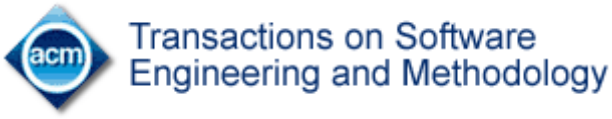

CupCleaner: A Data Cleaning Approach for Comment Updating

Journal:	*Transactions on Software Engineering and Methodology*
Manuscript ID	TOSEM-2023-0288
Manuscript Type:	Journal-First Paper
Date Submitted by the Author:	24-Aug-2023
Complete List of Authors:	Liang, Qingyuan; Peking University Zeyu, Sun; Zhongguancun Laboratory Zhu, Qihao; Peking University Hu, Junhao; Peking University Zhao, Yifan; Peking University Zhang, Lu; Peking University
Computing Classification Systems:	Data cleaning, Software evolution

CupCleaner: A Data Cleaning Approach for Comment Updating

QINGYUAN LIANG, Peking University, China
ZEYU SUN, Zhongguancun Laboratory, China
QIHAO ZHU, Peking University, China
JUNHAO HU, Peking University, China
YIFAN ZHAO, Peking University, China
LU ZHANG*, Peking University, China

Recently, deep learning-based techniques have shown promising performance on various tasks related to software engineering. For these learning-based approaches to perform well, obtaining high-quality data is one fundamental and crucial issue. The comment updating task is an emerging software engineering task aiming at automatically updating the corresponding comments based on changes in source code. However, datasets for the comment updating tasks are usually crawled from committed versions in open source software repositories such as GitHub, where there is lack of quality control of comments. In this paper, we focus on cleaning existing comment updating datasets with considering some properties of the comment updating process in software development. We propose a semantic and overlapping-aware approach named CupCleaner (Comment UPdating's CLEANER) to achieve this purpose. Specifically, we calculate a score based on semantics and overlapping information of the code and comments. Based on the distribution of the scores, we filter out the data with low scores in the tail of the distribution to get rid of possible unclean data. We first conducted a human evaluation on the noise data and high-quality data identified by CupCleaner. The results show that the human ratings of the noise data identified by CupCleaner are significantly lower. Then, we applied our data cleaning approach to the training and validation sets of three existing comment updating datasets while keeping the test set unchanged. Our experimental results show that even after filtering out over 30% of the data using CupCleaner, there is still an improvement in all performance metrics. The experimental results on the cleaned test set also suggest that CupCleaner may provide help for constructing datasets for updating-related tasks.

CCS Concepts: • **Information systems** → **Data cleaning**; • **Software and its engineering** → *Software evolution.*

Additional Key Words and Phrases: comment updating, data cleaning, software engineering

ACM Reference Format:
Qingyuan Liang, Zeyu Sun, Qihao Zhu, Junhao Hu, Yifan Zhao, and Lu Zhang. 2018. CupCleaner: A Data Cleaning Approach for Comment Updating. In . ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In recent years, an increasing number of artificial intelligence (AI) techniques, especially deep learning (DL) techniques, are explored for research on software engineering (SE) tasks [7, 33, 36, 44]. High-quality data is a key factor in deep learning-based approaches [42], and higher quality data often leads to more powerful models, while lower quality

*Lu Zhang is the corresponding author.

Manuscript submitted to ACM

```
// the forwarded value for object .
public final object reference get forwarded reference(object reference object)
{
  return object;
}

// the new location of object .
public object reference get forwarded reference(object reference object)
throws inline pragma
{                                                    Inserted Code
  if(object.is null()) return object;
  if(ss.hi & & space.is in space(ss.ss0, object))
  {
    return ss.copy space0.trace object(this, object);
  }
  else if(!ss.hi & & space.is in space(ss.ss1, object))
  {
    return ss.copy space1.trace object(this, object);
  }
  return object;
}
```

Fig. 1. An example of a nosing data sample in comment updating task.

data can have the opposite effect. This also applies to SE-related tasks, where more practical and realistic high-quality datasets can help train models that better improve developer productivity [31].

Comment updating is an important SE task that can also be boosted with deep learning techniques [16, 21, 22, 43]. This task aims to automatically update the corresponding comments based on code changes made by developers. On one hand, automatic comment updates can save the time and effort required for programmers to manually write new comments. Unlike directly generating comments, comment updating can take into account more information, such as code changes and old comments. Thus, this task is more practical and relevant to the daily development scenarios of programmers. On the other hand, if comments are not updated in a timely manner, they may mislead future development activities. For example, if obsolete TODO comments are not removed, they may introduce new bugs in future development [6]. For the task of comment updating, the data we expect is to be able to meet the real editing intention of developers and reflect the changes in code functionality.

However, due to the complex nature of the software development, it is difficult to avoid the introduction of noises for software engineering tasks such as comment updating. Besides, comment updating datasets are often crawled from online repositories, such as Github, by extracting commit history versions. This also makes it easier for noisy data to be included in datasets for the comment updating task[19, 28, 34, 41]. Figure 1 shows an example of a noisy data sample in the comment updating task. This sample includes old code and comment (top part of the figure), as well as new code and comment (bottom part of the figure). We can see that a new code snippet is inserted in the new code compared to the old code. The newly added code snippet mainly adds if-statements for conditional checking. In comparison to the old comment, the new comment has changed 'forwarded the value for' to 'the new location of'. From a code change perspective, the comment modification does not reflect the changes in the code. Therefore, we can consider such data as noise, because even humans would be confused with why the old comment is updated in this way when facing the code change. If the dataset contains a large number of such noisy data, the results generated by a model trained with the dataset would also be misleading.

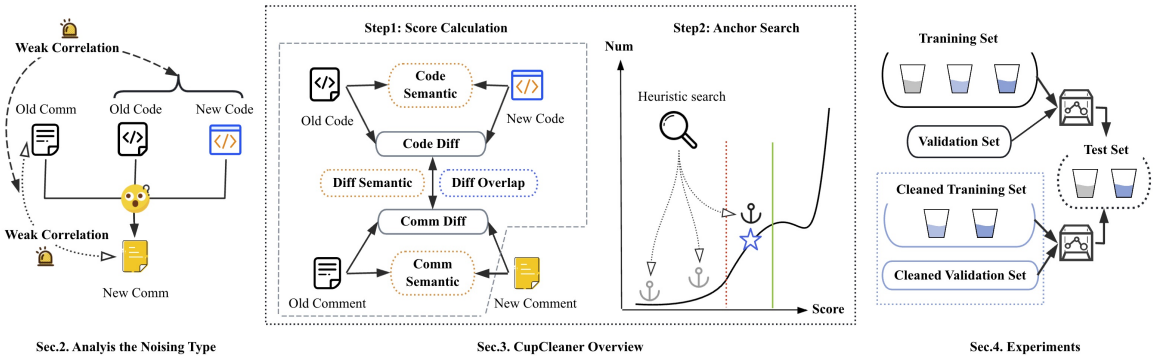

Fig. 2. Overview of our research procedure.

Figure 2 shows an overview of our research procedure. We first identified that there are two main types of noise in the comment updating: weak correlation between old and new comments, and weak correlation between code changes and comment changes. Then, in order to filter out noisy data and improve data quality in the comment updating dataset, we propose CupCleaner[1] (Comment UPdating's CLEANER), an automated data cleaning approach. A cup cleaner literally means a machine that cleans cups, but here we use it as a metaphor for using cleaned data to train DL-based model, just like using clean cups to serve coffee. CupCleaner consists of two steps. The first step is to design a criterion to calculate the score of all data and map them to a distribution. The second step is to find a suitable point from this distribution to determine which data to filter out. To identify noisy data, we first calculate the semantics within the comments or code. Then we consider the diff information of the code changes and comment changes. We use the semantic and overlap of the diff information to supplement the noise identification process. Finally, we consider all factors and summarize them into one criterion to calculate the quality score of each data sample. As different datasets may have varying sources of collection, in the second step, we design an anchor point search technique to adapt to different datasets. We are able to adaptively trim the tail of the score distribution, i.e., filter out the noisy data.

To evaluate our approach, we first conduct a human evaluation by mixing the noise data and high-quality data identified by CupCleaner. The evaluation results show that, with a maximum score of 5, the average score for the noise data identified by CupCleaner is only 2.8, while the high-quality data received a score of 4. Then we conduct experiments on three representative models: PLBART[1], UniXcoder[7], and CodeT5[36]. The experimental results show that cleaning the training and validation sets with our approach can effectively improve the performance of the models on the same test set. Specifically, for the dataset provided by *Panthaplackel et al., 2020*[22], we use CupCleaner to filter out more than 30% of the data in the training set, and the prefromance of the model trained on the cleaned dataset still improves on all performance metrics. For example, the token-level BLEU-4 score increases from 57.54 to 60.72, and the probability of generating identical comments increases from 19.97% to 21.6%. In addition, we also explore the impact of the cleaned test set and the pure noise test set on the model evaluation. We find that the evaluation performance on the cleaned test set is significantly better than that on the pure noise test set, regardless of whether the training set is cleaned or not. The performance of the cleaned test set evaluation is also consistent with the trend of the original test set, but higher than the original test set. This suggests that the cleaned test set alleviates the interference of noise and reflects the true ability of the model. Therefore, CupCleaner can not only be used to clean the training set

[1]Our code and data are publicly available at https://github.com/LIANGQINGYUAN/CupCleaner

and improve the performance of multiple models, but also to help construct real-world datasets to evaluate the real performance of a single model.

2 MOTIVATION

As depicted in Figure 2 (Sec.2), we analyze the types of noises in this section. The data sample for the comment updating task typically contains four parts: old code, old comment, new code, and new comment. These data in the updating scenarios are often more complex than individual code-comment pairs. Therefore, it is difficult to obtain large-scale high-quality dataset for updating-related tasks, and the data samples are likely to contain more noises. In this paper, we define "noise" in the dataset as data samples where the target output cannot be generated based on the existing input. In other words, these are data samples where there is a significant semantic gap between the input and the output after an update. Even for humans, it is challenging to understand why such an output is generated. The noises in the updating-related datasets mainly come from the weak correlation between information before and after the updates. In comment updating datasets, weak correlations are mainly coming from two aspects. The first weak correlation is within the old and new comments or code. The second weak correlation is between the comments changes and code changes. Below we discuss specific situations of these two types of weak correlations.

```
// Add a new element to this builder.
public XmlStringBuilder element(String name, CharSequence content) {
   return element(name, content.toString());
}

// Deprecated.
public XmlStringBuilder element(Element element) {
   assert element != null;
   return append(element.toXML());
}
```

(a). The comment that needs to be updated is invalid.

```
// Add to Log.
public void addLog(int P_ID, Timestamp P_Date, BigDecimal
P_Number, String P_Msg)
{
   final Timestamp timestampToUse = P_Date != null ? P_Date :
SystemTime.asTimestamp();
   addLog(new ProcessInfoLog(P_ID, timestampToUse, P_Number,
P_Msg));
}

// ********************************************************
**********
public void addLog(int Log_ID, Timestamp P_Date, BigDecimal
P_Number, String P_Msg)
{
   addLog(new ProcessInfoLog(Log_ID, P_Date, P_Number, P_Msg));
}
```

(b). The target comment is purely non-literal.

Fig. 3. Examples of noising data including invalid data.

On one hand, the correlation between some new and old comments may be weak, and even invalid comments can be found among them. If either the old comment or the new comment is invalid, then for the comment updating task, the data is not expected as of high quality. For example, comments that consist of purely invalid characters and comments that only contain a single unrelated word do not fit the scenario of real updates. In this type of noise, there are often purely disruptive contents that the model cannot reference, and learning from these data can typically

affect the accuracy of the results generated from the learned model. Figure 3 shows two examples of this noising type. Subfigure (a) shows an example of a new comment indicating that new code has been deprecated. However, the model cannot know whether new code means that the code has been deprecated, so it can mislead the model's generation process. Subfigure (b) shows an example where the new comment contains pure non-literal characters, which also is obviously noisy and could mislead the model.

On the other hand, the correlation between the code changes and the corresponding comment updating may also be weak. Specifically, if the code is significantly modified but the comments barely change, it belongs to this type of noise. These data may mislead the model to believe that comments do not need to be changed when code is modified in such a way. Similarly, if the code barely changes while the comments need to be significantly modified, this can also mislead the model. Additionally, noise can exist in data where both comments and code undergo changes but their changes are not related to each other. The data examples in Figure 1 illustrate this type of noise.

In summary, weak correlations between comments and code, as well as weak correlations between comment changes and code changes, are typical reasons for data noise in datasets for comment updating. These noisy data can mislead the model's generation process, thus reducing the overall performance. Therefore, we may need a criteria to evaluate the quality of individual data samples and filter out noises based on their data quality scores.

3 APPROACH

To evaluate the data quality and clean comment updating datasets, we consider the real requirements in comment updating task and design the CupCleaner to automate the data cleaning process. The overall framework of our approach is shown in Figure 2 (Sec.3). Our data cleaning process consists of two steps: calculating the quality score for all data samples, and searching for an anchor to filter out the noising data. This section describes the details of our data cleaning approach.

3.1 Score Calculation

To identify the noises caused by the two weak correlations mentioned above, we designed a criterion to calculate the quality scores for each data sample. The first type of weak correlation exists between comments or code within each data sample. We evaluate their correlation by calculating their internal semantic similarity. The second type of weak correlation exists between comment changes and code changes and we evaluate this correlation based on diff information. Below we discuss the detail of how to calculate the score.

Weak correlations existing between comments and code (Type I). This type of noises is a common data quality issue, mainly caused by invalid data. We evaluate this correlation by calculating their internal semantic similarity. To determine whether there is invalid data within the comments or code, we use the pre-trained models to convert them into embedding vectors to represent semantics. Currently, there are many pre-trained models used to represent the semantics of text [4, 5, 15]. We choose the state-of-the-art model pre-trained on code-related datasets, GraphCodeBert [8], to calculate the semantics of comments or code. It is worth noting that the semantic representation mentioned above is at the token level. As comments of source code typically consist of one or a few short sentences, we also use sentence-level embedding vectors [27] to represent the semantics of comments. We use the cosine similarity of semantic embeddings to describe the correlation between old and new comments, and between old and new code, respectively. We use multiplication to combine the token-level semantic representations of comments and code, as well as the sentence-level semantic representations of comments, in order to calculate the overall similarity score. If there are comments that are

not reasonable, their similarity will be lower, thus reducing the final score. Below we present the specific calculation details.

$$C^i_{token} = Cos(M_{gcb}(c^i_{old}), M_{gcb}(c^i_{new})) \tag{1}$$

$$C^i_{sent} = Cos(M_{sent}(c^i_{old}), M_{sent}(c^i_{new})) \tag{2}$$

$$S^i_{token} = Cos(M_{gcb}(s^i_{old}), M_{gcb}(s^i_{new})) \tag{3}$$

$$S^i_1 = C^i_{token} * C^i_{sent} * S^i_{token} \tag{4}$$

Where c_i represents the i-th comment, s_i represents the i-th snippet of code, M represents the pre-trained model, and Cos represents cosine similarity. We use two pre-trained models, GraphCodeBert and Sentence-BERT, denoted as M_{gcb} and M_{sent}, respectively, to compute semantic embeddings. Due to the fact that code comments contain not only code information but also pure natural language descriptions, in order to consider the semantic aspects of code comments, we additionally utilized Sentence-BERT alongside GraphCodeBert. Sentence-BERT helps in capturing natural language semantic information to enhance the overall performance and robustness of the evaluation. Finally, we compute the evaluation score S_1 for identifying noise in type I.

Weak correlation existing between comment changes and code changes (Type II). This type of noises is unique to the comment update task, mainly caused by inconsistencies between comment changes and code changes. We use the word-level diff information as the basic information for calculate the semantics and overlapping, and use them to evaluate the data quality score. We follow the approach proposed by [22] to get the basic diff information between comments and code. For the semantics of diff information, we concatenate the changed words in the diff into a string and use GraphCodeBert to calculate the semantic embeddings. However, the semantic in diff information alone is not sufficient to describe the similarity between changes, as it does not consider the evolutionary relationship between comments or code. Thus we combine the diff semantics with the semantics between code and comments. We utilize the following formula to calculate the score for the semantic of diff information.

$$D^i = Cos(M_{gcb}(dc^i), M_{gcb}(ds^i)) \tag{5}$$

$$S^i_2 = D^i * Max(C^i_{token}, S^i_{token}) \tag{6}$$

Where dc^i denotes the diff information among comments in the i-th data sample, while ds^i denotes the diff information among source code in the i-th data sample. We use Max to represent that the semantics of the diff only needs to be combined with the maximum value of the semantic similarity between comments and code. This is actually intuitive, as once the semantic of changes is similar, and if there is a high similarity between the comments or code, then it is highly likely to be a data sample that fits the update scenario. For the overlapping perspective, we consider the overlap of words at the token level as the similarity. The basic calculation process remains the same as above.

$$O^i = Overlapping(dc^i, ds^i) \tag{7}$$

$$S^i_3 = O^i * Max(C^i_{token}, S^i_{token}) \tag{8}$$

In the specific overlapping calculation, we use the longest common subsequence (LCS) to determine whether the changed words in the comments and the changed contents in the code are similar. Due to the differences in syntax and vocabulary between comments and code, we employ LCS to consider the similarity of the letter compositions within different words. For each word dc^i_j in the i-th comment change dc^i, we calculate the length of the LCS between dc^i_j and

each word in the code change ds^i_k, denoted as l^i_{jk}. The similarity score of overlapping is then defined as the ratio of l^i_{jk} to the length of dc^i_j. We select the k-th word in ds^i that is most similar to dc^i_j as the final score of dc^i_j and denoted as L^i_{jk}. We use L^i_{jk} as the score for the j^{th} word in the comment. Then, we take the average of the scores of all words in dc^i as the overlapping score for the i^{th} comment.

Finally, we take into account the above-mentioned factors and calculate a unified score to measure the data quality of comment updating.

$$S^i = Max(S^i_1, S^i_2, S^i_3) \tag{9}$$

We take the maximum score among S^i_1, S^i_2, and S^i_3 as the final score for the i-th sample. If a sample performs well in one of the scores, we usually consider it as a good comment updating data. CupCleaner maps all the scores obtained from the entire dataset to a distribution. The horizontal axis of the distribution represents the scores, and the vertical axis represents the number of data samples that obtained that score. The final score distribution is similar to the distribution in step 2 of Figure 2 (Sec.3), where most of the high-quality data are located on the right side, while low-quality data are distributed on the left side.

3.2 Anchor Search

The second step of CupCleaner is to search for an anchor to determine where to trim the tail of the distribution. However, the comment updating data in different datasets come from different open source projects and have different distributions. To better adapt to the characteristics of different datasets, we design a simple search approach to find the anchor point for trimming in the score distribution.

The distribution of scores is such that the majority of high-score data is on the right, and a small portion of low-score data is on the left. Step 2 of Figure 2 (Sec.3) demonstrates the basic process of searching for an anchor point. Whenever we select an anchor point in the distribution, the data to its left is filtered out. The area of the distribution to the left of the anchor point represents the proportion of data that is being filtered out. The basic idea of searching for an anchor point involves two steps. First, we select an initial position on the left side of the distribution. Then, we continuously move this position to the right and compare the difference in area changed. The point is selected as a potential anchor point only after the data changes beyond a certain threshold. Step 2 of Figure 2 (Sec.3) also shows three anchor points at different thresholds.

Algorithm 1 describes the details of search process. We select the initial position in the distribution as the point located at the $\mu - 2 * \delta$, where μ and δ represents the mean and the variance respectively. In a normal distribution, the area to the left of this point represents 5% of the entire distribution, making it a representative choice. Since there are more scores on the right-hand side of the distribution of data quality scores, the initial point is likely to fall into a region with plenty of noisy data. That satisfies the condition for searching towards the right. Therefore, we initialize λ to 2 (line 1), where λ represents a search parameter that reflects the movement of the search. For example, a decrease in λ means that the selected point is moving to the right. We set the area change threshold to be between 0.01 and 0.1, and the change in λ per iteration to be 0.01 (line 2-4). We compare the change in area generated before and after the change in λ. If the change in area exceeds the set threshold, we stop and record the anchor point generated at that threshold (line 5-6). Function AreaChanged in Algorithm 1 is used to calculate the specific area change. This function first calculates the position of the point generated using the old λ (line 12-15), then calculates the position of the point using the new λ (line 16-18), and finally returns the change in area before and after the change and the specific point (line 19-21). If a point is moved all the way to the right and the area change caused by lambda does not exceed the

Algorithm 1 Search an anchor to filter out noising data

Input: A set of quality scores S, including s_0, s_1, ..., s_n.
Output: A set of anchor points P.

```
Initialize λ to 2.
for threshold = 0.01 to 0.1 step 0.01 do
    for i = 1 to 201 step 1 do
        x = i * 0.01
        R_c, p = AreaChanged(S, λ, x)
        if R_c > threshold then Add p to P.
        end if
    end for
end for
return P
function AreaChanged(S, λ, x)
    μ = average score of S
    δ = standard deviation of S
    S_o = a set of scores where s_i lower than μ − λ * δ
    R_o = The ratio of the number of scores in S_o to the number of scores in S.
    λ' = λ − x
    S_n = a set of scores where s_i lower than μ − λ' * δ
    R_n = The ratio of the number of scores in S_n to the number of scores in S.
    R_c = The gap between R_o and R_n.
    p = μ − λ * δ
    return R_c, p
end function
```

threshold, no point is selected as the anchor point. In general, the area change caused by lambda does not exceed 5% of the entire distribution, which is also intuitive. In CupCleaner, we adopt an aggressive cleaning strategy to select the final anchor points with the highest possible threshold. Just like in step 2 of Figure 2 (Sec.3), where there are three potential anchor points, we choose the last anchor point to filter out more data.

4 EXPERIMENTAL SETUP

In this section, we first introduce the research questions (RQs). Then we describe the three datasets for comment updating task and the metrics for evaluating the effectiveness of updated comment. Finally we present baselines and experiment settings.

4.1 Research Questions

To evaluate the effectiveness and the efficiency of CupCleaner, we aim to answer the following questions:

RQ1: Whether CupCleaner can filter out true noise data? To evaluate CupCleaner's ability to remove noise data, we first record the changes in the number of data. Then we provide examples that CupCleaner identifies as noisy and high-quality data, respectively. Finally, we conducte human evaluation on the noisy and high-quality data identified by CupCleaner to explore whether the filtered data are truly noise.

RQ2: How effective is the training data cleaned by CupCleaner? To evaluate the effectiveness of our data cleaning approach, we conduct experiments on three representative models, namely PLBART, UniXcoder, and CodeT5. During

Table 1. Table of dataset statistics.

Name	*Panthaplackel et al., 2020*	*Liu et al., 2020*	*Panthaplackel et al., 2021*
Year	2020	2020	2021
Published	ACL[1]	ASE[2]	AAAI[3]
Language	Java	Java	Java
#Pairs	7239	104805	20344
Train/Valid/Test	5791/712/736	85657/9475/9673	16494/1878/1972

the experiment, we keep the test set unchanged and use CupCleaner to clean only the training and validation sets. We evaluate the effectiveness of CupCleaner by comparing the performance of the models trained on the cleaned dataset with the models trained on the original dataset .

RQ3: How effective is the use of CupCleaner-cleaned data for evaluating models? To further investigate whether the data cleaned by CupCleaner can lead to more reasonable model evaluation, we conduct experiments on different versions of the test set using the best-performing model from the previous research question. We divide the test set into two versions, cleaned and noise, based on the anchor points searched on the non-test set, and analyze the effectiveness of using the cleaned test set to evaluate the model.

RQ4: How efficient is CupCleaner? The calculation of quality scores for each data sample involved the use of the pre-trained models. This also brings a potential risk that the time required to compute the scores for all data samples may be unacceptable. To answer this question, we calculate the time consumption for cleaning the entire dataset, including the time required to compute scores and filter out the data. We chose the time it takes to train a model for one epoch as a baseline for comparing the time required to clean the data.

4.2 Datasets

In order to evaluate the effectiveness of our data cleaning approach, we conduct experiments on three datasets: *Panthaplackel et al., 2020* [22], *Liu et al., 2020* [16], *Panthaplackel et al., 2021* [21]. Table 1 describes the basic statistics of these three datasets. These datasets contain both old and new comment-code pairs that are considered to have potential update relationships. The inputs for these datasets include old comment, old code, and new code, while the output is the updated new comment.

Specifically, the *Panthaplackel et al., 2020* dataset collects 7.2k data samples from the commit history of open-source Java projects on Github. The *Liu et al., 2020* dataset is constructed from 1,063K method-doc co-change instances from Github, resulting in 104k data samples. The *Panthaplackel et al., 2021* dataset is originally designed for comment consistency detection, but some of the data can also be used for the comment updating task. We extract the data with comment changes from this dataset to construct a new comment updating dataset, and discard the data with no changes in the comments. The processed dataset contains around 20k data samples that can be used for research on comment updating. These datasets are constructed using simple cleaning approaches to select data with potential comment

[1] Association for Computational Linguistics.
[2] International Conference on Automated Software Engineering
[3] Association for the Advancement of Artificial Intelligence

updating relationships. For example, the *Panthaplackel et al., 2020* dataset filtered out comment changes that are purely stylistic, such as spelling corrections, re-formatting. The *Liu et al., 2020* dataset, on the other hand, excluded samples containing special tags and those with excessively long edit distances. As for *Panthaplackel et al., 2021* dataset, they removed samples where comment changes are minor and samples where the code couldn't be parsed into an AST. However, due to a lack of exploration into the characteristics of comment updating, the constructed datasets still contain a significant amount of noisy data.

4.3 Metrics

To evaluate the performance of the models, we select six representative evaluation metrics in the field of text generation and updating, including XMatch, BLEU at the token level (BLEU4-T), BLEU at the sentence level (BLEU4-S), METEOR, SARI, and GLEU.

4.3.1 XMatch. The first metric represents exact match, i.e, the percentage of generated comments that are identical to the ground truth at the string level.

4.3.2 BLEU4-T. The second metric we use is the token-level BLEU score. BLEU[23] is originally used to evaluate the performance of machine translation. BLEU [23] is commonly used to evaluate the similarity of generated code from a lexical perspective [14, 31]. The token-level BLEU matches and calculates the generated results and reference texts on a word-by-word basis, so it can capture word-level similarity and accuracy.

4.3.3 BLEU4-S. To consider both token-level and sentence-level similarities, we use sentence-level BLEU [16, 23] as the third metric. The calculation of sentence-level BLEU is based on matching and calculating the whole sentence, thus it focuses more on the fluency and readability at the sentence level. We use BLEU-4 as the BLEU score at both the token level and the sentence level.

4.3.4 METEOR. Our fourth metric is METEOR [3]. METEOR utilizes word alignment to align the model-generated results with the reference texts and evaluate the generated results. Compared to BLEU, METEOR does not rely on n-gram matching, but instead places more emphasis on the syntax, word order, and semantics of the generated results.

4.3.5 SARI. SARI [40] (System for Automatic Evaluation of Text Simplification) is an evaluation metric based on edit distance, originally designed to assess the overall quality of text simplification. SARI compares the generated results with the reference in terms of edit distance, as well as the matching of words, phrases, and sentences at various levels, to provide an overall assessment of the quality of the generated results.

4.3.6 GLEU. Our last metric is GLEU [20] (Generalized Language Evaluation Understanding). GLEU is a variant of the BLEU metric originally designed for evaluating grammatical correction systems. By rewarding correct edits and penalizing ungrammatical ones, GLEU is closer to human-level judgment than BLEU and is more suitable for tasks involving edits.

4.4 Baselines

Data and models are both important factors that can significantly affect experimental results. To evaluate our data cleaning approach more fairly, we use some general baseline models instead of one designed specifically for a particular dataset. We select three general and widely used models, namely PLBART [1], UniXcoder [7], and CodeT5 [36], as

baselines to evaluate our data cleaning approach. These models are recently proposed and achieved state-of-the-art results on multiple code-related tasks, making them both representative and advanced for our purpose.

4.4.1 PLBART. PLBART is a pre-trained model based on the BART[12] architecture and pre-trained on programming language data. The used pre-training data include Java and Python code collected from Github, as well as natural language descriptions collected from StackOverflow. PLBART's pre-training uses denoising sequence-to-sequence techniques, including three types of noising strategies: token masking, token deletion, and token infilling. The experimental results demonstrate that PLBART outperforms CodeGPT[18] and CodeBERT[5] in multiple code understanding and code generation tasks.

4.4.2 UniXcoder. UniXcoder is a unified cross-modal pre-trained model designed for programming language, based on the Transformer[35] framework. UniXcoder incorporates semantic information from code comments and syntactic information from AST, enabling it to support both code-related understanding and generation tasks. UniXcoder was pre-trained on two datasets: C4[25] and CodeSearchNet[10]. The pre-training objectives include masked language modeling[4], unidirectional language modeling[24], and denoising objective[26]. The experimental results show that UniXcoder achieves improvements on multiple code-related tasks.

4.4.3 CodeT5. CodeT5 is a pre-trained model based on the encoder-decoder structure of T5[26]. CodeT5 considers the identifier in the code in addition to T5's denoising sequence-to-sequence pre-training. During the pre-training process, CodeT5 applies four pre-training objectives, namely masked span prediction, identifier tagging, masked identifier prediction, and bimodal dual generation. Identifier tagging is used to label whether a token is a code identifier, while the masked identifier prediction task is used to generate the masked identifier content. Experimental results demonstrate that CodeT5 can better capture semantic of code, and outperforms other approaches significantly on tasks related to code.

4.5 Implementation Settings

The main experimental environment is an Ubuntu OS with two Intel Xeon CPUs, 188GB RAM, and a NVIDIA TITAN RTX GPU with 24GB of memory. To save training time and resources, we set the maximum number of epochs for training to 20 and implement an early stopping strategy. Specifically, we implement a common strategy during the training process, where we terminate the training process if the experimental performance on the validation set does not improve within three consecutive epochs. For each dataset, the model settings we use are exactly the same across all variants of that dataset. In addition, to make the experimental process more generalizable, we simply concatenate the inputs into a long sequence without performing any special processing.

5 RESULTS

In this section, we present the experimental results and answer each research question.

5.1 Noise data identified by CupCleaner (RQ1)

To answer this question, we explore the data filtered out by CupCleaner. First, we present the statistical information of data cleaning, then we analyze some samples of high-quality and low-quality data that CupCleaner identifies, and finally we conduct a human evaluation for the identified clean data and noisy data.

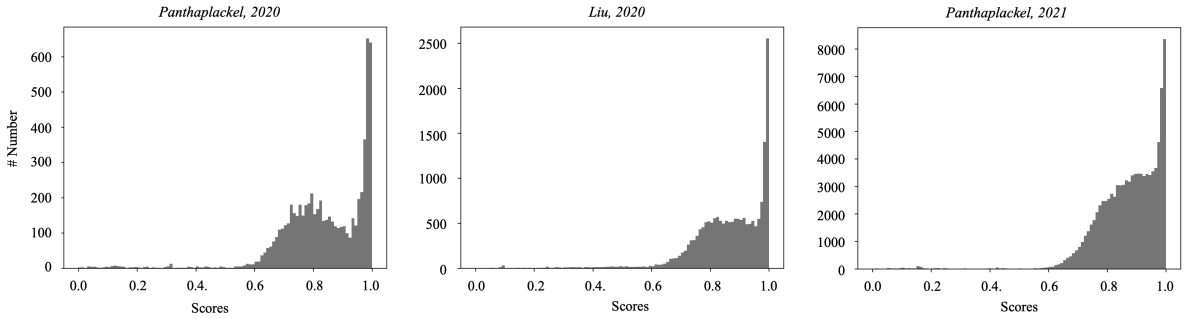

Fig. 4. The score distribution of the three datasets.

Table 2. Table of dataset statistics after data cleaning.

Dataset Name	Preprocess	#Train	#Valid	#noising	threshold	anchor	delete rate
Panthaplackel et al, 2020	Original	5,791	712	-	-	-	-
	Random	3899	471	-	-	-	-
	Cleaned	3899	471	2127	0.04	0.7836	32.7%
Liu et al, 2020	Original	85,657	9,475	-	-	-	-
	Random	74975	8403	-	-	-	-
	Cleaned	74975	8403	11754	0.03	0.7605	12.36%
Panthaplackel et al, 2021	Original	16,494	1,878	-	-	-	-
	Random	13556	1523	-	-	-	-
	Cleaned	13556	1523	3293	0.02	0.7703	17.92%

The final scores of the three datasets are shown in Figure 4, and the overall shape of the distribution is consistent with the schematic in Figure 2 (Sec.3). The red dashed line in the second step of Figure 2 (Sec.3) corresponds approximately to 0.65, and the green solid line corresponds to 0.8 in the score distribution. We can observe from the distribution that the part below 0.65 is certainly the tail of the distribution and the part above 0.8 is unlikely to be the tail of the distribution. In other words, scores below 0.65 may likely be noisy data, while scores above 0.8 are unlikely to be noisy data. Table 2 shows the basic statistical information of the filtered dataset on the training and validation sets. It is worth noting that if the anchor points we search for exceed the green line in the second step of Figure 2 (Sec.3) (i.e., if they exceed 0.8), we choose to use the anchor of the previous search. Furthermore, it can be seen from Figure 4 that the majority of data is located on the right of 0.8, which is not in line with our initial intention of trimming the tail of the score distribution. As shown in Table 2, we filter out approximately 33%, 12%, and 18% of the data in the three datasets, respectively.

To demonstrate the discriminating ability of CupCleaner on high-quality and noisy data, we provide a data sample analysis. Previously presented Figures 1 and 3 are both data that CupCleaner can recognize as noisy. In addition, Figure 5 (a) illustrates the case where the comments are insufficient to describe the code and the comment changes are inconsistent with the code changes. Figure 5 (b) illustrates the case where the comment remains almost unchanged after the code change, where the new comment adds label-like content that cannot be captured from code changes. Figure 5 (c) shows a data sample that CupCleaner considers to be of high quality, where the update logic is clear and

```
// Typical usage :
public static Offset<Double> byLessThan(Double value) {
   return Offset.offset(value);  }

// Examples:
public static Offset<Double> byLessThan(Double value) {
   return Offset.strictOffset(value);  }
```

(a). The comments are not enough to describe the code, and the changes in comments and code are inconsistent.

```
//total sum of squares
public double calculate total sum of squares()
{  return new second moment().evaluate(y.get data());  }

// ssto - the total sum of squares
public double calculate total sum of squares()
{
   if(is no intercept())
   { return stat utils.sum sq(y.get data()); }
   else
   {  return new second moment().evaluate(y.get data());  }
}
```

(b). While the code undergoes significant changes, the comments fail to reflect the changes in the code.

```
// @return a hashmap of all formulars
public HashMap<String, String> getFormulas() {
   return formulas;  }
// @return a list of all formulas
public PactFormulaList getFormulas() {
   return formulas;  }
```

(c). An example where the update intention is clear and the code changes match the comment changes.

Fig. 5. Examples of noising data with weak correlation between code changes and comment changes.

meets the requirements of the update scenario. These data samples all demonstrate that CupCleaner can distinguish whether they are noisy data.

To further investigate whether the data filtered out by CupCleaner is truly noisy, we conduct a small-scale human evaluation, where three Ph.D students majoring in computer science participate in a human evaluation process. Each Ph.D student has over 4 years of development experience and do the study independently. We first randomly select 50 samples from the high-quality data and 50 samples from the noisy data identified by CupCleaner in the dataset of *Panthaplackel et al., 2020*. Then, we make sure that the evaluators understand the context and the intention of the comment updating task and ask them the question "Would you accept this data sample into the comment updating dataset?" This question means that the annotators need to judge whether the comment update process of each data is reasonable and understandable to humans. The label for each data sample ranges from 1 to 5, representing reject, weak reject, weak accept, accept, and strong accept, indicating the willingness to accept the data. If annotators encounter difficulties in determining whether a data sample can be used for the comment updating task, they may assign a rating after conducting discussions. Finally, the average score of data that CupCleaner considers to be retained is 4.0, while the average score of data filtered out by CupCleaner is 2.8. This indicates that the majority of the data filtered out by CupCleaner belong to noisy data, while the data retained is predominantly in line with the intent of the comment updating task.

Table 3. Table of experimental results.

Dataset Name	Models	Preprocess	XMatch	BLEU4-T	BLEU4-S	Meteor	SARI	GLEU
Panthaplackel, 2020	PLBART	Original	13.72	22.32	44.59	39.42	42.96	38.54
		Random	14.4	35.79	46.21	41.56	43.12	40.84
		Cleaned	**15.9**	**42.55**	**49.73**	**44.1**	**45.12**	**43.8**
	UniXcoder	Original	**20.52**	55.52	49.92	46.15	45.14	45.89
		Random	18.07	55.63	49.47	44.39	43.92	45.32
		Cleaned	19.57	**57.76**	**51.11**	**46.21**	**45.8**	**47.45**
	CodeT5	Original	19.97 (25.46 / 4.64)	57.54 (64.81 / 36.17)	52.95 (59.95 / 33.38)	47.15 (52.8 / 31.37)	46.88 (48.57 / 38.3)	47.44 (54.2 / 28.44)
		Random	20.52 (25.65 / 6.19)	58.79 (65.21 / 38.42)	53.22 (60.29 / 33.46)	47.63 (53.19 / 32.09)	46.93 (48.7 / 38.64)	48.16 (54.86 / 29.33)
		Cleaned	**21.6** (**27.49** / 5.15)	**60.72** (**67.4** / 39.7)	**53.9** (**61.23** / 33.41)	**48.21** (**54.18** / 31.52)	**47.34** (**49.21** / 38.03)	**49.11** (**56.15 / 29.32**)
Liu, 2020	PLBART	Original	23.48	65.01	65.88	58.67	48.34	56.93
		Random	23.41	69.21	66.42	59.05	48.59	56.98
		Cleaned	**24.5**	**73.58**	**69.12**	**61.57**	**49.46**	**58.42**
	UniXcoder	Original	30.64	72.63	68.54	62.57	50.87	61.15
		Random	31.77	72.08	68.68	62.73	50.47	62.3
		Cleaned	**32.52**	**73.63**	**70.2**	**63.94**	**51.44**	**63.25**
	CodeT5	Original	32.54 (35.36 / 7.87)	76.06 (77.7 / 57.01)	71.57 (73.95 / 50.73)	65.36 (**67.91** / 43.02)	52.03 (**55.59** / 42.17)	63.79 (66.15 / 43.03)
		Random	**32.99** (35.67 / 9.49)	75.69 (77.22 / 58.0)	71.34 (73.61 / 51.42)	64.94 (67.32 / 44.03)	51.86 (55.44 / 43.16)	64.06 (66.29 / 44.45)
		Cleaned	32.97 (**35.73** / 8.78)	**76.1** (**77.7** / 57.26)	**71.91** (**74.27** / 51.3)	**65.42** (67.89 / 43.8)	**52.16** (55.57 / 42.95)	**64.06** (**66.37** / 43.71)
Panthaplackel, 2021	PLBART	Original	**22.67**	54.3	53.53	51.34	48.24	**48.72**
		Random	20.94	50.86	52.75	50.17	47.68	47.77
		Cleaned	22.62	**56.33**	**54.38**	**51.67**	**48.49**	48.42
	UniXcoder	Original	25.86	55.4	54.09	52.87	48.85	49.9
		Random	26.62	55.59	54.66	53.6	48.98	50.2
		Cleaned	**26.88**	**57.58**	**56.29**	**54.1**	**49.44**	**51.17**
	CodeT5	Original	27.54 (31.64 / 6.27)	59.02 (64.08 / 30.72)	57.99 (62.64 / 33.89)	55.8 (59.75 / 35.33)	50.66 (52.73 / 39.57)	51.82 (56.48 / 27.53)
		Random	27.69 (32.12 / 4.7)	59.02 (64.16 / 30.39)	58.23 (62.97 / 33.74)	55.7 (59.98 / 33.55)	50.73 (52.95 / 39.62)	52.34 (56.97 / 28.28)
		Cleaned	**28.25** (**32.55** / 5.96)	**59.58** (**64.74** / 30.66)	**58.51** (**63.21** / 34.17)	**56.28** (**60.32** / 35.38)	**50.90** (**53.26** / 39.48)	**52.34** (**57.02** / 28.0)

Answer to RQ1: We filter out approximately 33%, 12%, and 18% of the data in three datasets, respectively. The results of the human evaluation show that the average score of the data filtered out by CupCleaner is only 2.8/5, while the average score of the high-quality data identified by CupCleaner as high-quality is 4.0/5. This indicates that CupCleaner can distinguish noisy data from high-quality data in comment updating datasets.

5.2 Effectiveness of training using cleaned data (RQ2)

To answer RQ2, we conduct experiments on three representative architectures, namely PLBART, UniXcoder, and CodeT5. For each dataset, while keeping the test set unchanged, we organize the remaining data (i.e., training data and validation

data) into three versions: original, random, and cleaned. The original version refers to the raw data, while the cleaned version is the data cleaned by CupCleaner. The random version is a subset of the original data with the same size as the cleaned data. Our experimental settings ensure that the same settings are used for different versions of the same dataset. In each experiment on each dataset, we also ensure that the original test set remained unchanged and we process only the training and validation sets.

Table 3 shows the performance statistics for each dataset and each model. We can see from Table 3 that among all models, CodeT5 has the best performance, followed by UniXcoder and PLBART. We can see from each model that simply cleaning the training and validation sets can improve multiple evaluation metrics. Specifically, for the *Panthaplackel et al., 2020* dataset, using the cleaned data for training and validation significantly improves the model's BLEU-related metrics. For example, the PLBART model improves the BLEU4-T metric from 22.32 to 42.55, an improvement of almost 100%, UniXcoder improves the BLEU4-T metric from 55.52 to 57.76, and CodeT5 improves the BLEU4-T metric from 57.54 to 60.72. It can be seen that regardless of whether the model itself performs well on the original dataset, the cleaned data can further improve their performance. Additionally, training on the dataset cleaned by CupCleaner outperforms using randomly sampled data of the same size, further demonstrating the effectiveness of CupCleaner.

It can be observed that the dataset of *Panthaplackel et al., 2020* is most affected by noisy data. Because this dataset has the highest proportion of filtered data, indicating that there are a large number of low-scored data samples. Correspondingly, training on the cleaned data results in the greatest performance improvement for this dataset. Compared to that, the impact of noisy data on dataset *Liu et al., 2020* and *Panthaplackel et al., 2021* is relatively small, but cleaning the data can still lead to improvements in almost all metrics.

> **Answer to RQ2**: We conduct experiments with the three models on the three comment updating datasets. The experimental results show that using the cleaned data for training without changing the test set can improve almost all the metrics on all models. For example, CodeT5 trained on the cleaned dataset improved the BLEU4-T from 57.54 to 60.72 on the first dataset. CupCleaner shows its effectiveness regardless of whether the model performs well on the original dataset or whether the original data is heavily contaminated with noises.

5.3 Effectiveness of evaluating models using cleaned data. (RQ3)

To answer RQ3, we design a experiment on the test sets of both identified noises and cleaned versions. We filter test set based on the anchor searched on the non-test set of each dataset. The data with scores higher than the anchor point are assigned to the cleaned version of the test set, while the rest are assigned to the noisy version of the test set. As the answer to RQ2 has revealed that the CodeT5 performed the best, we select CodeT5 to conduct this experiment.

We summarize the experimental results of CodeT5 in Table 3, where the results in the format of *(number1 / number2)* represent the evaluation results on the cleaned and noisy test sets, respectively. For example, in the *Panthaplackel et al., 2020* dataset, the XMatch metric (27.49 / 5.15) for CodeT5 trained on cleaned data indicates that 27.49% of the predicted outputs perfectly match the reference on the cleaned test set, while only 5.15% on the noise test set. It is worth noting that the number of samples in the noisy version of the test set is significantly less than that in the cleaned version of the test set. Therefore, the 5% corresponding to the noisy version represents a much smaller number of samples compared to the 27% corresponding to the cleaned version.

The experimental results on different test sets in Table 3 led to two main findings:

The first finding is that evaluation results on the cleaned test set significantly outperform the results on the noisy test set. For example, in the *Panthaplackel et al., 2021* dataset, when comparing the evaluation results on the cleaned test

set and the noise test set, the XMatch metric is on average 400% higher and the BLEU4-T metric is on average over 100% higher on the cleaned test set. In addition, the evaluation results on the noisy test set are not only worse but also inconsistent across different training data, with different models showing varying performance. This suggests that the test data filtered out by CupCleaner deviates significantly from the intended comment updating, resulting in poor performance regardless of whether the original training set or the cleaned training set is used.

The second finding is that evaluating the model on the cleaned test set shows a similar trend as evaluating it on the original test set, but with improved performance compared to the original. When evaluating the model on the clean test set of the *Panthaplackel et al., 2021* dataset, the model trained on the cleaned training set outperforms the models trained on other dataset variants on all metrics, and the XMatch score improved from 28.25 to 32.55. This finding indicates that the cleaned test set may provide a more reasonable evaluation of the model's performance. That is to say, the cleaned test set, by removing data that deviates from the intent of the comment updating task, better reflects the true ability of the model. Thus, during the construction of comment updating-related datasets, CupCleaner may provide help for producing datasets that reflect the real capabilities of the model.

> **Answer to RQ3**: We compare the performance of different models on the cleaned test set and the noisy test set. First, we find that the performance on the cleaned test set is significantly better than that on the noisy test set, with an improvement of 400% in XMatch on the *Panthaplackel et al., 2021* dataset. This indicates that we filter out data that do not match the intention of comment updating. Second, we find that the evaluation results on the cleaned test set are higher than those on the original test set in all metrics, but with the same trend. This indicates that the cleaned test set may better reflect the model's real performance, and also shows that CupCleaner may help in constructing datasets for accurate measurement of model abilities.

5.4 Time consumption of data cleaning process (RQ4)

Table 4. Table of the time consumption.

	Panthaplackel et al, 2020	*Liu et al, 2020*	*Panthaplackel et al, 2021*
Training one epoch	6 minutes	1 hour and 40 minutes	23 minutes
Compute semantics	3 minutes	1 hour and 19 minutes	11 minutes
Compute scores & Search anchor	30 seconds	6 minutes	2 minutes

To answer RQ4, we examine the time required to clean the dataset and compare it to training one epoch of the model on the raw dataset. An epoch means that the model goes through all the training data once and adjusts its weights based on the loss function during the training process. We select CodeT5 as the baseline model to calculate the time required to train one epoch, as it has exhibited relatively high performance in previous experiments. In one epoch of model training, we calculate the time spent on the training set and the time it spent on validation set after one epoch. During the data cleaning process, we calculate the total time spent on semantic analysis, final score calculation, and searching for anchor points.

Table 4 shows the basic statistics of time consumption. We can see from the table that the time consumption of our data cleaning approach is less than the time required for the model to train one epoch on all three datasets. Specifically, on datasets *Panthaplackel et al., 2020* and *Panthaplackel et al., 2021*, the time required for data cleaning is roughly half of the time required for training one epoch of the model. Even on the dataset *Liu et al., 2020*, which contains almost 100k examples, the time required for data cleaning is still less than the time required for training one epoch. It is worth noting that we have not intentionally optimized or reduced the time required for data cleaning. This indicates that our data cleaning approach can produce cleaned data within an acceptable time. In addition, using common early stopping strategies can significantly reduce the number of epochs required for training. For example, the model is trained until the performance on the validation set stops improving for 3 consecutive times, then the training is terminated. Therefore, using the time equivalent to training one epoch to clean better data may a reasonable choice.

> **Answer to RQ4**: We compare the time required to train an epoch with the time required to clean the dataset. The experimental results indicate that the total time required to clean the dataset is often less than the time required to train an epoch. Thus, the time to clean the dataset may not be a burden for training.

6 THREATS TO VALIDITY

Threats to internal validity may arise from two factors. The first threat relates to human evaluation. In the human evaluation, we presented annotators with only one question and asked them to provide ratings based on their responses to this question. A comprehensive assessment of a data sample may require analysis from multiple perspectives. However, the annotators we invited are all Ph.D students with relevant development experience. To ensure their understanding of comment updating task, we also engaged in discussions during the annotation process to mitigate the potential impact of this threat. The second threat is that while filtering out noisy data, we might inadvertently exclude normal data as well. This is easily understand because defining whether a data sample is noise can be challenging. On the one hand, programmers have different ideas and styles when coding, and on the other hand, many data samples require in-depth understanding to determine their appropriateness. However, our experimental results indicate that regardless of whether the test set is cleaned or not, our cleaned training set leads to improvements. In addition, the results of human evaluation also demonstrate that our cleaned data aligns better with the scenarios and intentions of the comment updating tasks.

Threat to external validity might come from the choose of the baseline models. In our experiments, we exclusively chose pre-trained models and did not include non-pre-trained models, which could potentially introduce bias into the final experimental results. We assume that pre-trained models possess a broader base of knowledge, making them more capable of understanding downstream tasks compared to training from scratch. Actually, pre-trained models typically exhibit better performance and also being employed in various generative tasks.

7 RELATED WORK

Below we discuss the two most related work of our paper: editing/updating-related tasks and data cleaning techniques.

7.1 Editing/Updating-Related Tasks

Editing and updating are fundamental nature to the software development process, and many software software development tasks are closely associated with these activities. For example, automatic program repair involves updating code containing bugs to correct bugs[11, 17, 37–39, 44], code review involves editing old code based on review

comments[2, 13, 29], and comment updating involves automatically updating old comments based on code changes[16, 21, 22, 43].

These types of tasks are often more closely related to the daily development scenarios of developers. Recently, there is also a large amount of deep learning research focused on tasks related to editing or updating. *Zhu et al.* [44] proposed Recoder, a syntax-guided edit decoder for the Automated Program Repair (APR) task. Recoder efficiently represents minor code changes by generating edits instead of the entire modified code. In the design of Recoder, the authors use a novel provider/decider architecture to ensure syntactic correctness and provides a placeholder generation mechanism to generate specific program identifiers. Experimental results indicate that Recoder achieves a 26% improvement over Defects4J v1.2 dataset and become the first deep learning-based approach that outperforms traditional APR approaches. *Li et al.* [13] collect a large-scale dataset of code changes and code reviews to pre-train model, CodeReviewer, to fit the tasks in the code review scenario. CodeReviewer applies three pre-training tasks, namely code change quality estimation, code review generation and code refinement. Specifically, code change quality estimation task aims to predict whether a code change is high-quality and can be accepted by reviewer. Code review generation task is to generate the review comment and code refinement task is to generate revised code based on review comments. Experimental results shows that CodeReviewer outperforms other models in the code review scenario. *Liu et al.* [16] proposed CUP, a comment updating approach that can be used to assist developers automatic updating comments and avoid bad comments. The authors collect a dataset with comment-code co-change samples for support comment updating task and utilize sequence-to-sequence model to generate updated comment. The evaluation results indicate that CUP outperforms baselines.

However, datasets supporting such tasks are often mined from the commit versions of open-source repositories. Due to the complexity of version iterations and the diversity of update intentions, such datasets are more prone to incorporating noisy data[19, 28, 34, 41]. To evaluate the quality of updating-related datasets and remove noises from them, we consider the need for updating scenarios and propose CupCleaner. The experimental results show that CupCleaner is effective in removing noises from comment updating datasets.

7.2 Data Cleaning

Data quality is crucial for deep learning models, as only high-quality datasets can comprehensively evaluate the capabilities of deep learning techniques. In recent years, researchers in the field of AI for SE also gradually focus on how to improve the quality of code-related data. *Sun et al.* [32] conduct an empirical study on the dataset for code search task and find that more than one-third of the queries contain noises. To improve the quality of code search datasets, they propose a data cleaning framework by combining syntactic and semantic filters. The experimental results show that the performance of code search is significantly improved with the cleaned dataset. *Shi et al.* [30] focus on evaluation and data issues in code summarization tasks, and find that the pre-processing choices of the data have a significant impact on the experimental results. *Huang et al.*[9] analyze the issue of whether code needs to be commented or not. They find that the percentage of methods with both header or internal comments in software systems is low. Then they design an approach to determining whether code needs to be commented by considering structural features, syntactic features, and textual features. *Shi et al.* [31] subsequently analyze the noisy data in code comments and improve the quality of code comments. They propose an automatic code-comment cleaning tool and conduct experiments on four code summarization datasets. Different from this approach that focus on cleaning individual comment or code, our proposed CupCleaner focuses on the changes between old and new comments/code and filters out data that deviates from the updating intention. CupCleaner leverages the semantic and diff information of code and comments, rather

than relying on statistical analysis of basic textual information. Additionally, CupCleaner and approaches that solely focus on cleaning code or comments are complementary, as the latter can still benefit from applying our approach.

8 CONCLUSION

In this paper, we propose CupCleaner, a data cleaning approach for comment updating datasets. The data cleaning strategy of CupCleaner mainly consists of two steps. First, we design a criterion to calculate the quality scores for all data samples, and then we remove the tail of the score distribution. During the scoring process, we consider the correlations within the comments or code, as well as the correlation between code changes and comment changes. In the experiments to evaluate CupCleaner, we first conduct human evaluation, and statistic results suggest that the data identified as noise by CupCleaner received generally low scores. Then, we conduct experiments on three comment updating datasets and three representative code-related models to evaluate the effectiveness of the CupCleaner. The experimental results show that CupCleaner can effectively filter out noisy data and improve the performance of the models without changing the test set. Additionally, we also explore whether the cleaned test set can be used to evaluate models. We find that the trend of the evaluation results on the cleaned test set is consistent with that of the original test set and there is an improvement in performance. Therefore, it may reflect the true ability of the model and be helpful in constructing a more reasonable dataset. We also find that the time consumption of CupCleaner for data cleaning is generally acceptable compared with the time for training.

In future work, we plan to explore other datasets for various software engineering tasks, and expand our data cleaning approach.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721.

[3] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[6] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the removal of obsolete TODO comments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 218–229.

[7] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[8] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[9] Yuan Huang, Nan Jia, Junhuai Shu, Xinyu Hu, Xiangping Chen, and Qiang Zhou. 2020. Does your code need comment? *Software: Practice and Experience* 50, 3 (2020), 227–245.

[10] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[11] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 298–309.

[12] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[13] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. CodeReviewer: Pre-Training for Automating Code Review Activities. *arXiv preprint arXiv:2203.09095* (2022).

[14] Qingyuan Liang, Zeyu Sun, Qihao Zhu, Wenjie Zhang, Lian Yu, Yingfei Xiong, and Lu Zhang. 2021. Lyra: A Benchmark for Turducken-Style Code Generation. *arXiv preprint arXiv:2108.12144* (2021).

[15] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[16] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 585–597.

[17] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.

[18] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[19] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating github for engineered software projects. *Empirical Software Engineering* 22 (2017), 3219–3253.

[20] Courtney Napoles, Keisuke Sakaguchi, Matt Post, and Joel Tetreault. 2015. Ground truth for grammatical error correction metrics. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. 588–593.

[21] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. 2021. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 427–435.

[22] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond J Mooney. 2020. Learning to update natural language comments based on code changes. *arXiv preprint arXiv:2004.12169* (2020).

[23] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *ACL* (Philadelphia, Pennsylvania). 311–318. https://doi.org/10.3115/1073083.1073135

[24] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[25] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[26] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[27] Nils Reimers and Iryna Gurevych. 2020. Making Monolingual Sentence Embeddings Multilingual using Knowledge Distillation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://arxiv.org/abs/2004.09813

[28] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E. Papalexakis, and Michalis Faloutsos. 2020. SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 149–163. https://www.usenix.org/conference/raid2020/presentation/omar

[29] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 181–190.

[30] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*. 1597–1608.

[31] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–119.

[32] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*. 1609–1620.

[33] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[34] Bogdan Vasilescu, Alexander Serebrenik, and Vladimir Filkov. 2015. A data set for social diversity studies of GitHub teams. In *2015 IEEE/ACM 12th working conference on mining software repositories*. IEEE, 514–517.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[36] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[37] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering*. 1–11.

[38] Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. 2018. Learning to synthesize. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*. 37–44.

[39] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 416–426.

[40] Wei Xu, Courtney Napoles, Ellie Pavlick, Quanze Chen, and Chris Callison-Burch. 2016. Optimizing statistical machine translation for text simplification. *Transactions of the Association for Computational Linguistics* 4 (2016), 401–415.

[41] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*. 476–486.

[42] Daochen Zha, Zaid Pervaiz Bhat, Kwei-Herng Lai, Fan Yang, Zhimeng Jiang, Shaochen Zhong, and Xia Hu. 2023. Data-centric artificial intelligence: A survey. *arXiv preprint arXiv:2303.10158* (2023).

[43] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. CoditT5: Pretraining for Source Code and Natural Language Editing. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[44] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009