blocks mean the data filtered out by the SIS algorithm.

C. Duplicate Bug Report Detection

Bug report filing is a major part of software maintenance. Developers rely on bug reports to fix bugs. Due to different expression habits, different reporters may use different expressions to describe the same bug in the bug tracking system. As a result, the bug tracking system usually contains many duplicate bug reports. Automated duplicate detection can reduce developers' workload on fixing duplicate bugs. In other words, capturing and tagging duplicate bug reports is crucial to avoid the assignment of the same bug to different developers. Efforts have been made in the past to detect duplicate bug reports by using deep learning methods [17], [18].

Xiao et al. [17] present HINDBR, a novel deep neural network (DNN) that accurately detects semantically similar duplicate bug reports using a heterogeneous information network (HIN). Results show that HINDBR is effective. Budhiraja et al. [18] propose Deep Word Embedding Network (DWEN) that uses a deep word embedding network for duplicate bug report detection. DWEN computes the similarity between two bug reports for duplicate bug report detection. Results show that the proposed approach is able to perform better than baselines.

Table V shows the metrics of both works under different masking rates, 15% masking rate, 40% masking rate, and 80% masking rate. To evaluate models, we use the following four metrics, Accuracy, Precision, Recall, and F1-Score. When the model is trained using the masking rate strategy, the performance of the model starts to decrease. As the masking rate



(a) The performance of Transformer-based using SIS.

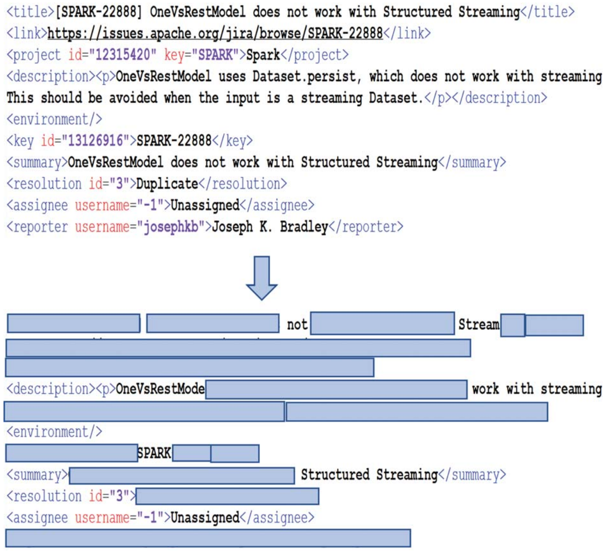
(b) The performance of CODE-NN under SIS.

Fig. 7. The performance of code summarization under SIS.

TABLE V
OVERALL PERFORMANCE OF DUPLICATE BUG REPORT DETECTION TASKS UNDER DIFFERENT MASKING RATES. "MR" MEANS MASKING RATE, AND "0%" INDICATES THE ABSENCE OF ANY MASKING STRATEGY

Works	Pre-T	Metrics			
	MR	Accuracy	Precision	Recall	F1 Score
HINDBR	0%	0.96	0.91	0.88	0.87
	15%	0.94	0.89	0.83	0.86
	40%	0.92	0.86	0.80	0.84
	80%	0.89	0.81	0.79	0.78
DWEN	0%	0.82	0.73	0.79	0.78
	15%	0.80	0.70	0.76	0.77
	40%	0.75	0.67	0.74	0.73
	80%	0.74	0.65	0.70	0.68

increases, all the metrics of the model decrease. The effect of the model is minimized when the masking rate reaches 80%. However, the difference between all metrics and the original model is maintained within 10%. This fact demonstrates that HINDBR and DWEN cannot learn key representations from the rich dataset. Fig. 8 displays an example of using SIS to detect duplicate bug reports. The top part shows the complete bug report, and the bottom part shows the bug report after masking. Detecting duplicate bug reports relies on learning the distinctive characteristics of a report based on its description. However, in the masked bug report, a significant portion of the content in the description tag is obscured or masked. This makes it challenging for humans to understand the remaining content in the description tag, as it appears to be meaningless or lacks the necessary context. The observation highlights the difficulty faced by humans in comprehending the masked bug report and drawing connections or identifying similarities between different bug reports based on the available information. However, it is worth noting that pre-trained language models, which have been trained on large-scale data and have learned to capture intricate patterns and relationships, can still utilize masked information to accurately detect duplicate bug reports. This example underscores the potential overinterpretation issue in the task of duplicate bug report detection. While humans may struggle to extract meaningful insights from the masked bug report, pre-trained language models can leverage the available information, even if it appears nonsensical to humans, to successfully identify duplicate reports based on learned patterns and correlations. Moreover, the results of the two schemes using SIS detection are depicted in Fig. 9. Although all four metrics decline for the model trained with SIS, the difference with the original model (the model trained with the full dataset) stays within 10%. Thus, there is an overinterpretation phenomenon in the studies related to duplicate bug report detection.



```
<title>[SPARK-22888] OneVsRestModel does not work with Structured Streaming</title>
<link>https://issues.apache.org/jira/browse/SPARK-22888</link>
<project id="12315420" key="SPARK">Spark</project>
<description><p>OneVsRestModel uses Dataset.persist, which does not work with streaming
This should be avoided when the input is a streaming Dataset.</p></description>
<environment/>
<key id="13126916">SPARK-22888</key>
<summary>OneVsRestModel does not work with Structured Streaming</summary>
<resolution id="3">Duplicate</resolution>
<assignee username="-1">Unassigned</assignee>
<reporter username="josephkb">Joseph K. Bradley</reporter>
```

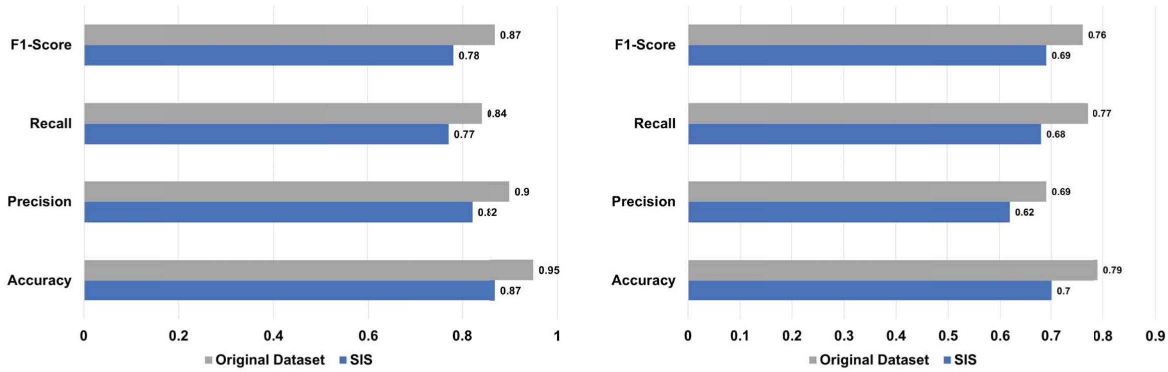
```
                                  not                      Stream


<description><p>OneVsRestMode                                     work with streaming

<environment/>
                SPARK
<summary>                            Structured Streaming</summary>
<resolution id="3">
<assignee username="-1">Unassigned</assignee>
```

Fig. 8. Example of SIS in the duplicate bug report detection tasks. The blue blocks mean the data filtered out by the SIS algorithm.

> **Finding 1.3:** Lacking a salient part of the description, the model can still find duplicate reports. The PLMs in the duplicate bug report detection task suffers from overinterpretation.

D. Analysis of SE Tasks

To understand whether the phenomenon of overinterpretation is prevalent in AI4SE tasks, we choose three types of tasks for our experiments, code search [13], [14], code summarization [15], [16], and duplicate bug report detection [17], [18]. Tables II–V present the results under different masking rates,



(a) Performance of HINDBR under SIS.

(b) Performance of DWEN under SIS.

Fig. 9. The performance of duplicate bug report detection tasks under SIS.

15% masking rate, 40% masking rate, and 80% masking rate. First, in the software development phase, we analyze the code search task. We use a variety of approaches to train DeepCS and CodeBERT. Moreover, we present detailed examples to describe the final data used for model decisions. The analysis shows that the selected studies can find the right code snippets in the absence of prominent representations. Second, code summarization is the foundation of software maintenance. The Transformer-based and CODE-NN can generate an accurate summary based on the masked code snippet. Finally, in the software maintenance phase, we compare two duplicate bug report detection approaches. HINDBR and DWEN can accurately detect duplicate bug reports after masking the content of the description tag. Meanwhile, to more comprehensively assess whether overinterpretation exists in the AI4SE task, we also present examples for a more detailed description, as shown in Figs. 4, 6, and 8. Finally, we show the performance of different models after training with SIS in Figs. 5, 7, and 9. These experimental results all show that PLMs in SE tasks achieve high performance despite the lack of salient features. However, these inputs without salient features, are meaningless. It contains only discrete letters and sparse words. Humans simply cannot read the masked text, let alone understand its meaning. These models do not really understand these SE tasks, but only learn meaningless statistical signals. This is an overinterpretation phenomenon of the SE tasks.

VI. Model-Oriented Overinterpretation Analysis

RQ 2:

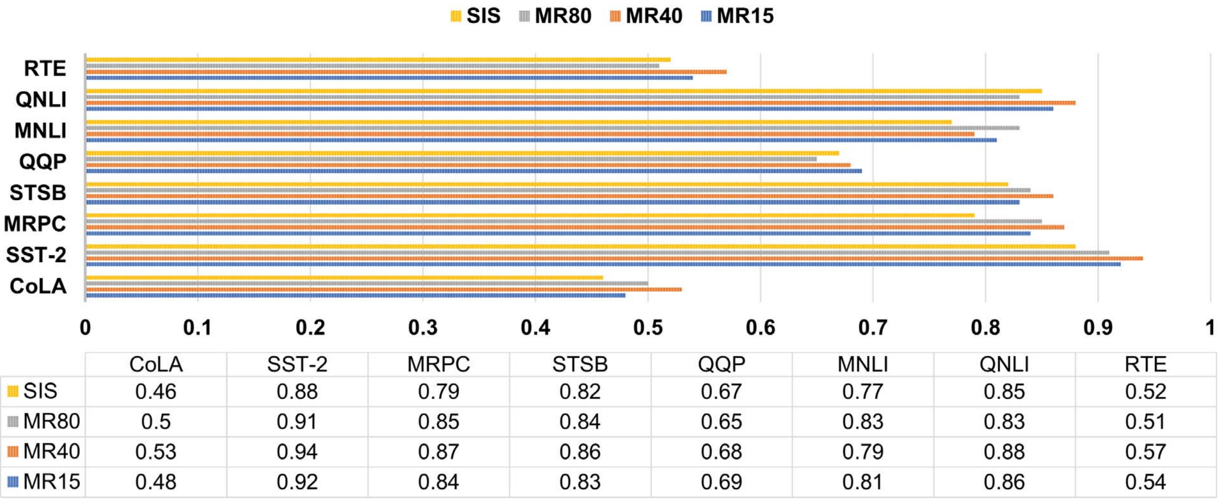
Does overinterpretation depend on software engineering tasks and how prevalent is overinterpretation in PLMs?

For various tasks of SE, researchers not only construct their models but also use well-known PLMs [9], [19], [20]. Pre-trained models are beneficial for downstream NLP tasks and can avoid training a new model from scratch. To demonstrate that overinterpretation is not task-dependent but is prevalent in pre-trained language models, we choose three representative PLMs, GPT [19], BERT [9], and XLNet [20] for evaluation. We perform the same training strategy for each model, i.e., different masking rates (15% masking rate, 40% masking rate, and 80% masking rate) and SIS. We evaluate them with diverse tasks, including, multi-genre natural language inference corpus (MNLI) [54], question-answering natural language inference (QNLI) [55], recognizing textual entailment (RTE), corpus of linguistic acceptability (CoLA) [56], the Stanford Sentiment Treebank (SST-2) [57], the Microsoft Paraphrase corpus (MRPC) [58], the Quora Question Pairs (QQP) [59], and the Semantic Textual Similarity benchmark (STS-B) [60]. These downstream tasks are widely used to evaluate PLMs.

A. GPT

The GPT [19] family is a series of very powerful pre-trained language models proposed by OpenAI, which can achieve stunning results in very complex tasks, such as article generation, code generation, machine translation, etc., without the need for supervised learning for model fine-tuning. In contrast to previous approaches, they make use of task-aware input transformations during fine-tuning to achieve effective transfer while requiring minimal changes to the model architecture. For a new task, GPT requires very little data to understand the requirements of the task and to approach or exceed the state-of-the-art approach.

We conduct two types of experiments using the GPT model, with different masking rates (15% masking rate, 40% masking rate, and 80% masking rate) and SIS. Fig. 10 depicts the metrics of GPT under different masking rates and SIS. In terms of overall performance, the 15% masking rate performs the best and achieves better results in all tasks. For the GPT model, the best-performing task with a masking rate of 15% is SST-2, which can achieve 92%; the worst-performing task is CoLA, with 48%. On the other hand, the performance of the model trained with 40% masking improves in most tasks, such as the SST-2 task and the CoLA task. When the masking rate is increased to 80%, most of the inputs have been masked, and the performance of the model decreases somewhat, but not significantly. Meanwhile, the results show that the performance does not decrease but improves as the masking rate increases. In the case of high masking rates, the GPT can still make effective judgments, and the results do not differ much between different masking rates. After training the model with SIS, the model



	CoLA	SST-2	MRPC	STSB	QQP	MNLI	QNLI	RTE
■ SIS	0.46	0.88	0.79	0.82	0.67	0.77	0.85	0.52
■ MR80	0.5	0.91	0.85	0.84	0.65	0.83	0.83	0.51
■ MR40	0.53	0.94	0.87	0.86	0.68	0.79	0.88	0.57
■ MR15	0.48	0.92	0.84	0.83	0.69	0.81	0.86	0.54

Fig. 10. Performance of GPT under different masking rates and SIS. “Orange” represents the results of the SIS trained model, “Blue” represents the results of the 15% masking rate trained model, and similarly, “Red” and “Gray” represent the results of 40% and 80% masking rate, respectively.

is still able to perform each task effectively, with results that vary within 10% from the other models. This result proves the effectiveness of the sufficient input subset algorithm. It also indicates that the GPT model produces an overinterpretation phenomenon of the input dataset. The data contained in the SIS is extremely sparse, and some words are not associated at all. Therefore, there is an overinterpretation phenomenon for the dataset by using GPT.

B. BERT

In October 2018, Google AI published their BERT [9], a pre-trained language representation model. It emphasizes that BERT uses the new masked language model (MLM) so that deep bidirectional language representations can be generated. The pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

Fig. 11 shows the results of BERT with different masking rate strategies (15% masking rate, 40% masking rate, and 80% masking rate) and SIS. As shown in Fig. 11, the graphs of the 15% masking rate strategy and the 40% masking rate strategy are largely merged, and the graph of the 40% masking rate masks the graph of the 15% masking rate. This result shows that the performance of the model does not decrease with increasing masking rate, but improves when applying 15% masking and 40% masking to the dataset. The performance of the BERT model is improved with reduced input datasets. Moreover, when the masking rate is increased to 80%, the performance of BERT decreases, but it is still better than the model trained with a 15% masking rate. The higher masking rate means that the input dataset to the model contains fewer data, and the knowledge learned by the classifier is relatively lower. However, the BERT can still achieve good results. Even when keeping only 20% of the input dataset, it still outperforms the 15% masking rate model. To validate the final data or features used by BERT, we extract sufficient input subsets to train the model using the SIS algorithm. In the case of extremely sparse data, BERT can perform each task accurately. This fact shows that the BERT model can learn “knowledge” that humans cannot understand. Meanwhile, the experimental results show that deep learning is not affected by this aspect and still learns useful information to make final judgments at high masking rates. This result suggests that BERT does not learn real knowledge, but rather statistical signals for the dataset. Therefore, BERT is suffering from overinterpretation.

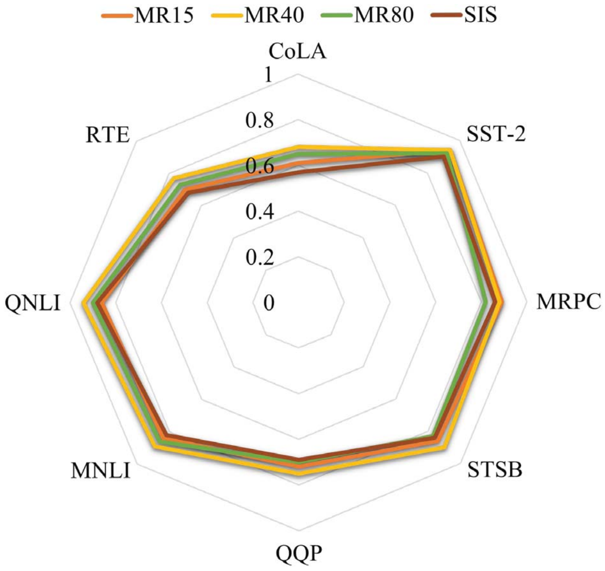


Fig. 11. Performance of BERT. “Orange” represents the performance in a 15% masking rate strategy. “Gold” represents the performance at 40% masking rate strategy. “Green” denotes the performance of BERT under 80% masking rate. “Brown” shows the performance of the SIS-trained BERT.

C. XLNet

Unlike BERT, XLNet [20] is essentially the idea of using an autoregressive language model to encode bi-directional semantic information simultaneously, which can overcome the

TABLE VI
XLNet Under Different Masking Rates and SIS. "MR15" Represents a 15% Masking Rate, and Similarly, "MR40" and "MR80" Represent a 40% and 80% Masking Rate, Respectively

Works	Pre-training	Metrics							
		MNLI	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE
XLNet	MR15	0.90	0.92	0.94	0.97	0.69	0.92	0.90	0.85
	MR40	0.93	0.93	0.92	0.95	0.73	0.94	0.95	0.89
	MR80	0.91	0.92	0.91	0.94	0.70	0.93	0.92	0.87
	SIS	0.86	0.88	0.90	0.94	0.66	0.88	0,85	0.81

problems of missing dependencies and inconsistent training/fine-tuning that exist in BERT. Furthermore, XLNet integrates ideas from Transformer-XL, the state-of-the-art autoregressive model, into pre-training.

Table VI shows the metrics of XLNet under different training strategies, 15% masking rate, 40% masking rate, and 80% masking rate. XLNet achieves good performance in all eight downstream tasks. By comparing the 15% masking rate model with the 40% masking rate model, all downstream tasks improve with increasing masking rate except for QNLI and SST-2. Among them, MRPC shows the most significant improvement at 5%, with the other tasks showing improvements between 2% and 3%. Then, comparing the 80% masking rate with the 40% masking rate, the performance of XLNet decreases. However, there are still some improvements in XLNet compared to the 15% masking rate. The results indicate that training XLNet using the masking rate strategy will further improve the model's capability. It means that XLNet can learn the "knowledge" to support its final decision after losing 80% of the input dataset. The remaining 20% of the dataset is beyond human comprehension, let alone using it to make some series of decisions, such as searching for codes, etc. The performance of XLNet trained with SIS decreases in all tasks, but it is close to the other masking rate models, i.e., results stay within 10%. This result proves that the SIS extracted by the SIS algorithm is valid. Meanwhile, the SIS contains much less data than the full dataset and lacks many salient features. The XLNet, however, can make accurate decisions from these data. XLNet has its own unique way of learning to understand the dataset, and it is the unknown way of understanding that leads to the overinterpretation of XLNet.

Finding 2: Overinterpretation is prevalent in PLMs. PLMs trained in different ways behave similarly, where the input varies widely, from a few characters to full queries.

VII. Discussion

RQ 3:

What is the impact of overinterpretation? What are the challenges in mitigating overinterpretation in general and how to mitigate overinterpretation?

A. *Impact*

The above experiments show that overinterpretation appears not only in SE tasks but also in PLMs. The essence of both types of experiments is that the input size is reduced and the final result is consistent with the full input. This fact suggests that the presence of overinterpretation allows PLMs to improve the efficiency of training by reducing the input. However, this model, which achieves ultra-high results on a very small set of data, is wrong [10]. The drawback is that PLMs learn contents that are incomprehensible from a human perspective, or more accurately they learn unique statistical signals in the dataset. Moreover, if model decisions are made based on statistical signals alone they can have serious consequences in terms of misclassification. PLMs make incorrect decisions when different datasets produce the same statistical signal. For example, the results of code search do not match and delay software development; the generated code summaries are inaccurate and increase the cost of software post-maintenance; duplicate bug reports are retained or new bug reports are misclassified as duplicates, resulting in bugs that cannot be fixed in time. Therefore, we summarize several implications of overinterpretation.

1. **Software design and architecture:** Overinterpretation can lead to misinterpretation of requirements or design specifications. If developers or architects make unwarranted assumptions or extrapolate beyond what is supported by the requirements, it can result in a flawed software design. This can lead to inefficiencies, poor system performance, or even critical failures in the software system.
2. **Implementation and coding errors:** Overinterpretation can influence the implementation phase of software development. If developers misunderstand the requirements or overgeneralize the expected behavior, it can result in coding errors and bugs. Overinterpretation can lead to incorrect logic, inadequate error handling, or improper handling of edge cases, all of which can compromise the quality and reliability of the software.
3. **Maintenance and evolution challenges:** Overinterpretation can make software maintenance and evolution more challenging. If the original design or implementation is based on overinterpreted requirements, future modifications or enhancements may become more complex and error-prone. Over time, these accumulated overinterpretations can lead to a codebase that is difficult to understand, modify, or extend, impeding the agility and maintainability of the software system.
4. **Communication and collaboration issues:** Overinterpretation can lead to miscommunication and misunderstandings within software development teams. If

different team members or stakeholders have conflicting interpretations of requirements or design decisions, it can create confusion and delays in the development process. Effective communication, clarification of expectations, and documentation of clear specifications are important in mitigating the risks of overinterpretation in SE tasks.

5. **Quality assurance challenges:** Overinterpretation can complicate the testing and quality assurance process. If testers rely on overinterpreted requirements or assumptions, they may overlook critical test cases or scenarios that are not adequately covered. This can result in incomplete test coverage, leaving potential defects undiscovered and increasing the risk of software failures in production.
6. **Project delays and cost overruns:** Overinterpretation can contribute to project delays and cost overruns. If incorrect assumptions or overinterpretations are discovered late in the development process, it may require significant rework, refactoring, or redesign efforts to rectify the issues. This can lead to project schedule slippage and increased development costs.

pgtagMeanwhile, to further verify the impact of overinterpretation, we designed a survey to collect developers' views on overinterpretation. The survey contains the following questions: 1) *Do you frequently use SE techniques (such as code search, code summarization, duplicate bug report detection, etc.) in developing and maintaining software?* 2) *Have these techniques ever resulted in errors with serious consequences?* 3) *Can you understand the real meaning of the query after masking?* 4) *Please choose five at random from the 20 examples. Please read the five masked queries carefully, can you understand and find the corresponding code snippet?*

We invite a total of 20 experienced developers who have more than five years of software development and maintenance experience to participate in our survey. Each developer randomly selects five samples to ensure diversity of scenarios and contexts. For the first question, we collect responses from all 20 developers. 16 out of the 20 developers (80%) report using code search frequently in their development process.

For the second question, we ask the developers to provide specific examples of errors that have occurred due to the use of SE techniques. The developers share their experiences, and the responses are analyzed to identify the frequency and severity of errors encountered. 12 out of the 20 developers (60%) report instances where SE techniques have led to errors with serious consequences.

For the third question, we present each developer with five masked queries and ask them to indicate whether they can understand the real meaning of each query after masking. This allows us to gauge the level of interpretability of the masked queries. 17 out of the 20 developers (85%) report that they are unable to understand the meaning of the masked queries.

For the fourth question, we provide five different examples of masked queries and ask the developers to identify the correct code fragment that would be queried based on the masked text. This helps us assess the effectiveness of code search when faced with masked queries. Only 4 out of the 20 developers (20%) correctly identify the intended code fragment in the given examples.

By including a larger sample size and randomly selecting examples for each developer, we can obtain a more representative and diverse set of responses. This approach allows us to gather a wider range of perspectives and experiences related to overinterpretation in code search, code summarization, and duplicate bug report detection.

B. Analysis and Mitigation

Our research aims to discover, explain, and provide initial mitigations for this problem. Despite these three tasks are fully difficult to help understand the true root cause of the over-interpretation of PLM applied in all software engineering domains. However, we have concluded the reason for overinterpretation on the basis of the scheme proposed in this article and the experiments conducted.

To provide substantial evidence for the existence of overinterpretation, we design two strategies: the masking ratio strategy and the subset importance sampling (SIS) strategy. Results obtained from different tasks and models, each using different masking ratios, are presented in Tables II–V in detail. These findings consistently show that the model's performance remains relatively consistent and tightly consistent across different masking rates, suggesting that it relies on a small set of features when making judgments.

Furthermore, to further investigate this hypothesis, we employ the SIS strategy to extract subsets from the data and train the model for testing. The results depicted in Figs. 5, 7, and 9 show that models trained on these extracted subsets perform comparable to models trained on the full dataset. These experimental results show that pre-trained language models often over-interpret small and insignificant patterns present in the data. These patterns consist of subsets of characters that serve as strong evidence for model predictions. Despite the lack of salient features, these sparse subsets contain statistical signals that can be effectively generalized from training data to test data.

Meanwhile, it is worth noting that different models achieve similar results when based on different sufficient subsets of inputs. This observation suggests that the behavior of pre-trained language models is significantly influenced by the characteristics of the training data. **Based on our analysis, we can conclude that the root cause of overinterpretation lies in the presence of spurious statistical signals in the training data.**

However, we duly acknowledge the inherent limitations of this study and recognize the need for further research to delve more deeply into potential factors leading to overinterpretation. In future work, we are eager to conduct a more comprehensive and in-depth investigation to gain a deep understanding of overinterpretation and its implications.

Meanwhile, the following issues affect the mitigation of overinterpretation. First, overinterpretation is not well understood and studied at present. It can be easily misunderstood. Second, overinterpretation is not easily detected. The overinterpretation may arise from the true statistical signal in the underlying dataset distribution. Thus, overinterpretation can be harder to diagnose

TABLE VII
THE PERFORMANCE OF WHOLE WORD MASKING UNDER QNLI

Whole Word Masking	GPT	BERT	XLNet
15%	0.81	0.87	0.91
40%	0.85	0.92	0.90
80%	0.83	0.89	0.87

as it admits decisions that are made by statistically valid criteria, and models that use such criteria can excel at benchmarks. Finally, PLMs are black-box models. Then, we find that the whole word mask strategy and ensembling can mitigate overinterpretation. Both of them can enrich the input dataset.

Whole Word Masking Strategy: If a subword is masked, the other parts of the same word are also masked, i.e., the whole word masking strategy [61]. We describe and summarize the experimental results in this article. We still use three mask rates of 15%, 40%, and 80% to train the GPT [19], BERT [9], and XLNet [20]. Whole word masking strategy can effectively improve the integrity of the words after masking. Meanwhile, the readability of the text after masking is increased and the number of meaningless words decreases (Table VII). Whole word masking strategy includes the following steps:

1. **Data Preprocessing:** Prepare your dataset for training by tokenizing the code snippets, bug reports, or summaries into individual words or subword units.
2. **Masking:** Instead of masking individual tokens, apply whole word masking during the training phase. Replace entire words with a special "mask" token to encourage the model to focus on the contextual meaning of complete words.
3. **Model Training:** Train your code search, code summarization, or duplicate bug report detection model using the modified dataset with whole word masking. This trains the model to consider the context of complete words rather than relying solely on individual tokens.
4. **Evaluation:** Evaluate the performance of your trained model on appropriate evaluation datasets or metrics for the specific task. Compare the results with models trained without whole word masking to assess the impact on overinterpretation and task performance.

Ensembling: It is known to improve classification performance [62], [63]. But it can also be used to increase the SIS size, hence mitigating overinterpretation. We observe that SIS subsets are generally not transferable from one model to another i.e., an SIS for one model is rarely an SIS for another. Thus, different models rely on different independent signals to arrive at the same prediction. We find that ensembling uniformly increases test accuracy as expected but also increases the SIS size (Fig. 12). The ensembling strategy includes the following steps:

1. **Individual model training:** Multiple models are trained independently using the same architecture for each case.
2. **Ensemble classifier construction:** From the individually trained models, we construct ensemble classifiers by grouping different models together.
3. **Emphasizing diversity:** To promote diversity within the ensemble, we incorporate various techniques, such as data augmentation, to introduce variations in the training process.
4. **Hyperparameter optimization:** We fine-tune the hyperparameters of each model, including learning rates, regularization methods, and optimization algorithms, to maximize their individual and collective potential.
5. **Subset training data:** Each model is trained on a well-curated subset of the corresponding task's training data, ensuring comprehensive coverage of relevant samples.
6. **Objective-driven design:** Our primary objective is to develop ensemble classifiers that excel in their respective tasks, efficiently retrieving code snippets for code search, generating informative and concise summaries for code summarization, or accurately detecting duplicate bug reports.

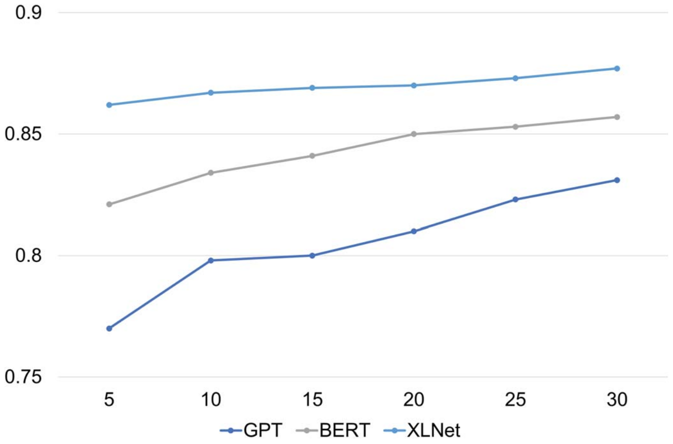


Fig. 12. SIS size on MNLI as the number of characters varies. The horizontal axis represents the number of characters contained in the SIS, and the vertical axis represents the MNLI.

> **Finding 3:** Overinterpretation makes the model learn only the statistical signal and ignore the crucial features. Whole word masking and ensembling are found to mitigate overinterpretation.

VIII. THREATS TO VALIDITY

Internal threats to validity. First, the inputs used for each scenario and model are different, and we have chosen to use the dataset that they originally used rather than the new uniform dataset. In the future, we will experiment on several different datasets. In this way, we will verify that overinterpretation does not depend on unique datasets. Second, although we have experimented and validated on pre-trained language models and SE tasks, there are compatibility challenges when comparing masked and SIS models with the original models. This is because the masked/SIS model is trained only on the masked/SIS data, which is different from the data used to train the original model. Therefore, the masked/SIS model should also have the ability to handle raw inputs. A masked model should be able to predict the full inputs and compare them to their respective masked versions. In future work, we will focus on refining the proposed solution and developing methods to effectively overcome the compatibility challenge between the masked/SIS model and the original model. By ensuring that the masked/SIS model can handle the original input and is consistent with the original model, we can facilitate more accurate comparisons and evaluations.

External threats to validity. Our study has a limited scope and only three types of AI4SE tasks (code search [13], [14], code summarization [15], [16], and duplicate bug report detection [17], [18]) and three pre-trained language models (GPT [19], BERT [9], and XLNet [20]) have been selected. These tasks are only a part of the SE tasks. There are many different types of tasks that have not been studied and demonstrated for overinterpretation. In the future, we will select more various AI4SE tasks and models to validate our scheme.

IX. Related Work

AI4SE In recent years, many empirical studies have focused on AI and SE tasks. But these researches are limited to only one aspect. Wu et al. [64] conduct an empirical comparison and analysis of four representative deep learning frameworks with three unique contributions. Christian et al. [65] conduct an empirical study to investigate the effect of dropout and batch normalization on training deep learning models. Hu et al. [66] first conduct a systemically empirical study to reveal the impact of the retraining process and data distribution on model enhancement. Abdallah et al. [67] propose an evaluation of vulnerability detection performance on source code representations and evaluates how DL strategies can improve them. Du et al. [68] present the first comprehensive empirical study on fault triggering conditions in three widely-used deep learning frameworks. Pan et al. [69] perform empirical studies using DL models in cross-version and cross-project software defect prediction to investigate if using a neural language model could improve prediction performance. However, all these studies failed to examine the flaws of deep learning models themselves.

Meanwhile, many researchers have started to focus on explainable AI for SE tasks. Rabin et al. [70] propose a model-agnostic approach to identify critical input features for models in code intelligence tools, by drawing on software debugging research, and then exploring and analyzing the models. Cito et al. [71] explore counterfactual explanations for models of source code to help developers understand and use the model. Li et al. [72] propose IVDetect, an interpretable vulnerability detector that uses AI to detect vulnerabilities while providing an interpretation of the vulnerability detector. These works are concerned with how to interpret the decisions made by the model. However, we explore a potential flaw of the model, i.e. overinterpretation.

PLMs Previous studies study cross-modal pre-trained language models [73] and the robustness of pre-trained language models to spurious correlations [74]. Moreover, many studies focus on the design of pre-trained models and the enhancement of models [75]. The first generation pretrained models aim to learn good word embeddings, they are usually very shallow for computational efficiencies, such as Skip-Gram [76] and GloVe [77]. The second generation pre-trained models focus on learning contextual word embeddings, such as CoVe [78], ELMo [79]. All the above works do not focus on some defects hidden by the models, such as overinterpretation. This study explains overinterpretation by investigating different pre-trained language models.

X. Conclusion

Deep learning-based natural language processing techniques are becoming increasingly popular for researchers to solve various tasks in SE. This article constructs the first comprehensive study to reveal the overinterpretation in PLMs of AI4SE tasks. By investigating the most representative AI4SE tasks as well as PLMs, we identify the existence of overinterpretation in these models. The wide presence of these problems motivates future research to further tackle the overinterpretation of deep learning.

In the future, we will design a new evaluation scheme to identify overinterpretation and thus help researchers refine their models.

Data Availability

The data underlying this article are available in https://doi.org/10.17632/yz3gnwvzfm.3.

References

[1] B. Gezici and A. K. Tarhan, “Systematic literature review on software quality for AI-based software,” *Empirical Softw. Eng.*, vol. 27, pp. 1–65, 2022.

[2] A. M. Huang, G. S. Deng, J. Hu, and Y. J. Huang, “Study on CMM-based software quality assurance process improvement—A case of the educational software quality assurance model,” *Adv. Mater. Res.* vol. 1049, 2014, pp. 2032–2036.

[3] S.-Y. Chen, Y.-S. Su, Y.-Y. Ku, C.-F. Lai, and K.-L. Hsiao, “Exploring the factors of students’ intention to participate in AI software development,” *Lib. Ti Tech*, vol. 15, pp. 1–17, 2022.

[4] M.-P. Duarte, A.-G. Domínguez, and A. Balderas, “Assessment in software development for competitive environments: An AI strategy development case study,” *Electronics (Basel)*, vol. 10, pp. 1566–1584, 2021.

[5] Y. Sheoraj and R. K. Sungkur, “Using AI to develop a framework to prevent employees from missing project deadlines in software projects—Case study of a global human capital management (HCM) software company,” *Adv. Eng. Softw.*, vol. 170, pp. 103143–103156, 2022.

[6] Z.-M. Zheng and A.-H. Ren, “Earned value method and application in software project management,” *Comput. Eng. Des.*, vol. 29, pp. 4302–4304, 2008.

[7] G. Z. Espinoza, R. M. Angelo, P. R. Oliveira, and K. M. Honorio, “Evaluating Deep Learning models for predicting ALK-5 inhibition,” *PLoS One*, vol. 16, pp. 1–16, 2021.

[8] Y. Qu and H. Yin, “Evaluating network embedding techniques’ performances in software bug prediction,” *Empirical Softw. Eng.*, vol. 26, pp. 60–104, 2021.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Conf. North Amer. Chapter Assoc. Comput. Ling.: Hum. Lang. Technol. (NAACL-HLT)*, 2019, pp. 1–16.

[10] B. Carter, S. Jain, J. W. Mueller, and D. Gifford, “Overinterpretation reveals image classification model pathologies,” in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34. Curran Associates, 2021, pp. 15395–15407. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/8217bb4e7fa0541e0f5e04fea764ab91-Paper.pdf

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet?” *IEEE Trans. Softw. Eng.*, vol. 48, pp. 3280–3296, 2022.

[12] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why should I trust you?': Explaining the predictions of any classifier," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Min.*, 2016, pp. 1135–1144.

[13] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2018, pp. 933–944.

[14] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics*, Association for Computational Linguistics, 2020, pp. 1536–1547.

[15] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proc. 58th Annu. Meet. Assoc. Comput. Lingu.* Stroudsburg, PA, USA: Association for Computational Linguistics, 2020, pp. 4998–5007.

[16] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Annu. Meet. Assoc. Comput. Ling.*, vol. 4, 2016, pp. 2073–2083.

[17] G. Xiao, X. Du, Y. Sui, and T. Yue, "HINDBR: Heterogeneous information network based duplicate bug report prediction," in *Proc. IEEE Int. Symp. Softw. Rel. Eng.*, vol. 2020, 2020, pp. 195–206.

[18] A. Budhiraja, K. Dutta, R. Reddy, and M. Shrivastava, "Poster: DWEN: Deep word embedding network for duplicate bug report detection in software repositories," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.: Companion (ICSE-Companion)*, vol. 137351, 2018, pp. 193–194.

[19] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," pp. 1–12, 2018.

[20] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized autoregressive pretraining for language understanding," in *Proc. 33rd Conf. Neural Inf. Process. Syst.*, 2019, pp. 1–18.

[21] A. Wettig, T. Gao, Z. Zhong, and D. Chen, "Should you mask 15% in masked language modeling?," in *Proc. 17th Conf. Eur. Chapter Assoc. Comput. Ling.*, Dubrovnik, Croatia, 2023, pp. 2977–2992, 2022.

[22] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2010, pp. 43–52.

[23] L. Pollock, K. Vijay-Shanker, E. Hill, G. Sridhara, and D. Shepherd, "Natural language-based software analyses and tools for software maintenance," in *International Summer School on Software Engineering* (Lecture Notes in Computer Sciences), vol. 7171. Salerno, Italy: Springer, 2013, pp. 94–125.

[24] J. Fowkes et al., "Autofolding for source code summarization," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1095–1109, Dec. 2017.

[25] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, pp. 1527–1554, 2006.

[26] M. Kaneko, M. Mita, S. Kiyono, J. Suzuki, and K. Inui, "Encoder-decoder models can benefit from pre-trained masked language models in grammatical error correction," in *Proc. Annu. Meet. Assoc. Comput. Ling.*, 2020, pp. 4248–4254.

[27] L. Kryeziu and V. Shehu, "A survey of using unsupervised learning techniques in building masked language models for low resource languages," in *Proc. 11th Mediterr. Conf. Embedded Comput.*, Piscataway, NJ, USA: IEEE, 2022, pp. 1–6.

[28] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.

[29] Y. Levine et al., "PMI-Masking: Principled masking of correlated spans," in *Proc. 9th Int. Conf. Learn. Represent., Virtual Event*, Austria. OpenReview.net, 2020, pp. 1–13.

[30] J. Perez, J. L. Flores, C. Blum, J. Cerquides, and A. Abuin, "Optimization techniques and formal verification for the software design of Boolean algebra based safety-critical systems," *IEEE Trans. Ind. Inform.*, vol. 18, no. 1, pp. 620–630, Jan. 2022.

[31] J. Maletic and R. Reynolds, "A tool to support knowledge based software maintenance: The Software Service Bay," in *Proc. Int. Conf. Tools Artif. Intell.*, 1994, pp. 11–17.

[32] H. Niu, I. Keivanloo, and Y. Zou, "Learning to rank code examples for code search engines," *Empirical Softw. Eng.*, vol. 22, pp. 259–291, 2016.

[33] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what I mean—Code search and idiomatic snippet synthesis," in *IEEE/ACM Int. Conf. Softw. Eng.*, vol. 14, 2016, pp. 357–367.

[34] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for JavaScript frameworks," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 690–701.

[35] F. Lv, H. Zhang, J.-G. Lou, S. Wang, D. Zhang, and J. Zhao, "CodeHow: Effective code search based on API understanding and extended Boolean model (E)," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 260–270.

[36] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *Proc. Int. Conf. Softw. Eng.*, 2011, pp. 111–120.

[37] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining StackOverflow to turn the IDE into a self-confident programming prompter," in *Proc. Work. Conf. Min. Softw. Repositories*, 2014, pp. 102–111.

[38] S. Robertson, H. Zaragoza, and M. Taylor, "Simple BM25 extension to multiple weighted fields," in *Proc. Int. Conf. Inf. Knowl. Manage.*, 2004, pp. 42–49.

[39] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 253–262.

[40] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 115–124.

[41] M. Abbes, F. Khomh, Y.-G. Gueéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension," in *Proc. Eur. Conf. Softw. Maintenance Reeng.*, 2011, pp. 181–190.

[42] C. Politowski et al., "A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension," *Inf. Softw. Technol.*, vol. 122, pp. 106278–106292, 2020.

[43] S. Olbrich, D. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 2009, pp. 390–400.

[44] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 684–702, Sep./Oct. 2009.

[45] N. Noughi, S. Hanenberg, and A. Cleve, "An empirical study on the usage of SQL execution traces for program comprehension," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion*, 2017, pp. 47–54.

[46] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for? A benchmark and an empirical study on code search with natural-language queries," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng.*, 2020, pp. 344–354.

[47] P. W. McBurney and C. McMillan, "An empirical study of the textual similarity between source code and source code summaries," *Empirical Softw. Eng.*, vol. 21, pp. 17–42, 2014.

[48] R. Hao, Y. Li, Y. Feng, and Z. Chen, "Are duplicates really harmful? An empirical study on bug report summarization techniques," *J. Softw.: Evol. Process*, vol. 24, pp. 1–25, 2022.

[49] B. Carter, J. Mueller, S. Jain, and D. Gifford, "What made you do this? Understanding black-box decisions with sufficient input subsets," in *Proc. 22nd Int. Conf. AISTATS*, vol. 89, 2018, pp. 1–35.

[50] N. Safdari, "Learning to rank relevant files for bug reports using domain knowledge, replication and extension of a learning-to-rank approach," Thesis, GCCIS, Rochester Inst. Technol., New York, NY, USA 2018, pp. 1–53.

[51] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. Annu. Meet. Assoc. Comput. Ling.*, 2002, pp. 311–318.

[52] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. Intrinsic Extrinsic Eval. Meas. Mach. Transl. Summarization*, 2005, pp. 65–72.

[53] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Proc. Workshop Text Summarization Branches Out, Post-Conf. Workshop ACL*, 2004, pp. 74–81.

[54] N. Nangia, A. Williams, A. Lazaridou, and S. R. Bowman, "The RepEval 2017 Shared Task: Multi-genre natural language inference with sentence representations," *2nd Workshop Eval. Vector-Space Represent. NLP*, 2017, pp. 1–10.

[55] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proc. 2016 Conf. Empirical Methods Natural Lang. Process.*, Austin, TX, USA, 2016, pp. 2383–2392.

[56] A. S. A. Warstadt and S. R. Bowman, "Corpus of linguistic acceptability," 2018. Accessed: Jun. 5, 2022. [Online]. Available: http://nyu-mll.github.io/cola

[57] R. Socher et al., "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proc. EMNLP*, vol. 1631, 2013, pp. 1631–1642.

[58] W. Dolan and C. Brockett, "Automatically constructing a corpus of sentential paraphrases," in *Proc. Int. Workshop Paraphrasing*, 2022, pp. 1–8.

[59] X. Z. Z. Chen, H. Zhang, and L. Zhao, "Quora question pairs," 2018. Accessed: May 14, 2022. [Online]. Available: https://data.quora.com/First-QuoraDataset-Release-Questio-Pairs

[60] D. Cer, M. Diab, E. Agirre, I. Lopez-Gazpio, and L. Specia, "SemEval-2017 Task 1: Semantic textual similarity—Multilingual and cross-lingual focused evaluation." in *Proc. 11th Int. Workshop Semantic Eval.* Vancouver, BC, Canada: Association for Computational Linguistics, 2017, pp. 1–14.

[61] Y. Cui, W. Che, T. Liu, B. Qin, and Z. Yang, "Pre-training with whole word masking for Chinese BERT," *IEEE/ACM Trans. Audio, Speech, Language Process.*, vol. 29, pp. 3504–3514, 2021.

[62] K.-S. Goh, E. Chang, and K.-T. Cheng, "SVM binary classifier ensembles for image classification," in *Proc. Int. Conf. Inf. Knowl. Manage.*, 2001, pp. 395–402.

[63] C. Ju, A. Bibaut, and M. van der Laan, "The relative performance of ensemble methods with deep convolutional neural networks for image classification," *J. Appl. Stat.*, vol. 45, pp. 2800–2818, 2018.

[64] Y. Wu et al., "A comparative measurement study of deep learning as a service framework," *IEEE Trans. Services Comput.*, vol. 15, pp. 551–566, 2022.

[65] C. Garbin, X. Zhu, and O. Marques, "Dropout vs. batch normalization: An empirical study of their impact to deep learning," *Multimedia Tools Appl.*, vol. 79, pp. 12777–12815, 2020.

[66] Q. Hu et al., "An empirical study on data distribution-aware test selection for deep learning enhancement," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, pp. 1–30, 2022.

[67] A. O. A. Semasaba, W. Zheng, X. Wu, S. A. Agyemang, T. Liu, and Y. Ge, "An empirical evaluation of deep learning-based source code vulnerability detection: Representation versus models," *J. Softw.: Evol. Process*, vol. 24, pp. 1–23, 2022.

[68] X. Du, Y. Sui, Z. Liu, and J. Ai, "An empirical study of fault triggers in deep learning frameworks," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 4, pp. 2696–2712, Jul./Aug. 2023.

[69] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using CodeBERT model," *Appl. Sci.*, vol. 11, pp. 4793–4813, 2021.

[70] M. R. I. Rabin, V. J. Hellendoorn, and M. A. Alipour, "Understanding neural code intelligence through program simplification," 2021, pp. 441–452.

[71] J. Cito, I. Dillig, V. Murali, and S. Chandra, "Counterfactual explanations for models of code," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2022, pp. 125–134.

[72] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 292–303.

[73] Z. Zeng and W. Mao, "A comprehensive empirical study of vision-language pre-trained model for supervised cross-modal retrieval," 2022, *arXiv:2201.02772*.

[74] L. Tu, G. Lalwani, S. Gella, and H. He, "An empirical study on robustness to spurious correlations using pre-trained language models," *Trans. Assoc. Comput. Ling.*, vol. 8, pp. 621–633, 2020.

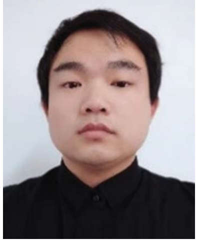
[75] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 2017, 2017, pp. 5999–6009.

[76] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *27th Annual Conference on Neural Information Processing Systems*, 2013, pp. 1–9.

[77] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empirical Methods Nat. Lang. Process.*, 2014, pp. 1532–1543.

[78] B. McCann, J. Bradbury, C. Xiong, and R. Socher, "Learned in translation: Contextualized word vectors," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 6297–6308.

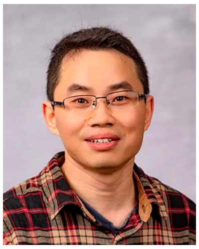
[79] M. E. Peters et al., "Deep contextualized word representations," in *Proc. Conf. North Am. Chap. Assoc. Comput. Ling.: Hum. Lang. Technol.*, vol. 1, 2018, pp. 2227–2237.



Yao Li received the M.E. degree from Shantou University, in 2021. He is working toward the Ph.D. degree with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST), under the supervision of Prof. Tao Zhang. His research interests lie in software engineering and malware detection.

Tao Zhang (Senior Member, IEEE) received the B.S. degree in automation, the M.Eng. degree in software engineering from Northeastern University, China, and the Ph.D. degree in computer science from the University of Seoul, South Korea. After that, he spent one year with the Hong Kong Polytechnic University as a Postdoctoral Research Fellow. Currently, he is an Associate Professor with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST). Before joining MUST, he was a faculty member with Harbin Engineering University and Nanjing University of Posts and Telecommunications, China. He published more than 90 high-quality papers at renowned software engineering and security journals and conferences. He served as the General Chair of SANER 2023 and the PC members of several top-tier SE conferences such as FSE and ASE. He is a senior member of ACM.

Xiapu Luo received the Ph.D. degree in computer science from Hong Kong Polytechnic University and then spent two years with the Georgia Institute of Technology as a Postdoctoral Research Fellow. He is a Full Professor with the Department of Computing, Hong Kong Polytechnic University. His current research interests include mobile/IoT security and privacy, blockchain, network security and privacy, software engineering, and Internet measurement. He has received seven best paper awards (e.g., INFOCOM'18, ISPEC'17, and ISSRE'16) and one paper received best paper nomination (i.e., ESEM'19).



Haipeng Cai (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Notre Dame, IN, USA. He is currently a Huie-Rogers Endowed Chair in computer science and an Associate Professor with the School of Electrical Engineering and Computer Science at Washington State University, Pullman. His research generally lies in software engineering and software security, with a focus on program analysis and machine/deep learning for security applications to multilingual software, distributed systems, and mobile apps. He has published in top venues in software engineering and security. He is a senior member of ACM and an Associate Editor of ACM TOSEM. More information is available at https://chapering.github.io/.

Sen Fang received the M.Sc. degree in electronics and communication engineering from Central China Normal University, in 2020. He is working toward the Ph.D. degree with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST), under the supervision of Prof. Tao Zhang. His research interests lie in software engineering and NLP, particularly using NLP technologies to build effective models for representing the source code.

Dawei Yuan received the bachelor's degree in network engineering from Nanjing University of Post and Telecommunication (NJUPT) and the master's degree in software engineering from the University of Science and Technology of China (USTC). He is working toward the Ph.D. degree in artificial intelligence from Macau University of Science and Technology (MUST). Before embarking on the doctoral journey, he accumulated 4 years of professional experience as a Software Engineer at United Imaging Healthcare (UIH).