International Journal of Software Engineering
and Knowledge Engineering
Vol. 35, No. 6 (2025) 835–860
DOI: 10.1142/S0218194025500238

RESEARCH NOTES

Multiclass Classification for Self-Admitted Technical Debt via Large Pre-Trained Language Model

Yiyang Du [*,‡], Xingguang Yang [*,§], Zhenyu Shu [*,¶], Zijie Huang [†,‖], Gang Wang [*,**] and Libo Xu [*,††]

[*]*School of Computer and Data Engineering*
NingboTech University Ningbo
Zhejiang Province 315100, P. R. China

[†]*Shanghai Key Laboratory of Computer Software Testing and Evaluating*
Shanghai Development Center of Computer Software Technology
Shanghai 201112, P. R. China
[‡]*2781410051@qq.com*
[§]*Xingguang2955@163.com*
[¶]*shuzhenyu@nit.zju.edu.cn*
[‖]*huangzj@sscenter.sh.cn*
[**]*smile588@sina.com*
[††]*xlb@nbt.edu.cn*

Received 18 January 2025
Revised 12 March 2025
Accepted 22 April 2025
Published 19 June 2025

Technical debt refers to suboptimal solutions adopted for short-term goals. Self-admitted technical debt (SATD) is the debt that is explicitly marked through comments or documentation, making it traceable. Multi-classification of SATD helps developers understand different debt types and improve efficiency. This paper proposes a SATD multi-classification method based on Fine-Tuning the GPT-3.5-turbo model for SATD prediction. This study uses a public dataset containing 10 projects with code comments. We classify design debt, requirement debt, and defect debt and evaluate our method's performance. The experimental results show that compared to the best baseline model, our method achieves average improvements of 11.41%, 1.72% and 3.72% in MacroF, MacroP and MacroR metrics, respectively, in the MTO scenario. In the OTO scenario, improvements are 2.33%, 3.70% and 2.18%, respectively. These results indicate that our method has a strong generalization ability in SATD multi-classification and offers a new approach to managing technical debt.

Keywords: Technical debt; self-admitted technical debt; multiclass classification; fine-tuning GPT; software analysis; software quality.

[§]Corresponding author.

1. Introduction

As the scale of software development continues to expand and the complexity of code increases dramatically, technical debt has become one of the key factors affecting the quality, long-term maintainability and development efficiency of software projects [1]. Technical debt stems from imperfect design and implementation decisions made by developers in pursuit of short-term goals or rapid delivery during project progress. Although these decisions can increase development speed in the short term, they will have a profound impact on code quality in the long run, leading to increased maintenance costs and even seriously affecting the scalability and stability of the software [2].

In order to help developers reduce software maintenance costs, automatic detection of technical debt has become a very important part of software maintenance. In the existing research, researchers have developed a variety of methods for detecting technical debt, some of which focus on identifying technical debt based on source code [3–5]. They use analysis tools to extract various indicators related to the source code and use these indicators to determine whether there is a technical debt in the code [3–5]. However, source code-based detection methods usually result in a high false positive rate [1]. To address this problem, some studies have attempted to identify technical debt through code comments, a method called self-admitted technical debt (SATD) [6–8]. Specifically, these studies extract features from code comments and encode them, and then use these features to train a classifier to predict whether each new code comment belongs to SATD [6, 8]. Compared with directly analyzing source code, code comments can better reflect the developer's true intention because technical debt is actively identified by the developer in the comments [1]. This makes this method more reliable in terms of detection accuracy and easier to implement. In addition, the code annotation-based method can be more conveniently combined with text mining technology to further improve the detection efficiency.

Currently, using code comments to identify technical debt is the research focus of technical debt detection, but it mainly focuses on solving the binary classification problem (classifying code comments as "SATD" or "non-SATD"). In fact, different types of SATD may have different effects [1]. For example, Maldonado and Shihab [8] pointed out that identifying different technical debts can help developers better understand technical debts and improve work efficiency.

With the increasing popularity of large language models such as ChatGPT, it has been extended to various fields of artificial intelligence and software engineering, showing good performance [9–11]. However, the accuracy of Fine-Tuning GPT in SATD multi-class detection remains unverified. As the first study on Fine-Tuning GPT for SATD multi-class detection, we discussed the performance of Fine-Tuning GPT in the SATD multi-classification task and compared it with traditional baseline methods.

We used an open source project dataset containing 62,566 code comments provided by Maldonado *et al.* [1], focusing on identifying three types of technical debt (i.e. defect debt, design debt and requirement debt) from raw code comments. In

order to better simulate actual application scenarios, we introduced two scenarios, many-to-one (MTO) prediction and one-to-one (OTO) prediction and used the fine-tuned model to perform multi-classification detection on code comments. Experimental results show that Fine-Tuning GPT achieves an average MarcoF of 0.625, an average MarcoP of 0.650 and an average MarcoR of 0.613 in the MTO scenario, which are 11.40%, 1.72% and 3.72% higher than the best baseline model, respectively. In the OTO scenario, it achieves an average MarcoF of 0.439, an average MarcoP of 0.466 and an average MarcoR of 0.396, which are 5.02%, 3.25% and 2.68% higher than the best baseline model, respectively.

In this paper, we make the following contributions:

- To the best of our knowledge, this is the first work to leverage Fine-Tuning GPT to make three types of predictions on SATD to help developers better understand technical debt.
- We used public datasets to conduct a large number of experiments in MTO and OTO scenarios, comparing the performance of Fine-Tuning GPT and five benchmark models. The experimental results show that the performance of Fine-Tuning GPT is better than that of the benchmark models.

2. Related Work

2.1. *Research on SATD*

In recent years, SATD has attracted widespread attention from researchers. Potdar and Shihab [13] revealed the characteristics of implicit technical debt in developer comments by reading the content of comments, and systematically analyzing the distribution, type and life cycle of SATD, laying the groundwork for the study of SATD. Maldonado *et al.* [1] proposed a complete set of SATD research methods from manual annotation to automatic detection to life cycle management, providing basic tools and data sets for subsequent research. Yan *et al.* [14] focused on change-level SATD detection using software change features. They identified SATD in source code comments, marked the changes that first introduced these comments as changes that introduced technical debt, and extracted 25 code, comment and commit features to build a machine learning model. The model can efficiently identify technical debt at the change level. Huang *et al.* [6] proposed a text mining-based method to automatically detect SATD in open source projects. They extracted useful text features for classification through feature selection technology, and combined classifiers from multiple source projects to create a composite classifier. This composite classifier can efficiently identify SATD annotations in the target project, thereby improving the accuracy and generalization of detection. Ren *et al.* [7] proposed a CNN-based method to identify SATD and achieved good results. Chen *et al.* [15] proposed a multi-classification SATD detection method based on XGBoost. They identified SATD in code comments and further classified them into three types: Defect debt, design debt and implementation debt. To solve the problem of class imbalance, they adopted a data enhancement strategy and used CHI feature

selection technology to extract key features, and finally built a classifier based on XGBoost. Although good results have been achieved in identifying SATD, it is far from enough to identify SATD alone. In order to further improve the efficiency of developers, identifying different types of SATD is an important task [7]. Therefore, identifying different types of SATD is an important supplement to existing research.

2.2. *LLMs for text classification*

Large Language Model (LLM) has demonstrated remarkable capabilities in a variety of natural language understanding tasks, especially in text classification tasks. Sheikhaei *et al.* [16] conducted an empirical study using the Flan-T5 series to validate its effectiveness in the identification and classification of SATD. Brown *et al.* [9] used unsupervised pre-trained GPT-3 to complete various natural language processing tasks, such as translation, question answering and text classification, with only a small number of task examples in the context, demonstrating the powerful generalization of GPT-3 ability. Zhong *et al.* [11] studied the performance difference between ChatGPT and fine-tuned BERT on natural language understanding tasks. Research has found that the fine-tuned version of BERT performs well on task-specific data sets, while ChatGPT can complete a variety of tasks through contextual learning, demonstrating stronger generalization capabilities and few-shot learning capabilities. Although the accuracy of ChatGPT is slightly lower than BERT, its ability to generate explanations and adapt to multi-tasking is outstanding. Li *et al.* [10] studied the performance of ChatGPT and small deep learning models in detecting SATD and proposed a collaborative framework that combines the two. By combining the context analysis capabilities of ChatGPT and the task specialization capabilities of small models, the collaborative framework significantly improves the performance of SATD detection while reducing computational costs. In this study, we made full use of ChatGPT's powerful natural language understanding and generation capabilities, and further fine-tuned it so that it can better adapt to SATD multi-classification tasks and generate accurate and contextual answers. We are the first to propose using Fine-Tuning GPT for SATD multi-class detection tasks.

3. Multiclass Classification for Self-Admitted Technical Debt based on Fine-Tuning GPT

In this study, we chose to use the "gpt-3.5-turbo-1106" version provided by OpenAI. Recent studies and reports [17] indicate that GPT-4 suffers from certain inconsistencies and inaccuracies, which may stem from a complete redesign of its architecture. In addition, using GPT-4 will significantly increase the experimental cost. In contrast, the GPT-3.5 series has advantages in reliability and cost and is therefore widely used.

To explore the performance of Fine-tuning GPT, tests were conducted in two scenarios: MTO and OTO [12]. In the MTO scenario, since the training set covers

nine projects and there is only a slight class imbalance problem, we do not perform data balancing. This is because in this case, the performance of Fine-Tuning GPT will not be significantly affected by data distribution issues [9]. In the OTO scenario, some projects have serious class imbalance problems. For example, the number of samples for defect debt, design debt and requirement debt in the JFreeChart project are 9, 184 and 15, respectively. To address this issue, we performed random repeated oversampling of all projects to improve the data distribution. gpt-3.5-turbo-1106 was then fine-tuned using the preprocessed dataset. During the Fine-Tuning process, the model is trained on the divided dataset and its internal parameters are adjusted to improve its adaptability to the task. To ensure the certainty and consistency of generated results, we set the "temperature" parameter to 0, and other parameters are maintained at their default values [10]. In addition to creating a final fine-tuned model at the end of each Fine-Tuning job, OpenAI also creates a complete model checkpoint at the end of each training epoch. These checkpoints themselves are complete models and may represent a version of the fine-tuned model before it overfits. Finally, we choose the fine-tuned model for evaluation, using the evaluation metrics including MacroF, MacroP and MacroR. Experimental results show that the fine-tuned gpt-3.5-turbo-1106 shows good classification performance in different scenarios, providing reliable support for SATD classification tasks, see Fig. 1.

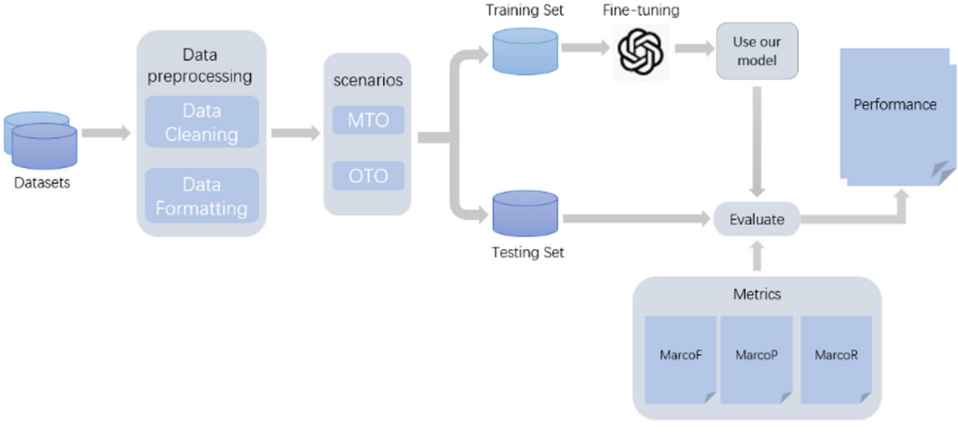

Fig. 1. The process of Fine-Tuning GPT.

4. Experimental Setup

In order to verify the effectiveness of the proposed method, we designed the following two research questions:

- RQ1: Does Fine-Tuning GPT outperform the baseline model for the SATD multi-classification task in the MTO scenario?

- RQ2: Does Fine-Tuning GPT outperform the baseline model for the SATD multi-classification task in the OTO scenario?

4.1. *Dataset*

The experiment used a public dataset provided by Maldonado *et al.*, [1] which contains 62,566 code comments from 10 open-source Java projects. The projects include Apache Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel. These projects cover multiple fields, and the proportion of SATD comments in each project is different. The specific information is shown in Table 1. The dataset annotates five types of technical debt: Design debt, defect debt, requirement debt, test debt and documentation debt. 6.54% of the comments are annotated as SATD, among which the proportions of test debt and documentation debt are relatively low, at 2.13% and 1.4%, respectively.

As can be seen from Table 1, there are differences in the distribution of debt types between different projects. For example, the Hibernate project has no test debt, and the Ant, EMF, JEdit and JFreeChart projects lack document debt. Therefore, consistent with the study of Chen *et al.*, [15] this study only focuses on three types of technical debt: Defect debt, design debt and requirement debt. In order to exclude the influence of minority classes and non-SATD annotations, we removed all non-SATD annotations, test debt annotations and document debt annotations, and only retained the relevant annotations of defect debt, design debt and requirement debt.

4.2. *Prediction scenarios*

Based on the existing work, [1, 6, 7] we designed the following two prediction scenarios for experiments based on the above dataset:

- Many-to-one (MTO) prediction. In this scenario, for each method, we select one project as the test set, and the other nine projects are combined as the training set. Ten models are trained and tested 10 times, respectively.
- One-to-one (OTO) prediction. In this scenario, for each method, we select one project as the test set, and the other nine projects are used as training sets in turn. Ten models are trained and tested 90 times, respectively.

4.3. *Baseline model*

Five state-of-the-art methods were selected to construct the benchmark model.

Baseline 1 (BERT-based method): Md. Faizul Ibne Amin *et al.* [18] proposed a BERT-based method to automatically identify and classify source code errors and validated its effectiveness through experiments. The results show that this method

Table 1. Details of the 10 projects used in experiments.

Project	#Version	#SLOC	#Contributors	#Comments	#Defect	#Design	#Requirement	#Test	#Documentation
Ant	1.70	115,881	74	4137	13	95	13	10	0
JMeter	2.10	81,307	33	8162	22	316	21	12	3
ArgoUML	0.34	176,830	87	9548	127	801	411	44	30
Columba	1.4	100,200	9	6478	13	126	43	6	16
EMF	2.41	228,191	30	4401	8	78	16	2	0
Hibernate	3.3.2	173,467	8	2968	52	355	64	0	1
JEdit	4.2	88,583	57	10,322	43	196	14	3	0
JFreeChart	1.0.19	132,296	19	4423	9	184	15	1	0
JRuby	1.40	150,060	328	4879	161	343	110	6	2
SQuirrel	3.3.0	215,234	46	7230	24	209	50	1	2
Total	—	1,462,058	909	62,566	472	2703	757	85	54

outperforms traditional approaches in terms of accuracy and efficiency, providing new insights for software error detection and correction.

Baseline 2 (XGBoost-based method): Chen *et al.* [15] proposed a multi-classification method based on XGBoost to identify three types of SATD: Design debt, defect debt and implementation debt. They used XGBoost to build a classifier to classify code comments. Experimental results show that this method significantly outperforms NLP-based baseline methods.

Baseline 3 (CNN-based method): Ren *et al.* [7] utilized CNN to build a classifier for detecting SATD. They use CNN to learn informative features in annotated text. To understand these learned text features, they adopted a backtracking approach to highlight salient key phrases. During their evaluation, they also revealed a number of less obvious and less common annotation patterns for identifying SATD.

Baseline 4 (NBM-based method): Huang *et al.* [6] proposed an automated method for identifying SATD comments. They first preprocessed the code comments and used information gain for feature selection. Then, they used NBM [19] to build a sub-classifier for each project and integrated these sub-classifiers into a composite classifier. Finally, they used a voting mechanism to predict the annotation labels from new target projects. Experimental results show that the NBM-based method can effectively improve the recognition effect compared with the baseline method.

Baseline 5 (NLP-based method): Maldonado *et al.* [1] developed an effective method to automatically detect the most common types of SATD, namely design debt and requirement debt. They built a maximum entropy classifier to predict the type of technical debt. Experimental results show that this NLP-based approach can effectively identify these two types of SATD.

4.4. *Data preprocessing*

4.4.1. *Data filtering*

When developers write code comments, they usually express themselves freely according to their personal style and habits, and have their own unique preferences for terms and phrases, so the presentation of code comments often varies, and the quality also varies. However, traditional models can usually only process standardized and structured text, so we need to preprocess the original code comments to remove noisy data.

4.4.2. *Deleting modification records*

Due to the continuous updating of software, developers may record revision dates in code comments. In fact, these comments have nothing to do with the category of SATD. Instead, they increase the number of features and are of no help in identifying SATD types. Therefore, we deleted the modification records presented in the form of

"xx-xx-xx: text", where "xx-xx-xx" represents the date and "text" represents the specific modified content.

4.4.3. *Standardization*

In the process of identifying SATD, punctuation marks usually do not carry valuable semantic information, and the case of the same word will also affect the computational efficiency. Therefore, we uniformly remove punctuation marks and convert all texts into lowercase to avoid feature splitting caused by case differences.

4.4.4. *Class imbalance processing*

For XGBoost, NBM and NLP models, we use the SMOTE oversampling method to deal with the class imbalance problem, thereby ensuring that the minority classes can be effectively represented. For the CNN model, the compute_class_weight method is used to calculate the weight of each class. During the model training process, these weights are used to adjust the model's attention to samples of different categories, thereby effectively alleviating the impact of class imbalance.

4.5. *Evaluation metrics*

In this study, our goal is to identify different types of technical debt (defect debt, design debt and requirement debt), which can be viewed as a multi-classification problem. MarcoP, MarcoR and MarcoF are widely used indicators to evaluate the effectiveness of multi-classification techniques [20, 21]. Therefore, we can use MarcoP, MarcoR and MarcoF to verify the effectiveness of our method. The specific formula is as follows:

Let C be the number of classes, TP_i represents the number of G_i code comments belonging to the class that are correctly predicted as the class G_i, and FP_i represents the number of G_i code comments that do not belong to the class that are incorrectly predicted as the class G_i. TN_i represents the number G_i of code comments that do not belong to the class and are correctly predicted as not belonging to the class G_i and represents the number of FN_i code comments G_i that belong to the class and are incorrectly predicted as not belonging to the class G_i.

MacroP is used to evaluate the accuracy between the predicted results and the actual results of all categories. It calculates the precision of all categories and then averages these precisions to get an overall evaluation index. The calculation formula is as follows:

$$\mathrm{MarcoP} = \frac{1}{C}\sum_{i=1}^{C}\frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FP}_i}. \tag{1}$$

MacroR is used to evaluate the consistency between the model's prediction results and actual results for all categories. It measures the model's overall coverage of all

categories of actual samples by calculating the recall rate of each category and then averaging the recall rates of all categories. The calculation formula is as follows:

$$\text{MarcoR} = \frac{1}{C}\sum_{i=1}^{C}\frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}. \tag{2}$$

MacroF is an indicator used to weigh MacroP and MacroR, and is used to evaluate the comprehensive performance of the model in multi-classification tasks. Its calculation formula is as follows:

$$\text{MarcoF} = \frac{\text{MarcoP} + \text{MarcoR}}{2 * \text{MarcoP} * \text{MarcoR}}. \tag{3}$$

4.6. *Parameter settings*

In order to compare the performance with Fine-Tuning GPT, the key parameters of each baseline model are set.

In the XGBoost model, we use grid search [22] to obtain the optimal hyperparameters of the model. CountVectorizer is used to extract text features and the bag-of-words model is used to represent the text. The training process is optimized by setting the maximum tree depth (max_depth = 6), learning rate (learning_rate = 0.06), subsampling ratio (subsample = 0.8) and early stopping mechanism (early_stopping_rounds = 30) to effectively prevent overfitting. In the CNN model, we also use grid search [22] to determine the appropriate hyperparameter values for the model. The maximum vocabulary size is set to 10,000 and the sequence length is fixed to 100. Each word is mapped to a 300-dimensional embedding space through an embedding layer to capture semantic information. The model architecture includes a convolutional layer (with 128 filters, a convolution kernel size of 3, a ReLU activation function and L2 regularization), a global maximum pooling layer, and a fully connected layer. The output layer uses a Softmax activation function for multi-classification tasks. The optimizer is Adam, the learning rate is set to 0.0001, the batch size is 16 and the number of training rounds is 5. In the NLP model, L2 regularization is used to control the model complexity, the maximum number of iterations is set to 1000 and the optimization algorithm is selected as L-BFGS. The TF-IDF method is used in the feature extraction stage to capture the important information about words in the text. At the same time, all feature words are extracted by default, and the n-gram feature expansion is supported. In the NBM model, CountVectorizer is used for feature extraction, and the Mutual Information method is combined for feature selection. The 50 features with the highest information gain are retained to improve the efficiency of the model. Each sub-classifier is trained using only the data of a single project, and then the naive Bayes classifier is used to build the sub-classifier. In the test phase, all sub-classifiers predict the data of the target project, and the final classification result is determined by majority voting. In the Fine-tuning BERT model, we have set the batch size to 8, the learning rate to 2e−5, the number of training epochs to 3 and the maximum sequence length to 128. In the OTO

scenario, for the NBM model, due to limited training data, we only retain the top 20 features with the highest information gain and build the Naive Bayes classifier using the default parameters. The parameters for other baseline models remain the same as mentioned above.

5. Experimental Results and Analysis

5.1. *Analysis for RQ1*

To analyze RQ1, we employed the Scott–Knott test to classify all methods into different ranks. Specifically, the test divides methods into ranks through hierarchical clustering analysis, initially grouping methods based on the rankings of MarcoF, MarcoP, or MarcoR. If two ranks are statistically significantly different, the methods within each rank are recursively divided further. The grouping process terminates when no statistically significant differences can be found between groups. The results, as shown in Figs. 2–4, use different colors to represent the groups identified by the Scott–Knott test.

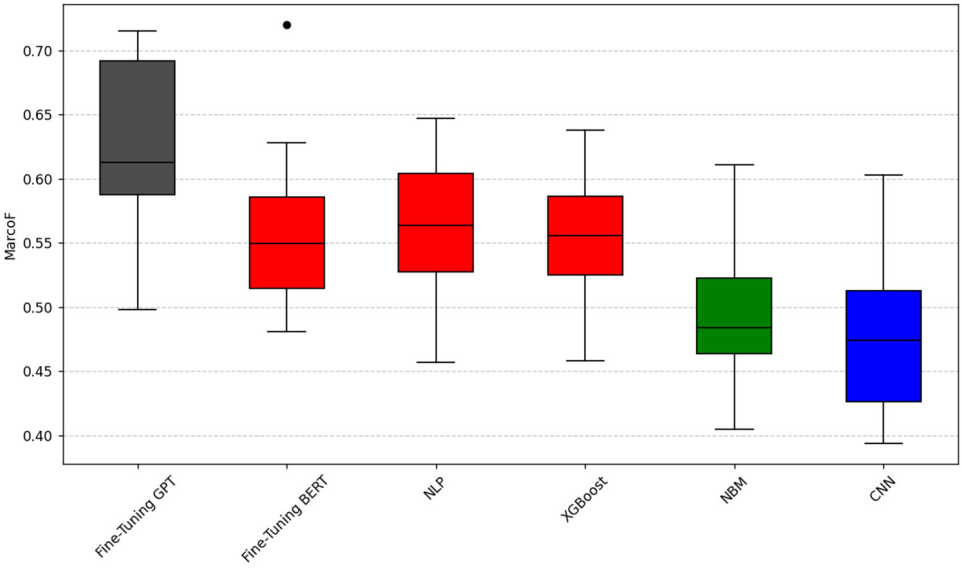

Fig. 2. The result of Scott–Knott test in MacroF in the MTO scenario.

From Figs. 2–4, it is evident that Fine-Tuning GPT consistently outperforms all baseline models, maintaining a significant performance advantage. On average, Fine-Tuning GPT ranks first across all models, demonstrating its superior performance. For example, in the case of MarcoF, Fine-Tuning GPT exhibits a notable improvement, belonging to the black group and surpassing all baseline models.

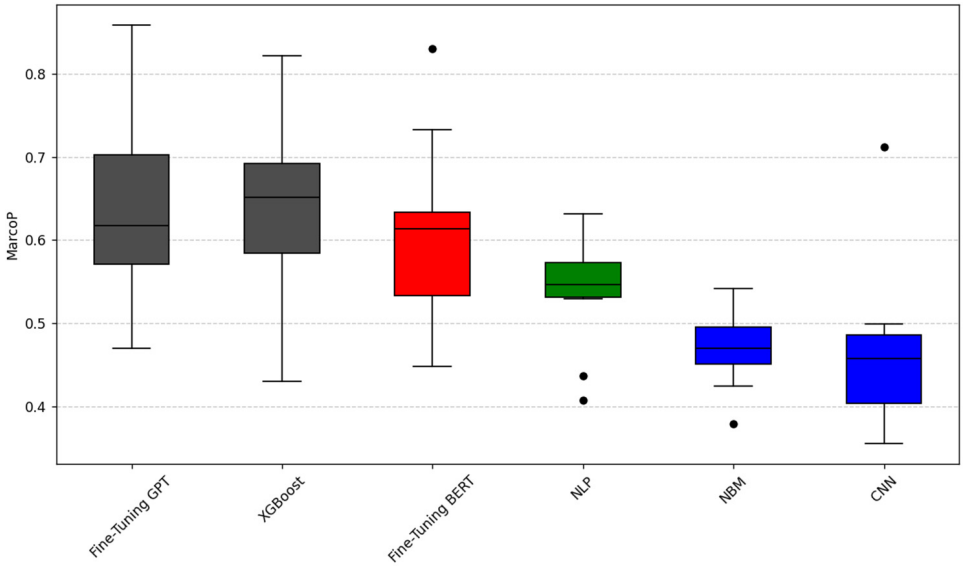

Fig. 3. The result of Scott–Knott test in MacroP in the MTO scenario.

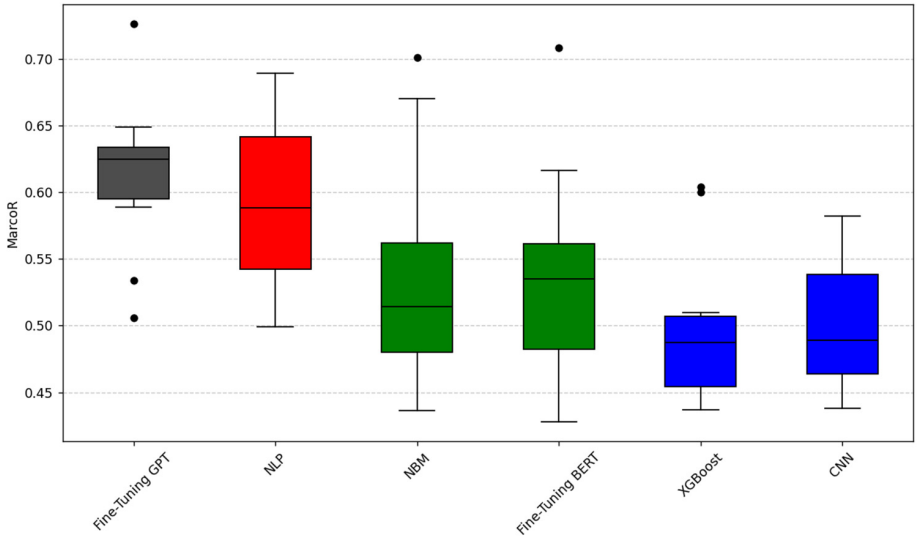

Fig. 4. The result of Scott–Knott test in MacroR in the MTO scenario.

Fine-Tuning BERT, NLP and XGBoost are categorized into the red group, securing the second position in terms of performance. In contrast, NBM and CNN show relatively lower performance, being assigned to the green and blue groups, respectively, further emphasizing the performance gap. The results for MarcoR reinforce this trend, with Fine-Tuning GPT again leading in the highest group and

significantly outperforming all other models. NLP's performance is distributed in the red group, while NBM and Fine-Tuning BERT are placed in the green group, showing a slight performance decline. Meanwhile, XGBoost and CNN perform poorly, falling into the blue group. In MarcoP, Fine-Tuning GPT slightly outperforms XGBoost, with both models belonging to the black group. Fine-Tuning BERT is assigned to the red group, NLP is categorized in the green group and NBM and CNN, demonstrating the weakest performance, are also placed in the green group.

After completing the significance grouping analysis, we conducted a detailed performance comparison for each project. Using the MarcoF, MarcoP and MarcoR metrics, we calculated the performance of Fine-Tuning GPT and the baseline models for each project, and counted the number of wins, draws and losses to obtain the W/D/L analysis results. The "Average" row displays the average values across all projects. We found that Fine-Tuning GPT significantly outperforms all baseline models in the three metrics: Average MarcoF, average MarcoP and average MarcoR. The results of these selected methods are presented in Tables 2–4.

Table 2. Fine-Tuning GPT versus five baseline methods in MTO scenario using MarcoF.

Project	Fine-Tuning GPT	Fine-Tuning BERT	XGBoost	CNN	NLP	NBM
Ant	**0.703**	0.628	0.577	0.537	0.614	0.463
JMeter	**0.586**	0.482	0.469	0.421	0.457	0.405
ArgoUML	**0.554**	0.534	0.521	0.419	0.521	0.465
Columba	0.593	0.529	0.458	**0.603**	0.575	0.519
EMF	0.621	0.577	0.630	0.517	**0.647**	0.573
Hibernate	**0.658**	0.589	0.538	0.443	0.559	0.488
JEdit	**0.715**	0.565	0.537	0.470	0.568	0.524
JFreeChart	0.498	0.481	**0.590**	0.394	0.484	0.457
JRuby	**0.605**	0.510	0.574	0.478	0.547	0.480
SQuirrel	0.715	**0.720**	0.638	0.500	0.630	0.611
Average	**0.625**	0.561	0.553	0.478	0.560	0.499
W/D/L	—	9/0/1	8/0/2	9/0/1	9/0/1	10/0/0

Table 3. Fine-Tuning GPT versus five baseline methods in MTO scenario using MarcoP.

Project	Fine-Tuning GPT	Fine-Tuning BERT	XGBoost	CNN	NLP	NBM
Ant	0.768	0.640	**0.811**	0.499	0.586	0.451
JMeter	**0.554**	0.456	0.465	0.391	0.408	0.379
ArgoUML	0.611	0.614	**0.625**	0.402	0.545	0.477
Columba	0.558	0.509	0.430	**0.712**	0.530	0.478
EMF	0.629	0.608	**0.821**	0.492	0.632	0.501
Hibernate	**0.694**	0.614	0.605	0.408	0.534	0.453
JEdit	**0.858**	0.830	0.696	0.454	0.548	0.508
JFreeChart	0.479	0.448	**0.577**	0.356	0.437	0.425
JRuby	0.643	0.613	**0.678**	0.467	0.550	0.463
SQuirrel	0.705	**0.733**	0.682	0.462	0.581	0.542
Average	**0.650**	0.607	0.639	0.464	0.535	0.468
W/D/L	—	8/0/2	5/0/5	9/0/1	9/0/1	10/0/0

Table 4. Fine-Tuning GPT versus five baseline methods in MTO scenario using MarcoR.

Project	Fine-Tuning GPT	Fine-Tuning BERT	XGBoost	CNN	NLP	NBM
Ant	**0.649**	0.616	0.448	0.582	0.646	0.475
JMeter	**0.623**	0.511	0.473	0.457	0.520	0.436
ArgoUML	**0.506**	0.473	0.446	0.438	0.499	0.454
Columba	**0.632**	0.551	0.491	0.522	0.628	0.569
EMF	0.634	0.549	0.510	0.546	0.662	**0.670**
Hibernate	**0.626**	0.565	0.484	0.483	0.586	0.529
JEdit	**0.613**	0.428	0.437	0.489	0.590	0.541
JFreeChart	0.534	0.521	**0.604**	0.441	0.541	0.495
JRuby	**0.589**	0.437	0.498	0.489	0.545	0.499
SQuirrel	**0.726**	0.708	0.600	0.544	0.689	0.701
Average	**0.613**	0.536	0.499	0.498	0.591	0.537
W/D/L	—	10/0/0	9/0/1	10/0/0	8/0/2	9/0/1

Under the MarcoF metric, Fine-Tuning GPT's average value is 0.625, which is higher than Fine-Tuning BERT's 561, XGBoost's 0.553, CNN's 0.478, NLP's 0.560 and NBM's 0.499. Compared to these baseline models, Fine-Tuning GPT's performance improved by 11.40%, 13.02%, 30.75%, 11.61% and 25.25%, respectively. In the MarcoP metric, Fine-Tuning GPT's average value is 0.650, which exceeds Fine-Tuning BERT's 607, XGBoost's 0.639, CNN's 0.464, NLP's 0.535 and NBM's 0.468. Fine-Tuning GPT's performance improved by 7.08%, 1.72%, 40.09%, 21.50% and 38.89%, respectively, compared to these baseline models. For the MarcoR metric, Fine-Tuning GPT's average value is 0.613, which is higher than Fine-Tuning BERT's 0.536, XGBoost's 0.499, CNN's 0.498, NLP's 0.591 and NBM's 0.537. Fine-Tuning GPT's performance improved by 14.37%, 22.85%, 23.09%, 3.72% and 14.15%, respectively, compared to these baseline models.

In the W/D/L analysis, for the MarcoF metric, Fine-Tuning GPT's W/D/L results against Fine-Tuning BERT, XGBoost, CNN, NLP, and NBM are 9/0/1, 8/0/2, 9/0/1, 9/0/1 and 10/0/0, demonstrating outstanding performance. For the MarcoP metric, Fine-Tuning GPT's W/D/L results against Fine-Tuning BERT, XGBoost, CNN, NLP and NBM are 8/0/2, 5/0/5, 9/0/1, 9/0/1 and 10/0/0. The W/D/L results for Fine-Tuning GPT and XGBoost are tied, but Fine-Tuning GPT has a higher average MarcoP. For the MarcoR metric, Fine-Tuning GPT's W/D/L results against Fine-Tuning BERT, XGBoost, CNN, NLP and NBM are 10/0/0, 9/0/1, 10/0/0, 8/0/2 and 9/0/1, respectively. Overall, Fine-Tuning GPT demonstrated exceptional performance in the MTO scenario.

5.2. *Analysis for RQ2*

To analyze RQ2, we used the Scott–Knott test to classify all methods into different levels. From Figs. 5–7, it can be seen that after performing significance-based

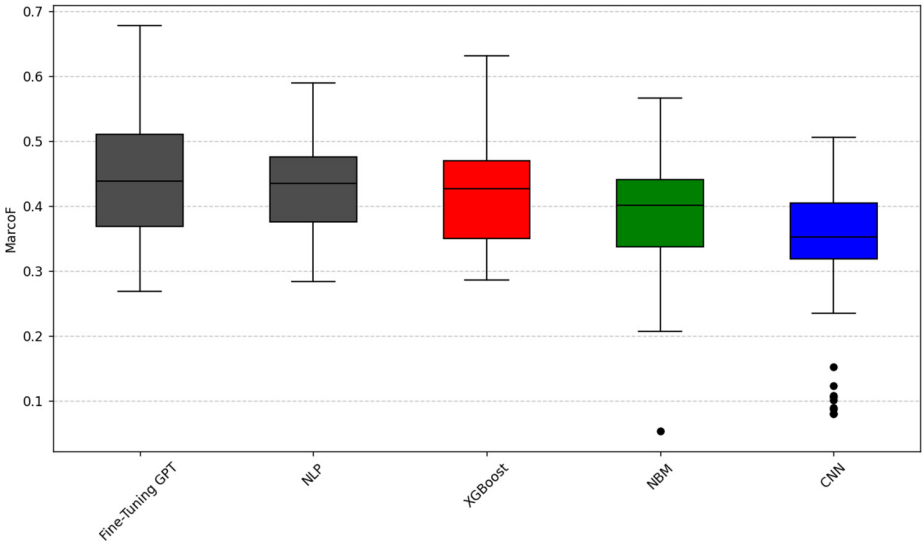

Fig. 5. The result of Scott–Knott test on MacroF in the OTO scenario.

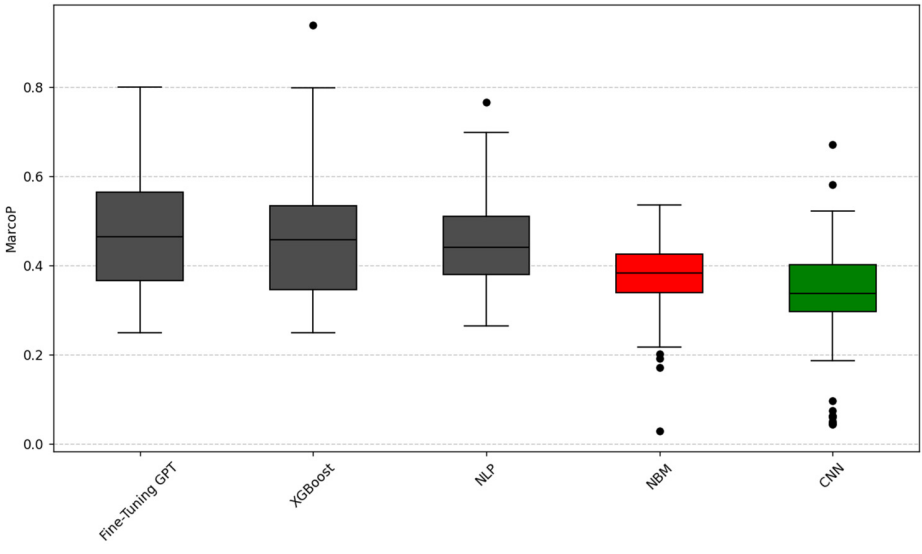

Fig. 6. The result of Scott–Knott test on MacroP in the OTO scenario.

grouping on these models' performance through the Scott–Knott test, the results are obtained in the OTO scenario.

Fine-Tuning GPT consistently demonstrates a significant advantage over all baseline models, with its average performance ranking first across all models, reflecting its outstanding performance. Taking MarcoF as an example, Fine-Tuning

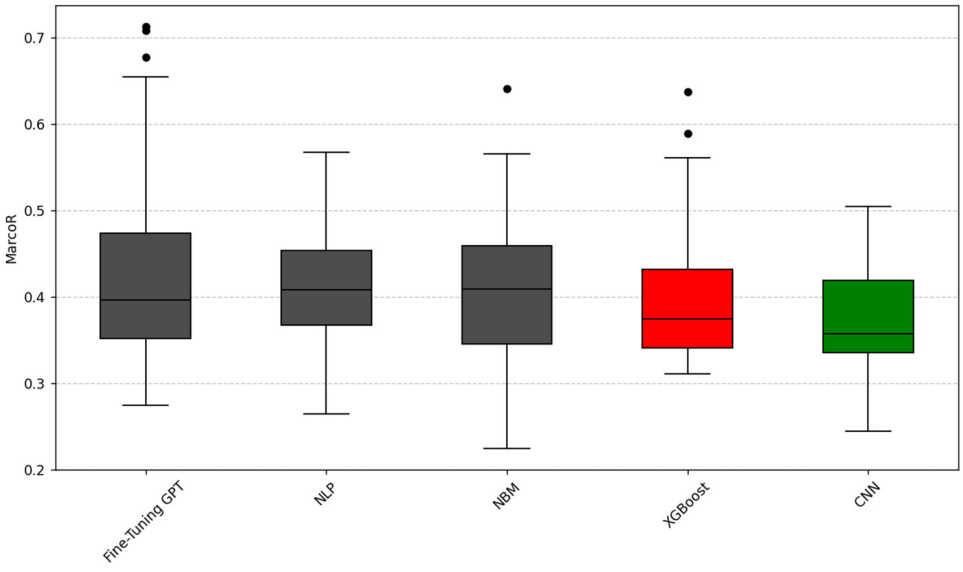

Fig. 7. The result of Scott–Knott test on MacroR in the OTO scenario.

GPT shows a significant performance, being assigned to the black group, outperforming all baseline models and showcasing its superiority. NLP ranks second; XGBoost has relatively lower performance and is assigned to the red group, while NBM and CNN perform poorly and are assigned to the red and green groups, further emphasizing the performance gap.

In MarcoP, Fine-Tuning GPT is again assigned to the top group. XGBoost and NLP slightly lag behind Fine-Tuning GPT, and both are placed in the black group. The performance of NBM is assigned to the red group, and CNN has the worst performance, being placed in the green group. In MarcoR, Fine-Tuning GPT performs better than NLP and NBM, and both are assigned to the black group. XGBoost is placed in the red group, and CNN is in the green group, showing the poorest performance.

After performing a significant group analysis, a detailed performance comparison between Fine-Tuning GPT and four baseline models was conducted. The results are presented in Table 5 (with detailed results in the github[a]). Using the MarcoF, MarcoP and MarcoR metrics, we analyzed the test results for each target item. The "Average" row shows the mean values across all items, while the "Median" row indicates the median values across all items. It was found that Fine-Tuning GPT outperforms all baseline models in terms of average MarcoP, average MarcoR and average MarcoF. For the MarcoF metric, Fine-Tuning GPT achieved an average value of 0.439, surpassing XGBoost's 0.418, CNN's 0.342, NLP's 0.429 and NBM's 0.390. Compared to these baseline models, Fine-Tuning GPT's performance

[a] https://github.com/ACV-sotc/Multiclass-Classification-for-SATD.git.

Table 5. Performance comparison of Fine-Tuning GPT and four baseline methods in the OTO scenario based on MacroF, MacroP and MacroR.

Indicators		Fine-Tuning GPT	XGBoost	CNN	NLP	NBM
MarcoF	Average	**0.439**	0.418	0.342	0.429	0.390
	Median	**0.439**	0.426	0.353	0.434	0.401
MarcoP	Average	**0.476**	0.459	0.333	0.453	0.378
	Median	**0.466**	0.459	0.338	0.442	0.385
MarcoR	Average	**0.422**	0.394	0.376	0.413	0.409
	Median	0.396	0.375	0.357	0.408	**0.409**

improved by 5.02%, 28.36%, 2.33% and 12.56%, respectively. For the MarcoP metric, Fine-Tuning GPT's average value was 0.476, exceeding XGBoost's 0.459, CNN's 0.333, NLP's 0.453 and NBM's 0.378. Compared to these baseline models, Fine-Tuning GPT's performance improved by 3.70%, 42.9%, 5.08% and 25.93%, respectively. For the MarcoR metric, Fine-Tuning GPT's average value was 0.422, higher than XGBoost's 0.394, CNN's 0.376, NLP's 0.413 and NBM's 0.409. Compared to these baseline models, Fine-Tuning GPT's performance improved by 7.11%, 12.23%, 2.18% and 3.18%, respectively.

In terms of the median values, Fine-Tuning GPT demonstrated its strong robustness and superiority. For MarcoF, Fine-Tuning GPT's median was 0.439, higher than XGBoost's 0.418, CNN's 0.342, NLP's 0.429 and NBM's 0.390. For MarcoP, Fine-Tuning GPT's median was 0.466, exceeding XGBoost's 0.459, CNN's 0.338, NLP's 0.442 and NBM's 0.385. For MarcoR, Fine-Tuning GPT's median was 0.396, slightly lower than NLP's 0.408 and NBM's 0.409, but significantly higher than XGBoost's 0.375 and CNN's 0.357. These results indicate that Fine-Tuning GPT exhibits strong multi-classification performance in the OTO scenario.

Through the analysis of RQ1 and RQ2, it is evident that Fine-Tuning GPT not only surpasses other models in terms of precision and robustness but also establishes its strong generalization ability across different projects and scenarios. This is demonstrated by its consistent performance improvements over the baseline models in both the MTO and OTO scenarios, across various performance metrics such as MacroF, MacroP and MacroR. The results indicate that Fine-Tuning GPT is capable of effectively capturing and leveraging the nuanced information present in code comments to accurately classify different types of SATD.

While we acknowledge that Fine-Tuning GPT may not be perfect and may still have limitations in certain scenarios, its overall performance and generalization ability make it a promising approach for SATD multi-classification. We believe that further research and refinement can further enhance its capabilities and make it an even more valuable tool for managing technical debt in software development.

6. Discussion

6.1. *Discussion on predicting different types of technical debt*

The results from Sec. 5 indicate that the Fine-tuning GPT model outperforms the baseline model in most projects, although performance degradation is still observed in some projects. For further analysis, Table 6 presents the F1-score comparison results of all models across 10 projects for different types of technical debt. Here, "Require" represents "Requirement" and the 10 projects are identified by the first three letters of their names, with AVE denoting the average. From the data in the table, it can be seen that the average F1-score of Fine-tuning GPT for various types of technical debt prediction remains significantly higher than that of the baseline model. Additionally, the F1-scores of all models for detecting design debt are notably higher than those for defect debt and requirement debt. This phenomenon may be related to the inherent characteristics of technical debt: Design debt typically exhibits more distinct structural features and patterns, while defect debt and requirement debt rely more heavily on contextual semantic understanding.

Table 6. Results of all methods on 10 projects in terms of F1-measure for each type of technical debt.

Classification		Ant	JMe	Arg	Col	EMF	Hib	JEd	JFr	JRu	SQu	AVE
Fine-tuning GPT	Defect	0.500	0.423	0.383	0.500	0.438	0.608	0.542	0.400	0.432	0.593	**0.482**
	Design	0.914	0.919	0.770	0.609	0.808	0.889	0.908	0.786	0.852	0.892	**0.835**
	Require	0.667	0.408	0.438	0.503	0.190	0.451	0.500	0.169	0.446	0.653	**0.443**
Fine-tuning BERT	Defect	0.480	0.333	0.396	0.414	0.364	0.418	0.561	0.400	0.277	0.744	0.439
	Design	0.899	0.859	0.805	0.772	0.889	0.849	0.916	0.671	0.780	0.827	0.827
	Require	0.286	0.190	0.385	0.537	0.500	0.323	0.250	0.262	0.469	0.563	0.377
XGBoost	Defect	0.267	0.263	0.193	0.190	0.444	0.246	0.273	0.400	0.312	0.550	0.314
	Design	0.894	0.875	0.798	0.670	0.867	0.846	0.840	0.829	0.808	0.810	0.824
	Require	0.286	0.207	0.388	0.620	0.444	0.396	0.250	0.455	0.489	0.507	0.404
CNN	Defect	0.230	0.207	0.210	0.200	0.200	0.193	0.293	0.129	0.328	0.318	0.231
	Design	0.464	0.245	0.392	0.510	0.154	0.499	0.426	0.106	0.327	0.724	0.385
	Require	0.500	0.116	0.428	0.400	0.457	0.337	0.133	0.267	0.419	0.457	0.351
NLP	Defect	0.563	0.262	0.311	0.303	0.429	0.375	0.381	0.261	0.331	0.492	0.371
	Design	0.886	0.778	0.793	0.722	0.825	0.805	0.872	0.713	0.733	0.785	0.791
	Require	0.500	0.207	0.430	0.564	0.300	0.485	0.444	0.320	0.480	0.506	0.424
NBM	Defect	0.219	0.192	0.280	0.286	0.300	0.307	0.333	0.170	0.325	0.447	0.286
	Design	0.541	0.478	0.493	0.471	0.524	0.437	0.648	0.396	0.482	0.608	0.508
	Require	0.364	0.134	0.421	0.471	0.500	0.345	0.417	0.276	0.439	0.532	0.390

To delve deeper into the reasons for misclassification, we conducted a qualitative analysis of specific misclassified samples. Table 7 displays the statistical results of all misclassification cases, with the highest number of misclassifications occurring between design debt and requirement debt. Through an in-depth interpretation of code comments, we found that the model struggles to accurately distinguish between functional design requirements and future functionalities to be implemented.

Table 7. Analysis of misclassified samples

Sample	True label	Predicted label	Cause
*/** TODO: Spring-inject when this class is a Spring bean */*	Design	Requirement	This describes dependency injection as a design approach, not a functional requirement, but the model may misclassify it as a requirement
// BAD — Converting a URL to a file this way is problematic when the file path contains spaces	Design	Defect	This comment describes a design issue, not a bug, but the model may misclassify it as a defect
// TODO: Setup title correctly	Defect	Requirement	This TODO highlights a bug, but its simple wording may lead the model to mistake it for a requirement change
// If we are connecting to a database, then this is fine. However, when // using MockObjects, this is problematic . . .	Defect	Design	This discusses database connection issues and the impact of the unimplemented MockObjects method, which is a defect. However, the model may misinterpret it as an architectural design issue
// TODO: if the user checks "export entire table" and doesn't select all, // then the selected indices are not set, and this check doesn't properly // detect missing data.	Requirement	Defect	This describes the requirement for a table export feature, which may need optimized data detection logic, falling under requirement debt. However, the model might misinterpret it as a bug in the code implementation rather than a functional adjustment
//?? remember which column is the rowID (if any) so we can //?? prevent editing on it	Requirement	Design	This comment describes the management requirement for column IDs, which falls under requirement debt. However, the model might interpret it as detailing design implementation specifics

Specifically, design debt primarily focuses on code structure and architectural issues, whereas requirement debt emphasizes functional requirements. However, in code comments, the boundaries between the two are often blurred, particularly in "TODO" statements, which may simultaneously involve design improvements and new functionalities, leading to difficulties in accurate classification by the model.

Furthermore, significant misclassification is also observed between defect debt and requirement debt. Some code bugs are incorrectly categorized as requirement issues, while certain descriptions of unimplemented requirements are misclassified as defects. This confusion may stem from the similarity in semantic expressions between the two. As for misclassifications between defect debt and design debt, the model erroneously identifies some design decisions related to code implementation as defects, while misclassifying potential code bugs as design issues. These misclassification phenomena highlight the complexity of technical debt detection tasks and the limitations of the model in handling semantic ambiguity and contextual dependencies.

Through a systematic analysis of misclassified samples, it was observed that design-related comments frequently contain terms associated with structure, patterns,

or methods (e.g. "implement" and "constructors"). This indicates that the current feature extraction mechanisms require further optimization to enhance the model's understanding of design semantics. Notably, many misclassification cases arise from the model's overgeneralization of the "TODO" keyword, which is erroneously classified as requirement-related technical debt, while phrases such as "Check" and "not serialized yet" are more likely to indicate defect-related technical debt. Therefore, refining the feature extraction strategy is essential, particularly by increasing the weight of keywords such as "FIXME," "bug" and "error," in order to improve the model's ability to identify defect-related descriptions.

The causes of misclassification can be attributed to two primary factors: First, the lack of sufficient contextual information hinders the model's ability to accurately interpret the semantics of comments. Second, certain keywords (e.g. "fix," "check" and "need to") possess semantic ambiguity, leading to multiple interpretations across different types of technical debt. Specifically, Defect, Design and Requirement are not entirely distinct but rather exhibit a degree of semantic overlap. This overlap increases the complexity of the classification task, necessitating stronger semantic understanding and contextual reasoning capabilities from the model. To address these challenges, it is recommended to incorporate additional contextual information (e.g. code snippets or project documentation) to help the model gain a more comprehensive understanding of the comments' context. Additionally, the model's ability to differentiate between design, defect and requirement-related language patterns should be optimized, enabling more accurate identification of various types of technical debt. Furthermore, the model should be improved to handle edge cases, such as distinguishing between bug fixes and feature improvements.

6.2. *Practical implications and deployment of Fine-Tuning GPT*

6.2.1. *Performance improvement and resource requirements*

Fine-tuning GPT models can be optimized for specific tasks (such as SATD multi-class classification), improving performance for task-specific needs. For example, Fine-Tuning helps the model better understand language related to technical debt, which general-purpose models may overlook. While Fine-Tuning enhances performance, it also demands significant computational resources, especially during training and inference. When deploying, hardware configuration, storage requirements and memory needs must be considered to support large-scale data processing.

6.2.2. *Integration and scalability challenges*

Fine-tuned models often need to be integrated with existing systems and workflows, which may require custom APIs or optimized server architectures. In large-scale deployments, inference performance may face bottlenecks, requiring techniques such as model distillation or accelerated inference to improve efficiency. To ensure the

system can scale to handle more data or users, deployment solutions must thoroughly address scalability concerns.

6.2.3. *Ongoing maintenance and bias issues*

Fine-tuning GPT models is not static. Over time, they may require periodic updates or retraining to address new data and evolving requirements. At the same time, Fine-Tuning may introduce biases that affect the fairness of the model, especially when the training data is unbalanced. Therefore, organizations need to carefully select training data and implement effective monitoring mechanisms to regularly evaluate and adjust the model's output, ensuring its applicability and fairness across different contexts.

6.3. *Trade-off between efficiency and effectiveness*

To evaluate the computational cost of Fine-Tuning and training different models, we conducted experiments on multiple projects. The detailed results are presented in Table 8.

Table 8. Comparison of Fine-Tuning and training time for different projects across various models.

	Fine-Tuning time (minute)		Training time (minute)			
Project	GPT	BERT	XGBoost	CNN	NLP	NBM
Ant	41	61	<1	<1	<1	<1
JMeter	45	40	<1	<1	<1	<1
ArgoUML	39	35	<1	<1	<1	<1
Columba	40	42	<1	<1	<1	<1
EMF	48	43	<1	<1	<1	<1
Hibernate	51	38	<1	<1	<1	<1
JEdit	39	40	<1	<1	<1	<1
JFreeChart	46	42	<1	<1	<1	<1
JRuby	49	37	<1	<1	<1	<1
SQuirrel	47	40	<1	<1	<1	<1
Average	45	42	<1	<1	<1	<1

The experimental results show that the average Fine-Tuning time for the GPT model is 45 min, with the most time-consuming task (Hibernate) taking 51 min and the fastest task (ArgoUML) requiring only 39 min. In comparison, the average Fine-Tuning time for the BERT model is 42 min, slightly lower than GPT, but the difference between the two is not significant. For traditional machine learning methods (such as XGBoost, CNN, NLP and NBM), the training time for all tasks is less than 1 min, indicating that the computational cost of traditional methods is much lower than that of large language models.

Although the Fine-Tuning time for GPT is longer compared to traditional methods, the computational overhead remains within an acceptable range given

the significant performance improvement. Therefore, our approach achieves an effective performance enhancement with only a slight increase in computational time, striking a favorable balance between computational cost and model performance.

The experiments were conducted on a machine equipped with a 12th Gen Intel Core i5-1240P processor, an Intel Xe Graphics GPU with 7.9 GB of VRAM and 16 GB of RAM. Python 3.10 was used for the experiments, and the development environment consisted of PyCharm 2024.3, running on Windows 11 operating system.

7. Threats to Validity

7.1. *External validity*

External validity focuses on the generalizability of research findings. Our findings are based on code comments in open source projects, and to reduce external validity threats, we selected open source projects from different fields that have been widely used in the existing research [6, 7]. However, our results may not be generalizable to other types of open source or commercial projects, especially those that use different programming languages, come from different fields, or have different technology stacks. In addition, the small number of reviews, lack of reviews, or reviews written in non-English languages may also limit the generalizability of the results. In our study, the experiments used datasets from multiple open-source projects, and different projects may have different code structures, development styles, comment habits and technology stacks. These differences may cause the model to perform well on some projects but poorly on others. To mitigate these threats, future research should repeat our experiments on a wider variety of projects.

7.2. *Internal validity*

Internal validity focuses on the issue of data leakage. Our research method is based on Fine-Tuning a model pre-trained on a large amount of data. Data leakage means that since ChatGPT is trained on a wide and diverse corpus, it may have been exposed to certain specific content or patterns, and thus be able to recognize these contents when applied to SATD detection [23]. This recognition does not come from the model's learning and generalization capabilities, but because it has been exposed to similar data before, the model may rely on previous exposure information when testing. To mitigate this threat, we ensure that we use datasets that are highly relevant to the experimental objectives and remove redundant information during fine-tuning, and control the independence and integrity of the data through strict experimental design. Although data leakage remains a potential risk, we believe that with proper data processing and experimental design, the results still have strong internal validity.

7.3. *Construct validity*

The experiments used a dataset from 10 open-source projects that was obtained by Maldonado *et al.* [1] by manually annotating technical debt types. However, our approach relies heavily on code comments. It is important to note that developers may choose not to express technical debt in source code comments, and using comments to identify SATD may not be fully representative because comments and code may not be updated consistently. Nevertheless, previous studies have shown that changes in source code are highly consistent with changes in comments [13, 24]. In addition, there may be many unself-admitted technical debts in the analyzed projects. However, given that the focus of this paper is to improve the detection capabilities of the most common types of SATD, considering all types of technical debt is beyond the scope of this paper.

7.4. *Computational validity*

When Fine-Tuning LLMs such as GPT-3 and GPT-4, computational efficiency is crucial for ensuring the effective utilization of resources. The Fine-Tuning process for such models typically requires high-performance GPU or TPU clusters, leading to substantial economic costs and energy consumption. Additionally, large-scale datasets further increase storage and preprocessing overhead. To address these challenges, researchers have proposed various optimization techniques. Zixuan Zhou *et al.* [25] highlighted that model compression methods, including quantization, pruning, knowledge distillation and sparsification and can significantly reduce computational demands. Furthermore, the use of hardware acceleration (e.g. GPUs and TPUs) and distributed inference techniques can enhance efficiency. These approaches help maintain model performance while reducing resource consumption, providing essential support for achieving efficient and sustainable model training.

8. Conclusion

This paper makes full use of the performance of Fine-Tuning GPT to study the SATD multi-classification detection problem in software development for the first time, setting a precedent in the comparative study with the benchmark model. By Fine-Tuning the GPT-3.5-turbo model to adapt to the SATD multi-class detection task, an efficient multi-class detection framework is constructed. The experiment designed two scenarios, MTO and OTO, and conducted a systematic comparison with the benchmark model to evaluate the MarcoF, MarcoP and MarcoR indicators of the model. Experimental results show that in MTO and OTO scenarios, the fine-tuned GPT-3.5-turbo model significantly outperforms the baseline model in multi-classification tasks. In the MTO scenario, the average MarcoF of Fine-Tuning GPT is 0.625, 11.40% higher than the optimal benchmark model BERT, the average MarcoP of Fine-Tuning GPT is 0.650 which is 1.72% higher than the optimal

benchmark model XGBoost, the average MarcoR of Fine-Tuning GPT is 0.613, which is 3.72% higher than the optimal benchmark model NLP; in the OTO scenario, the average MarcoF of Fine-Tuning GPT is 0.439 which is 2.33% higher than the optimal benchmark model NLP; the average MarcoP of Fine-Tuning GPT is 0.476, which is 3.70% higher than the optimal benchmark model XGBoost; the average MarcoR of Fine-Tuning GPT is 0.422 which is 2.18% higher than the optimal benchmark model NLP. These results show that Fine-Tuning GPT not only has higher classification accuracy and recall capabilities in cross-project SATD multi-classification but also exhibits superior integrity and balance.

This study presents for the first time, a generative pre-training model framework for SATD multi-class detection and validates the unique potential of Fine-Tuning GPT in capturing information from complex texts. Through comprehensive experimental comparisons, this paper reveals the advantages of Fine-Tuning GPT in cross-project multi-classification tasks, providing new ideas and references for SATD management research. In order to promote research in the field of SATD, we have made the experimental data and code public at https://github.com/ACV-sotc/Multiclass-Classification-for-SATD.git.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Nos. 62172356 and 61872321), Zhejiang Provincial Natural Science Foundation of China (No. LY22F020026), Ningbo Key R&D Program (Nos. 2023Z229 and 2023Z231), General Research Project funded by the Department of Education of Zhejiang Province (No. Y202353115), NingboTech University Talent Introduction Fund Project (No. 20220518), China Postdoctoral Science Foundation (2024M761927), the Shanghai "Science and Technology Innovation Action Plan" Star Cultivation Program (Sailing Program, 24YF2719900, 24YF2720000) and the Ningbo Natural Science Foundation (No. 2022J162).

ORCID

Yiyang Du https://orcid.org/0009-0000-9666-1994
Xingguang Yang https://orcid.org/0000-0001-7489-2265
Zhenyu Shu https://orcid.org/0000-0001-5733-6638
Zijie Huang https://orcid.org/0000-0002-8911-9889
Gang Wang https://orcid.org/0000-0001-5283-7126
Libo Xu https://orcid.org/0000-0002-7779-5011

References

1. E. da S. Maldonado, E. Shihab and N. Tsantalis, Using natural language processing to automatically detect self-admitted technical debt, *IEEE Trans. Software Eng.* **43**(11) (2017) 1044–1062.

2. W. Cunningham, The wycash portfolio management system, *ACM SIGPLAN OOPS Messenger* **4**(2) (1992) 29–30.
3. Z. Li, P. Liang, P. Avgeriou, N. Guelfi and A. Ampatzoglou, *An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt* (ACM, 2014), pp. 119–128.
4. F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh and M. D. Penta, *Recommending when Design Technical Debt should be Self-Admitted* (IEEE Computer Society, 2017), pp. 216–226.
5. R. Marinescu, *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws* (IEEE Computer Society, 2004), pp. 350–359.
6. Q. Huang, E. Shihab, X. Xia, D. Lo and S. Li, Identifying self-admitted technical debt in open source projects using text mining, *Empir. Softw. Eng.* **23**(1) (2018) 418–451.
7. X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang and J. Grundy, Neural network-based detection of self-admitted technical debt: From performance to explainability, *ACM Trans. Softw. Eng. Methodol.* **28**(3) (2019) 15.
8. E. da S. Maldonado and E. Shihab, *Detecting and Quantifying Different Types of Self-Admitted Technical Debt* (IEEE Computer Society, 2015), pp. 9–15.
9. T. Brown *et al.*, Language models are few-shot learners, in *Advances in Neural Information Processing Systems*, 33, (Curran Associates Inc., 2020), pp. 1877–1901.
10. J. Li, L. Li, J. Liu, X. Yu, X. Liu and J. W. Keung, Large language model chatgpt versus small deep learning models for self-admitted technical debt detection: Why not together?, *Software: Pract. Exper.* **55**(1) (2025) 3–28.
11. Q. Zhong, L. Ding, J. Liu, B. Du and D. Tao, Can chatgpt understand too? A comparative study on chatgpt and fine-tuned BERT, CoRR abs/2302.10198 (2023).
12. Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu and Y. Zhou, How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study, *ACM Trans. Softw. Eng. Methodol.* **30**(4) (2021) 45:1–45:56.
13. A. Potdar and E. Shihab, *An Exploratory Study on Self-Admitted Technical Debt* (IEEE Computer Society, 2014), pp. 91–100.
14. M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin and X. Yang, Automating change-level self-admitted technical debt determination, *IEEE Trans. Software Eng.* **45**(12) (2019) 1211–1229.
15. X. Chen, D. Yu, X. Fan, L. Wang and J. Chen, Multiclass classification for self-admitted technical debt based on xgboost, *IEEE Trans. Reliab.* **71**(3) (2022) 1309–1324.
16. M. S. Sheikhaei, Y. Tian, S. Wang and B. Xu, An empirical study on the effectiveness of large language models for SATD identification and classification, *Empir. Softw. Eng.* **29**(6) (2024) p. 159.
17. L. Chen, M. Zaharia and J. Zou, How is chatgpt's behavior changing over time?, *Harvard Datasci. Rev.* **6**(2) (2024) 1–47.
18. M. F. I. Amin, Y. Watanobe, M. M. Rahman and A. Shirafuji, Source code error understanding using BERT for multi-label classification, *IEEE Access* **13** (2025) 3802–3822.
19. A. McCallum and K. Nigam, A comparison of event models for naive bayes text classification, in *AAAI Conf. Artificial Intelligence*, 1998, pp. 41–48.
20. X. Chen, H. Jiang, X. Li, L. Nie, D. Yu, T. He and Z. Chen, A systemic framework for crowdsourced test report quality assessment, *Empir. Softw. Eng.* **25**(2) (2020) 1382–1418.
21. G. Aceto, D. Ciuonzo, A. Montieri and A. Pescapé, Multi-classification approaches for classifying mobile app traffic, *J. Netw. Comput. Appl.* **103** (2018) 131–145.
22. Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, Gradient-based learning applied to document recognition, *Proc. IEEE* **86**(11) (1998) 2278–2324.
23. W. Rahmaniar, Chatgpt for software development: Opportunities and challenges, *IT Prof.* **26**(3) (2024) 80–86.

24. B. Fluri, M. Würsch and H. C. Gall, *Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes* (IEEE Computer Society, 2007), pp. 70–79.
25. X. N. Zixuan Zhou *et al.*, A survey on efficient inference for large language models, arXiv:2404.14294 (2024).