algorithms

MDPI

Article

Investigating Novice Developers' Code Commenting Trends Using Machine Learning Techniques

Tahira Niazi [1], Teerath Das [2], Ghufran Ahmed [3], Syed Muhammad Waqas [4], Sumra Khan [1], Suleman Khan [5,*], Ahmed Abdelaziz Abdelatif [6] and Shaukat Wasi [1]

[1] Department of Computer Science, Mohammad Ali Jinnah University, Karachi 75400, Pakistan
[2] Faculty of Information Technology, University of Jyväskylä, 40014 Jyväskylä, Finland
[3] School of Computing, National University of Computer Emerging Sciences, Karachi 75400, Pakistan
[4] Department of Computer Science, Bahria University, Karachi 75260, Pakistan
[5] School of Psychology and Computer Science, University of Central Lancashire, Preston PR1 2HE, UK
[6] Khawarizmi International College, Al Bahya, Abu Dhabi 25669, United Arab Emirates
* Correspondence: skhan92@uclan.ac.uk

Abstract: Code comments are considered an efficient way to document the functionality of a particular block of code. Code commenting is a common practice among developers to explain the purpose of the code in order to improve code comprehension and readability. Researchers investigated the effect of code comments on software development tasks and demonstrated the use of comments in several ways, including maintenance, reusability, bug detection, etc. Given the importance of code comments, it becomes vital for novice developers to brush up on their code commenting skills. In this study, we initially investigated what types of comments novice students document in their source code and further categorized those comments using a machine learning approach. The work involves the initial manual classification of code comments and then building a machine learning model to classify student code comments automatically. The findings of our study revealed that novice developers/students' comments are mainly related to *Literal* (26.66%) and *Insufficient* (26.66%). Further, we proposed and extended the taxonomy of such source code comments by adding a few more categories, i.e., *License* (5.18%), *Profile* (4.80%), *Irrelevant* (4.80%), *Commented Code* (4.44%), *Autogenerated* (1.48%), and *Improper* (1.10%). Moreover, we assessed our approach with three different machine-learning classifiers. Our implementation of machine learning models found that *Decision Tree* resulted in the overall highest accuracy, i.e., 85%. This study helps in predicting the type of code comments for a novice developer using a machine learning approach that can be implemented to generate automated feedback for students, thus saving teachers time for manual one-on-one feedback, which is a time-consuming activity.

Keywords: source code comments; classification; machine learning techniques

Citation: Niazi, T.; Das, T.; Ahmed, G.; Waqas, S.M.; Khan, S.; Khan, S.; Abdelatif, A.A.; Wasi, S. Investigating Novice Developers' Code Commenting Trends Using Machine Learning Techniques. *Algorithms* **2023**, *16*, 53. https://doi.org/10.3390/a16010053

Academic Editors: Xiang Zhang and Xiaoxiao Li

Received: 12 October 2022
Revised: 2 January 2023
Accepted: 4 January 2023
Published: 12 January 2023

1. Introduction

Code comments are considered an integral and indispensable activity across various tasks in the software development life cycle (SDLC). Indeed, it is necessary for the developers and peer reviewers to understand what the code is intended to perform and how it works. In recent times, with the increase in software complexity and the number of developers working on a single project, it has become necessary to write code comments to make any sense of what is happening within the software. With growing team sizes, it is important for all the developers in the team to have a better understanding of the code. This can be achieved by adhering to good programming conventions to better understand the codebase by all the developers within a team. Many code conventions are followed to make the code readable across the development teams, e.g., naming conventions, source code comments, etc. Code conventions are a set of formats, rules, and guidelines followed while writing code. The code is written in a specific format to make the program easy to

read and understand. There are many research studies that discuss the impact of adopting coding standards on their particular projects [1]. Many studies reveal that coding conventions significantly and positively impact the readability of code [2]. These programming conventions help developers produce more readable code that is better understood by others. At the same time, it also helps to produce adaptable code, which is easy to fix when it comes to bug fixing. Basic code conventions are generalized across all programming languages. However, the way code comments are written varies according to the programming language's syntax. Figure 1 illustrates different types of comments in Java program such as block, single-line and multi-line comments.

As discussed above, source code commenting is one of the programming practices widely followed by developers to explain their code to others, who intend to gain an understanding for either improving the code or bug fixing. For a particular project, the same development team is not always the one to work on that project continuously. Therefore, it is not guaranteed that the next developer will be as experienced as the development team that worked on it before. No matter how well the code is written or refactored, it still requires that the documentation be included in the source code, and therefore, code commenting is one of the good practices for documenting the code.

Code comments are part of the source code that developers produce in natural language to describe the purpose and implementation details of the programming code. The purpose of source code comments is not just limited to the code's explanation; developers also include comments to highlight any pending bug fixes, technical debt, or references to other code snippets. They play a pivotal role in many software engineering tasks, such as software maintainability, code modification, and code reuse. Multiple researches suggest that commenting on a source code enhances its readability [3,4], leads to bug detection [5,6], and improves testing [7].

```
public class STSubscriptExpression extends STExpression {

    private static CSpellingService fInstance;

    /**
     * Returns the created expression, or null in case of error.
     * @deprecated Replaced by {@link #getExpression()}
     */
    @Deprecated
    public STExpression getSubscriptExpression(){
        if (fInstance == null) {
            fInstance = new Expression(ConsoleEditors.getPreferenceStore());
        }
        return fInstance;
    }

    /**
     * Handle terminated sub-launch
     * @param launch a terminable launch object.
     * @author Jesse MC Wilson
     */
    private void STLaunchTerminated(ILaunch launch) {
        // See com.vaadin.data.query.QueryDelegate#getPrimaryKeyColumns
        if (this == launch)
            return;
        // Remove sub launch, keeping the processes of the terminated launch to
        // show the association and to keep the console content accessible
        if (subLaunches.remove(launch) != null) {
            // terminate ourselves if this is the last sub launch
            if (subLaunches.size() == 0) {
                // TODO: Check the possibility to exclude it
                //monitor.exlude();
                monitor.subTask("Terminated"); //$NON-NLS-1$
                fTerminated = true;
                fireTerminate();
                // %%%
            }
        }
    }
}
```

Figure 1. Example of code comments in Java program file. Reprinted/adapted with permission from [8] Copyright 2019, by Luca Pascarella, Magiel Bruntink, Alberto Bacchelli.

This research study has provided the key contributions to taxonomy introduced in [9,10] by analyzing student code comments. Further, we implemented machine learning models to achieve the automated classification of students'/developers' source code comments.

The main contributions of our study are:

- An extension to the *taxonomy* of the source code comments introduced in [9,10];
- Automated *classification* of students'/developers' source code comments using machine learning techniques.

The remainder of this paper is structured as follows: The related research work is highlighted in Section 2 to find the significant gap. Section 3 describes the methodology exploited to conduct the research, and the experiment is explained in Section 4. The results and analysis of the research are reported in Section 5. Finally, Section 6 represents the conclusions and provides potential future research directions.

2. Related Work

Research on code comments has been an active area of research in the past decades. Many researchers have investigated code comments regarding their relation to code concerning various factors. These studies help us understand the effectiveness of code comments and their influence on different aspects of software design and implementation. The related literature is divided into six categories connected to our study: (i) code comments for code maintainability, (ii) code comments for bug detection, (iii) comments generation and code summarizing, (iv) code comments as a means of documentation, (v) code comments quality aspect and categorization, and (vi) analysis of student code comments.

2.1. Code Comments for Code Maintainability

Many researchers have studied the source code comments and revealed interesting findings that encourage programmers to follow this useful code convention. The benefits of good commenting extend beyond the primary benefit of providing information to the reader. Comments are an important element of code quality. They help document how the code is supposed to work. This increases programmer understanding, making the code more maintainable. Tenny et al. [3,4] suggested that commenting on a source code enhances its readability, as discussed in that leads to bug detection, discussed by Rubio-Gonz et al. and Subramanian et al. [5,6] and improved testing, discussed by Goffi et al. [7]. Hartzman et al. [11] studied the roles of comments in the maintenance of large software systems depicting the need for source code comments for maintainability. Jiang et al. [12] suggested that outdated comments that no longer align with the associated method entities result in confusion for the developers and hinder the process of future code-changing. As evident from the results, writing quality source code comments in a program is regarded as a good practice, as studied by de Souza et al. [13]. Oman et al. and Garcia et al. [14,15] introduced a quality metric called the code/comment ratio to quantify the quality of the overall code. Further tools are developed to assess the quality of the source code comments. For example, Khami [16] designed a tool called JavaDocMiner to check the quality of JavaDoc comments. It is based on natural language processing and evaluates the comment content concerning "language" and its relevance with the associated code. Steidl et al. [17] suggested that for analyzing the quality of code comments, a machine learning model was used, and assessment was carried out on various comment categories, including "header comments, member comments, in-line comments, section comments, code comments, and task comments." Similarly, as an extension to the previous work, Sun et al. [18] gave useful recommendations by performing a comprehensive assessment of the comments in jdk8.0 and jEdit.

2.2. Code Comments for Bug Detection

Many researchers have exploited code comments to gain useful insights for software quality assurance perspectives. The developers often overlook inconsistencies between code and comments as the codebase grows. Tan et al. [19] suggest that bugs can be

automatically detected between inconsistent code and comments. The experimental results present evidence that their tool, iComment, can extract 1832 rules from comments with 90.8–100% accuracy and detect 60 comment-code inconsistencies, 33 new bugs, and 27 bad comments in the latest versions of the four programs. Nineteen of these issues (twelve bugs and seven bad comments) were confirmed by the corresponding developers, while the other issues are currently under investigation by the developers. Ratol et al. [20] studied the process of refactoring a source code. Code comments can be used to facilitate the change introduced by the refactoring. Code comments were used to help in the refactoring activities, thus enhancing the code's maintainability. Few studies have been conducted to analyze GitHub commits; for example, Das et al. [21] analyzed GitHub commits to investigate the performance issues in Android application.

2.3. Comments Generation and Code Summarizing

Various studies have been conducted in the context of experimenting with comment generation and code summarizing to produce comments from the existing code. The techniques employed by machine translation were suggested by Allamanis et al. and Hu et al. [22,23], and information retrieval was suggested by Haiduc et al. and Huang et al. [24–26] to generate comments. The study by Lawrie et al. [27] employed an information retrieval approach using the cosine similarity for assessing the program's quality with the hypothesis that "if the code is high quality, then the comments give a good description of the code". Marcus et al. [28] introduced an innovative information retrieval approach to distinguish traceability links between source code and comments. Chen et al. [29] worked on automatically identifying the scope of the code comments in Java programs by employing machine learning techniques. However, they propose that natural language processing techniques can also be applied to evaluate the similarities between code comments and the corresponding code entities.

2.4. Code Comments as a Means of Documentation

In the literature, researchers also investigated the contents of comments in their work to further assess the need for writing informative and meaningful code comments. Hata et al. [30] investigated the role of links in code comments, their prevalence, purpose, and targets. Their investigation reveals diversity in the usage of links in comments, and links decay over time and evolve after they have been referenced in the source code comments. Similarly, Alghamdi et al. [31] studied comments concerning the presence of primitive data types through advanced lexical methods and demonstrated that developers document the primitive data types in the code comments to give additional information regarding purpose and usage.

2.5. Code Comments Quality Aspect and Categorization

As apparent from the above sections, code commenting practice varies among developers, and different code comments serve different purposes and meanings. Eventually, this leads to an interesting research area of comment classification. Some of the earliest studies by Haouari et al. and Steidl et al. [17,32] that worked on comment classification presented valuable results. Additionally, Zhai et al. [33] introduced a taxonomy by considering the code entities and the code perspectives of the comments. They also experimented with the propagation of comments from one code entity to another. However, classifying comments was not their primary purpose. Moreover, Pascarella et al. [8] introduced a more fine-grained taxonomy of code comments by studying comments from six open-source Java projects and mobile applications. It resulted in two-layered taxonomy having 6 top layers and 16 sub-layer categories. A statistically representative 1925 comments from files were selected and then manually classified by the two authors using the COMMEAN application. The authors used the supervised machine learning technique, probabilistic (Naïve Bayes Multinomial), and the Decision Tree algorithm (Random Forest or J48).

2.6. Analysis of Student Code Comments

Mohammadi-Aragh et al. [9] also assessed the commenting habits of students and categorized them into different types. Beck et al. [10] collected student source code comments and labeled them as "sufficient" or "insufficient" according to their codebook from their previous research work and then implemented supervised machine learning techniques. Their results suggest that introducing the lemmatization technique improved the performance of the Random Forest classifier. However, it lowered the performance on Multinomial Naïve Bayes on average. Additionally, Random Forest exceeded Naïve Bayes classifier in both testing rounds based on the results. Vieira et al. [34] worked on promoting in-code comments to self-explain the code written by students. Beck et al. [35] studied the structure and proportion of student comments and code.

Furthermore, various studies have been conducted to build the taxonomy of source code comments. For example, Table 1 contains the various aspects along with their names and descriptions. It presents the aspects that were considered in various types of research work on source code comments carried out in this particular domain. Table 2 is an overview of the research studies that cover specific aspects of Table 1.

Table 1. Aspects considered in research.

Aspect Category	Aspect Name	Description
A1	Analysis of student code commenting habit	Whether the dataset was taken of the professional developers or students
A2	Taxonomy based on code cognition	Types of comment categories based on comments insights from the author re- flection
A3	Taxonomy based on program aspect	Types of comment categories based on program structure and related code entities
A4	Classification using machine learning method	Any classification techniques that are applied to carry out the research

Table 2. Analysis of existing research (A = Addressed, NA = Not Addressed).

Research Work	A1	A2	A3	A4
J. Zhai et al. [33]	NA	NA	A	A
L. Pascarella and A. Bacchelli, [8]	NA	NA	A	A
P. Beck et al. [10]	A	A	NA	A
L. Pascarella [36]	NA	NA	A	A
R. E. Garcia [34]	NA	NA	NA	NA
M. J. Mohammadi-Aragh et al. [9]	A	A	NA	NA
H. Hata, C. Treude et al. [30]	NA	NA	A	NA
M. Alghamdi et al. [31]	NA	NA	A	NA
P. J. Beck [35]	A	NA	NA	A

The research study discussed herein mainly differs from the existing research work in all the above aspects. In particular, previous studies were mostly based on codebases produced by professional developers, whereas the current study investigates the code commenting habits of novice developers. The work by Mohammadi-Aragh et al. [9] is also related to students' commenting habits, but the experiment was carried out for the Python language; however, this research work has taken Java as the programming language. To illustrate this point, consider the fact that programming languages differ in their structure, as the former is a dynamically typed language and the latter is a statically typed language, therefore, having a consequent impact on the programming concepts. Moreover, the research work mentioned above used supervised machine learning methods to train a binary classifier. In contrast, our model is capable of classifying data into different categories, i.e., a multi-class classification model. Another aspect is a difference in the granularity of the classification with respect to comment categorization.

3. Methodology

The study aims to analyze the activities performed in the Java source code with the purpose of manually investigating and classifying the source code activities using machine learning techniques. The study was conducted from the viewpoint of novice developers and researchers. The context of the study is based on the projects of novice students/developers that were developed at Mohammad Ali Jinnah University.

3.1. Research Questions

The main objectives that drive the motivation behind this study are: is it possible to analyze the code activities by novice developers and further classify the source code comments that would help novice student developers to write meaningful code comments? The intent is to make their code more readable. We formulated two research questions (RQs) to investigate this study further.

RQ-1. *Which kind of code activities are performed by novice students/developers in the source code?*

Rationale: The primary rationale behind this research question is to analyze the key activities developers mention in their source code comments. This research question provides an idea to novice developers regarding the essential aspects that should be considered for development. The features exploited in this ***RQ-1*** are *comment_content* and *code*, as shown in Table 3. As already described, the main idea is to see the novice developers' activities by analyzing comment code and its corresponding source code and building a meaningful set of categories. The outcome of this research question will be a taxonomy of categories mentioned in the source code.

Table 3. Description of the features extracted for the dataset preparation.

No.	Feature Name	Description
1	comment_content	This feature contains the comment text written by the student
2	code	This feature contains the relevant code about which the comment was written.
3	begin_line	The line number of the file at which the comment begins.
4	end_line	The end line of the comment.
5	code_start_line	The start line for the relevant code section.
6	type	The type of the comment, i.e., single-line or multi-line.
7	category	This is the class that was labeled to the dataset using the taxonomy.

RQ-2. *Is it possible to classify novice students'/developers' source code comments using machine learning techniques*?

Rationale: This research question is dedicated to automatically classifying the novice students'/developers' source code comment categories obtained in ***RQ-1*** using machine learning techniques. Furthermore, the objective of this research question is to apply different machine-learning approaches to source code comments and eventually find the best machine-learning approach for classifying code comments. This will help novice developers to categorize the new comments in the correct categories.

It is important to note here that the work examines student code comments with a finer categorization of their comments, as discussed at the end of Section 2.6. The objective of this research question is also to lay out the methodology of machine learning techniques based on multi-label classification, which will provide an outcome of how effective this approach is at predicting code comments for novice developers.

The proposed method in Listing 1 represents the high-level pseudo-code algorithm, which describes the overall methodology. The methodology of our study consists of two parts: (i) Preparation of taxonomy of code comments using source code comments (M1), and (ii) classification of source code comments categories using machine learning techniques (M2). *M1* aims to address ***RQ-1***, whereas *M2* is dedicated to ***RQ-2***, described in the research question section. For the ***RQ-2*** methodology, we used three machine learning

techniques to classify the source code's comments: (i) Support Vector Machine, (ii) Random Forest, and (iii) Decision Tree.

Listing 1. Pseudo code of classification of code comments.

```
1
2    Input: Comments extracted from the source code
3    Parameter: Hyperparamter tuning
4    Output: Label comments
5    Steps of M1: Preparation of dataset of source code comments
6    building taxonomy of source code comments
7
8    1. Pre-processing the raw java source code.
9    2. Building the parser to parse the raw source code and extract code comments in JSON object
format
10   3. The JSON object is converted into CSV
11   4. Building Taxonomy and Dataset Annotation
12
13   Steps of M2: Classification of source code comments categories using machine learning
techniques
14
15   5. Extract the dependent feature.
16   6. Add the dependent feature to the original dataset.
17   7. Split the dataset into training and testing
18   8. Hyperparameter Tuning
```

3.2. Context Selection

The context of this study is 70 web development projects written in Java language by novice students/developers at the Department of Computer Science, Mohammad Ali Jinnah University. We built the dataset of source code comments from the Java code projects, and these comments are extracted by the *parser*, which is built in JavaScript. There are two approaches used in Java programming to comment in the source code, i.e., (i) single-line comment and (ii) multi-line comment.

Single-line comments start with//.
Example System.out.println("Hello World");//This is the example comment
Multi-line comments start with/* and end with */.
Example://* The code will print the character of words to the screen, then use it in line 4*/

Figure 2 represents the flow chart of the classification of code comments that describes the overall flow of our algorithm. The raw Java code is initially pre-processed, and then we parse it to create a JSON object, which is then converted to CSV. In CSV, we obtain our dataset, which consists of 5000 total comments, and then annotate it using our designed taxonomy, represented in Table 4. We then extract the dependent features, and add those features to our dataset. The dataset was then split into training and testing to apply the machine learning algorithm and tune their hyper-parameters to improve the results even further.

Table 4. Taxonomy and annotation with new categories.

SNO	Category	Frequency	Description
1	Literal	1333 (26.66%)	A comment that just restates the source code and does not provide any additional insight into the program's logic.
2	Insufficient	1333 (26.66%)	Code comments might be classified as "insufficient," either if they do not provide enough information for understanding the code or if, even if they are verbose, they add no value.
3	Conceptual	1111 (22.22%)	Conceptual comments explain source code functionality without simply restating the source code. Conceptual comments are not mere translations of source code in English but explain its functionality in greater detail to a code reviewer or another outside developer.

Table 4. *Cont.*

SNO	Category	Frequency	Description
4	**License**	**259 (5.18%)**	**The code comments that contain information on the terms of use and the licensing of the source code.**
5	**Profile**	**240 (4.80%)**	**These code comments provide references to the authors and their ownership of the work, as well as source credentials in the form of an "@author" tag.**
6	**Irrelevant**	**240 (4.80%)**	**The type of code comments for which it is not easy to comprehend their meaning and they do not clearly describe the associated code and are not related.**
7	**Commented code**	**222 (4.44%)**	**This category includes all comments that contain source code commented out by developers.**
8	Organizational	92 (1.84%)	Organizational comments are used to communicate the structure of code. They typically take the form of a short comment that explains a module or block of code, separating one functional unit from another. This demonstrates that the programmer is attempting to present code in a way that helps other coders easily understand it.
9	**Autogenerated**	**74 (1.48%)**	**This category includes Auto-generated code. These are typically the metadata left behind by an IDE and contain only the skeleton with a placeholder provided by the IDE.**
10	**Improper**	**55 (1.10%)**	**This category includes comments that are not properly implemented, e.g., a comment should have an associated code directly below the comment body without any empty lines in between.**
11	Empty	41 (0.82%)	This category includes the comments that do not contain anything, for example //

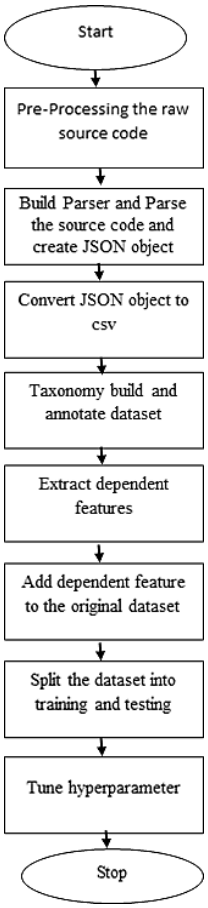

Figure 2. Flow chart of classification of code comments.

3.3. Data Extraction

As already discussed, we split the methodology for ***RQ-1*** and ***RQ-2*** as *M1* and *M2*, respectively. The data extraction process of the research questions is described in *M1* and *M2*.

3.3.1. M1: Preparation of Dataset of Source Code Comments and Building Taxonomy of Source Code Comments

As shown in Figure 3 and Listing 1, the steps from 1 to 4 of the figure and algorithm cover the data extraction of ***RQ-1***. It defines the overall procedure of dataset preparation which is sub-divided into four parts, (i) pre-processing of java source code, (ii) building the parser, (iii) the JSON object is converted into CSV, and (iv) building taxonomy and dataset annotation, which are described as follows:

- **Pre-processing of java source code:** The dataset required to carry out this research was prepared by pre-processing the raw source code. This raw source code is obtained from the lab assignments of sophomore-year students in the computer science discipline. We only considered the java source code files for our study. Initially, we prepared a dedicated script to obtain all the projects that are: (i) complete projects and (ii) programs built in java language. This results in a total of 70 projects.
- **Build the parser and Parse the raw source code to create JSON object:** A parser was developed in JavaScript language to parse the java source code files and extract the code-comment pairs. The parser goes through a directory, traverses all sub-directories within that directory, and searches all files with a .java file extension. It reads the files one by one, extracts the code-comments pairs from those files, and creates a JSON object. That extended JSON object is later converted to CSV file format so that this can be used for machine learning experiments.
- **The JSON object converted into CSV:** Data pre-processing is the first and most crucial phase in the research analysis. Data pre-processing is applied to the CSV file that is obtained by the parser. Intensive pre-processing would be required to convert the raw code into a usable dataset.
- **Building Taxonomy and Dataset Annotation:** The generated dataset was carefully analyzed, the annotation for the dataset was performed from the existing taxonomy [9,10], and new categories in the taxonomy were also introduced. Table 4 below contains all the information about the taxonomy, which consists of the comment type name and its description (the types in bold text are newly introduced types in the taxonomy).

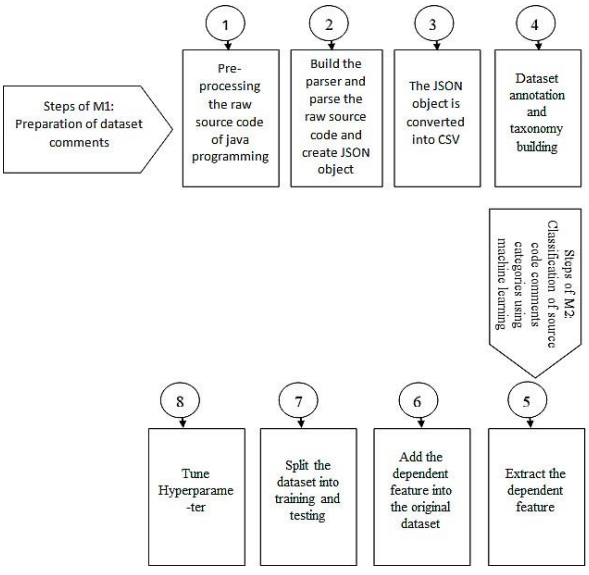

Figure 3. Overall methodology.

Figure 3 depicts the overall methodology, which consists of two steps (M1 and M2), each represented by an arrow symbol and each sub-step by a number. The first step, which we called M1, was the preparation of the dataset represented by 1–4. In substep 1, the raw source code is pre-processed. In the second sub-step, the parser is built through which the raw source code is parsed to create a JSON object containing the source code com-

ments. In the third sub-step, the JSON object is converted into CSV. In the fourth sub-step, a taxonomy is built, and the annotation is performed on a CSV file. The second step, M2, involves the classification of source code using various machine learning algorithms; the second part is represented as 5–8. In the fifth sub-step, feature extraction is performed, and in the sixth sub-step, we divided the dataset into training and testing. Then, in the seventh sub-step, a machine learning classification model was implemented, and in the last sub-step, three evaluation measures were used to evaluate the performance of the algorithm.

3.3.2. M2: Classification of Source Code Comments Categories Using Machine Learning Techniques

The methodology (M2) covers points from 5 to 8 in the algorithm and aims to answer ***RQ-2***. *M2* consists of 4 steps, i.e., (i) extract the dependent features from data, (ii) add the extracted features to the original dataset, (iii) split the dataset into training and testing, and (iv) tune the hyper-parameter for the machine learning model. We further explain these steps in detail in the next Experiment section.

4. Experiment

Several experiments are performed to assess the effectiveness of the suggested method. A machine equipped with an Intel Xeon E5-2630 v4 CPU, 64G RAM, Windows 10 with 64-bit OS, and the Jupyter notebook was used to conduct the tests. We exploited precision, recall, and accuracy as the performance metrics to report the results.

4.1. Characteristics of Datasets

In order to run machine learning experiments, it is important to have datasets available for these experiments. Often, the data required for an experiment are not readily available in the desired format. Therefore, it must be created from scratch. Similarly, this was the case in our experiment, and we had to prepare our datasets. A total of 5000 samples were acquired in the dataset preparation process. The information contained in Table 3 describes each feature/attribute present in our dataset.

4.2. Performing Feature Extraction

The feature engineering step was carried out on the data, and some new features were created from the existing data. This step is also called feature extraction. These new features were extracted to improve the machine learning results and enrich the feature set. Table 5 describes the new features that resulted after the feature engineering step.

Table 5. New features introduced in the dataset during feature engineering process.

No.	Feature Name	Description
1	comment_length	The number of characters present in the comment content.
2	is_license	Indicates whether the comment falls in the License category with a value of 1 or 0 (1: yes, 0: no)
3	comment_token_length	The number of tokens present in the comment content.
4	is_profile	Indicates whether the comment falls in the Profile category with a value of 1 or 0 (1: yes, 0: no)

4.3. Training and Testing the Model

After feature extraction, the dataset is split into the training and test set. The training set is used to train the classifier with the help of hyperparameter tuning, and later the model was evaluated according to accuracy and other parameters by using the test set for predictions.

Code comments written by students are often too general or premature and lack the precision of those written by more experienced developers. From the code perspective, the apparent characteristics of the code comments can be extracted as the features, e.g., what is the text that creates a comment, from where does the comment start, etc. The features listed in Table 3 were extracted based on these apparent characteristics of comments to

prepare the initial dataset for this experiment. The initial dataset was examined, and a feature engineering process was performed to derive more meaningful features that can enhance the overall prediction of the machine learning model. The impact of features from Table 5 is demonstrated in the Result and Discussion section.

4.4. Hyperparameter Tuning

Hyperparameters are the adjustable parameters of a machine learning model architecture, and these cannot be assigned randomly, but rather optimizing and selecting the ideal parameters is needed to improve the machine learning model's performance. Therefore, this process is called hyperparameter tuning. We performed the hyperparameter tuning of the machine learning model in our experiments to enhance their overall performance. K-fold cross-validation of 5 folds was applied to tune the machine learning model and gain the values for the hyperparameters.

Among the three classifiers, Decision Tree was the most accurate classifier for predicting the comment categories, and then Random Forest also gave relatively better results. However, Support Vector Machine only performed well on the textual data; therefore, we performed hyperparameter tuning on the Random Forest to further improve its results. As discussed above, the 5-fold cross-validation method was used to obtain the values for the hyperparameters. We employed *GridSearchCV* from the *Scikit-learn* library; it is a method that exhaustively considers all parameter combinations (as shown in Listing 2), providing a means for finding the best parameter settings.

Listing 2. Parameter combinations for the hyperparmater tuning of Random Forest classifier.

```
param_grid_rf = {
    ' n_estimator ': [200, 300, 400, 500, 600, 700,  800,  1000],
     max_features': [' auto '],
    ' max_depth': [20, 22, 24, 26, 28, 30],
    ' random_state': [ x for x in range (6,100,2)]
}
```

4.5. Evaluation and Performance Metrics

Every experiment evaluation requires some performance measure to validate the results. We used four performance measures in our study, i.e., *accuracy, precision, recall,* and *F*1-*Scores*. The main reasons for selecting these measures are their widespread acceptance in machine learning. These performance measures are obtained from the classification report of the machine learning model.

Model accuracy measures how accurately a classification model predicts classifications. A model's accuracy is defined by the number of correct classifications divided by the number of total predictions.

$$Accuracy = \frac{Total\ Number\ of\ Correct\ reponses\ for\ a\ class}{Total\ Number\ of\ responses\ for\ a\ class}$$

The precision measure is also known as *positive predictive value (PPV)*, which indicates the proportion of positive instances among all the positive class predictions. This prediction measure is defined for each class output individually. The outcome is different for each parameter.

$$Precision_{(class)} = \frac{Total\ Number\ of\ Correct\ predictions\ for\ a\ class}{Total\ Number\ of\ resulting\ predictions\ for\ a\ class}$$

The *recall* tells how many of all the positive samples were correctly identified as such by the classifier. It is also known as *true positive rate (TPR)* or sensitivity. Recall measure is also defined for each class output individually.

$$Recall_{(class)} = \frac{Total\ Number\ of\ Correct\ predictions\ for\ a\ class}{Total\ Number\ of\ Actual\ predictions\ for\ a\ class}$$

By calculating the harmonic mean of a classifier's precision and recall, the *F1-score* integrates both into a single metric.

$$F1 - Score_{(class)} = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

5. Results and Discussion

5.1. RQ-1 Which Kind of Code Activities Are Performed by Novice Students/Developers in the Source Code?

In order to answer this research question, we manually analyzed each comment's content and assigned them a suitable category. The category should be a representation of the whole source code comment. Master's students performed this activity for all the source code comments. The student's supervisor and co-supervisor cross-checked the labels to verify this activity. The outcome of this labeling activity was a taxonomy of categories representing different aspects of the source code.

According to our results, the source code comments are mainly distributed in the categories such as *Literal, Insufficient, Conceptual, License, Profile, Irrelevant, Commented code, Autogenerated, Improper, and Empty*. The distribution of the dataset according to the "category" is reported in Table 4 and Figure 4. We observed that the more frequent source code comment categories in our dataset are *"Insufficient" and "Literal"* (1333, 26.66% each). This may be due to students' inability to write better comments as the large proportion of comments falls in the "Insufficient" and "Literal" categories. The *Literal* category just restates the source code and does not provide any additional insight into the program logic. Whereas in the *Insufficient* category, comments either do not provide enough information for understanding or are verbose, i.e., adding no value. Moreover, large numbers of *conceptual* comments (1111, 22.22%) are present in our dataset, followed by *License* (259, 5.18%). This makes sense because, generally, students write some *conceptual* comments to explain the functionality of source code in detail without simply restating the code. *License* comments are more focused on the terms of use and the licensing of the source code. It usually appears at the top of the source code.

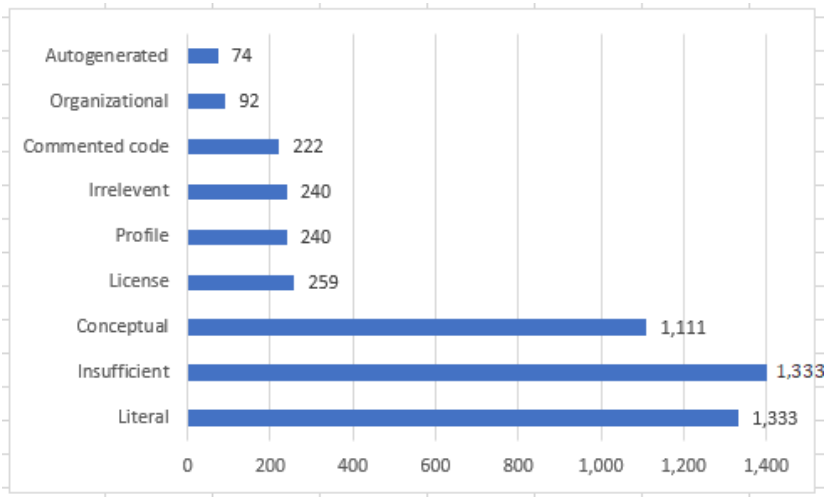

Figure 4. Distribution of code-comments data by the type column feature.

Empty (41, 0.82%) category source code comments are rare in our dataset. We presume that developers very rarely write "//" or does not write any comment on the source code.

Furthermore, as discussed, Table 4 depicts the complete taxonomy that is obtained after pre-processing, carefully analyzing, and annotation of the source code comments. The *bold* category type, frequency, and description are indicated as our *contribution* as new categories to the existing taxonomy. The new categories of the taxonomy that emerged from our dataset are:

- **License (259, 5.18%):** License is the code comments that contain information on the terms of use and the licensing of the source code. It usually appears at the top of the source code.
- **Profile (240,4.80%):** This comment contains the information of authors and the ownership of the work; it usually begins with the "@author" tag.
- **Irrelevant (240, 4.80%):** It is not easy to comprehend the meaning of the comment. This type of comment does not describe the associated code.
- **Commented code (222, 4.44%):** This category contains all the comments that contain source code which were commented out by the developer.
- **Auto-generated (74, 1.48%):** This category contains metadata left behind by an IDE and contains only the skeleton with a placeholder provided by the IDE.
- **Improper (55, 1.10%):** This category includes comments that are not properly implemented, e.g., a comment should have an associated code directly below the comment body without any empty lines in between.

From the above results, these obtained categories can be used as a checklist for novice developers to check what types of comments developers focus on during software development. It is interesting to note that novice developers frequently mention the *Literal* type of comments in their source code. This is reasonable because the comments of novice developers are more specific to what they are implementing in their source code. Moreover, the comments type *Empty* has been used significantly less by novice developers, which means developers tend to comment on what activities are performed in source code.

Distribution of Data According to Type of Comments

We further analyzed that there are two types of comments present in our dataset: (i) single-line comments and (ii) multi-line comments. The data distribution according to the "type" column is shown in Table 6, along with its graphical representation in Figure 5. It is evident from the figure that students are more comfortable writing single-line comments than writing more detailed comments in the form of multi-line. We attribute this to the fact that students are still in the learning phase and lack attention to the code documentation aspect.

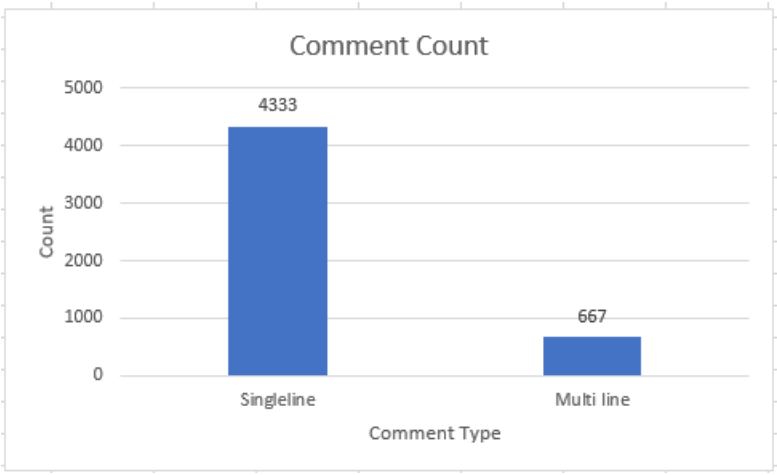

Figure 5. Distribution of code-comments data by the type column feature.

Table 6. Distribution of data by "type" column.

Type	Total
Single-line	4333
Multi-line	667

After the dataset and taxonomy preparation, the experiments were carried out in Python using the Jupyter notebook. Mainly, three machine learning algorithms were implemented to conduct the experiment: *Support Vector Machine, Random Forest, and Decision Trees*. Table 7 reports the results of three models for their accuracy measure with a comparison of the after and before feature extraction steps. The results show that all three classifiers' performance was enhanced after implementing feature extraction.

Table 7. Accuracy measure of proposed machine learning algorithm.

Method	Before Feature Extraction	After Feature Extraction
Random Forest	0.68 (68%)	0.84 (84%)
Decision Tree	0.72 (72%)	0.85 (85%)
Support Vector Machine	0.31 (31%)	0.59 (59%)

5.2. RQ-2 Is It Possible to Classify Student Source Code Comments Using Machine Learning Techniques?

In order to answer this research question, we applied three machine learning algorithms to our labeled dataset obtained in ***RQ-1***. The algorithms considered in our study to classify the source code comments are: (i) *Support Vector Machine (SVM)*, (ii) *Decision Tree*, and (iii) *Random Forest*. After the machine learning models were implemented, the results were recorded. We analyzed the results with three performance parameters, i.e., *accuracy, precision, and recall*.

Table 7 and Figure 6 represent the comparison of the *accuracy* measure for all three classifiers. The feature extraction helped in improving the overall performance of all three classifiers. As apparent from the results, the *Decision Tree* algorithm outperformed the rest of the two models with an overall accuracy of 0.85 (85%). The performance of the *Support Vector Machine* was poor. We presume that this is the one main reason it was not used previously for such a problem, i.e., classification of source code comments.

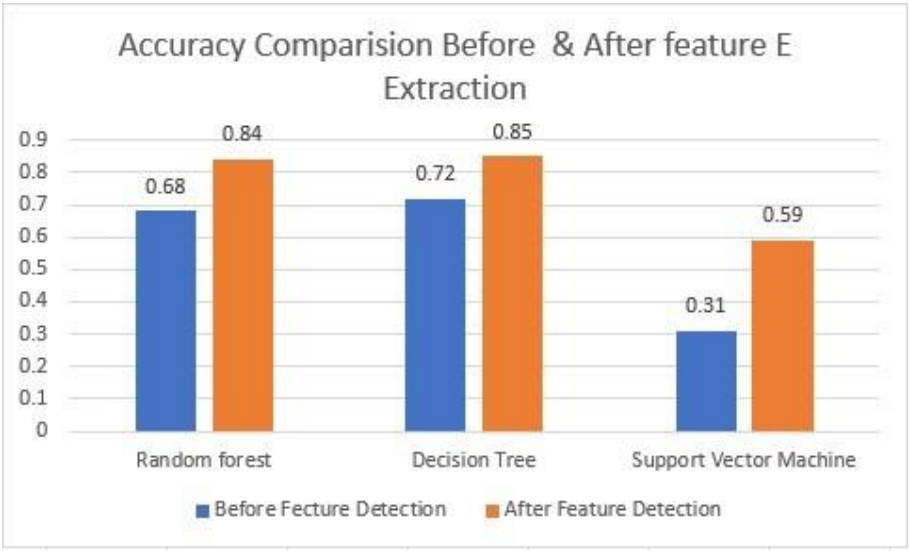

Figure 6. Classifiers accuracy before and after feature extraction.

Figure 6 shows a graphical representation of the accuracy before feature extraction and after feature extraction.

One interesting aspect to note from Table 8 and Figure 7 is that the *Support Vector Machine* classifier produced good results when only the text feature (i.e., comment content) was selected as the only predictor for the machine learning model. However, when both quantitative data (numerical information such as token comment size, length, etc.) and textual data (e.g., comment text) were used as predictors, both *Random Forest* and *Decision Tree* achieved good performance.

Table 8. Accuracy of all three classifiers on text data.

Method	Accuracy on Text Feature
Random Forest	0.83 (83%)
Decision Tree	0.86 (86%)
Support Vector Machine	0.83 (83%)

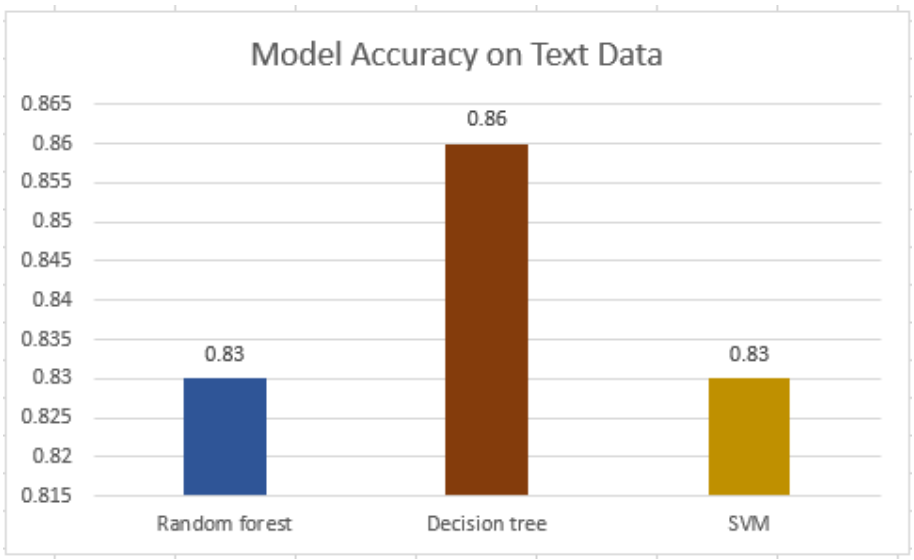

Figure 7. Accuracy of all three classifiers on text data only.

P. Beck et al. [10] evaluated and analyzed the code comments with a single label as either sufficient or insufficient using a binary classifier. In their research study, they only considered the text feature of the comments. They demonstrated that after the reduction in the vocabulary size due to lemmatization, the Multinomial Naive Bayes classifier's accuracy was reduced by 5%, but Random Forest Classifier's accuracy was improved by 6%. The results of their study reveal that they achieved an overall precision rate of 82% using Multinomial Naïve Bayes. By using a Random Forest classifier and lemmatization, they were able to achieve a classification precision of 90%. In another study, L. Pascarella [36] compared the performance of two machine learning models to automatically classify code comments in five open-source mobile applications. Their aim was to assess code comments produced by professional developers. Specifically, they used two well-known classes of supervised machine learning algorithms based on probabilistic classifiers and Decision Tree algorithms: Naive Bayes Multinomial and Random Forest. According to their results, Random Forest outperformed the Naive Bayes Multinomial classifier in automatically classifying the code comments. In our research study, we employed three different machine learning classifiers to compare their performance results. All three models produced better results on text data; however, when the other quantitative features were taken into account, both Random Forest and Decision Tree produced good results than the Support Vector Machine, with Decision Tree having the highest accuracy, i.e., 85%. It is interesting to note

that previous studies, as discussed above, also revealed the effectiveness of the Random Forest classifier for classifying source code comments.

Table 9 and Figure 8 represent the *precision*, *recall* and *F1-Score* of *Random Forest* and *Decision Tree* on all the source code comments categories of our dataset. The overall accuracy achieved in the case of *Random Forest* is 0.84 (84%), whereas the overall accuracy in the case of *Decision Tree* is 0.85 (85%), which means we obtain the best result on the Decision Tree.

Table 9. The Precision, Recall anf F1-Score for Random Forest and Decision Tree.

	Random Forest			**Decision Trees**		
	Precision	Recall	F1-score	Precision	Recall	F1-score
Autogenerated	1	0.5	0.67	1	0.5	0.67
Commented code	1	0.75	0.86	1	1	1
Conceptual	0.75	0.9	0.82	0.75	0.9	0.82
Improper	0	0	0	0	0	0
Insufficient	0.86	0.95	0.9	0.9	0.95	0.92
Irrelevant	1	0.67	0.8	0.67	0.67	0.67
License	1	1	1	1	1	1
Literal	0.78	0.74	0.76	0.82	0.74	0.78
Organizational	1	1	1	1	1	1
Profile	1	0.67	0.67	1	0.67	0.8
	Overall Accuracy 0.84			**Overall Accuracy** 0.85		

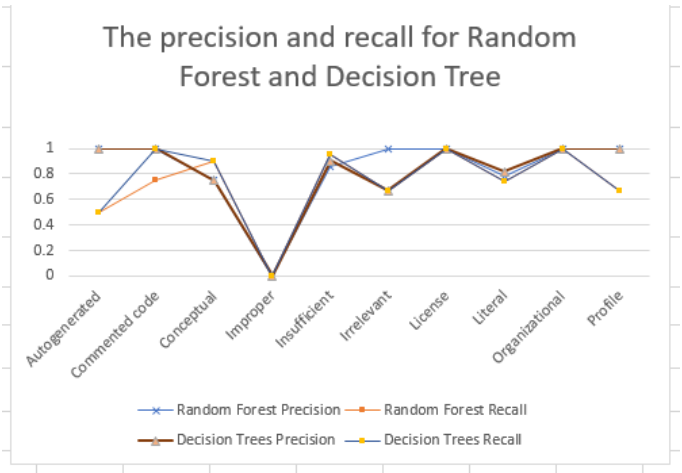

Figure 8. The precision and recall for Random Forest and Decision Tree.

6. Conclusions

In this study, initially, we manually classified the source code comments and then presented a machine-learning approach to classify source code comments written by novice developers/students enrolled at Mohammad Ali Jinnah University. This work is inspired by many aspects, such as student metacognition, focusing on internal student processes while writing code; teacher–student feedback activity, introducing automated feedback and reducing teacher dependency; and studying the machine learning approach to code-comment analysis. The results of our study depicted that novice developers/students' comments are mainly related to *Literal* (26.66%) and *Insufficient* (26.66%). Further, we proposed and extended a taxonomy of such source code comments by adding a few more categories, i.e., *License* (5.18%), *Profile* (4.80%), *Irrelevant* (4.80%), *Commented Code* (4.44%),

Autogenerated (1.48%), and *Improper* (1.10%). Moreover, after applying different machine learning algorithms, we found that the *Decision Tree* has the overall highest accuracy, i.e., 85%, and performed better than other studied techniques. Classification of source code comments is important from the perspective of how students utilize this important code convention in providing the documentation for their programming code. This study helps not only in predicting the type of code comments using the machine learning approach but also can serve as a basis for designing a utility that can be implemented to generate automated feedback for students, thus, saving teachers' time for manual one-on-one feedback, which is a time-consuming activity. The objectives of this study included building a source code parser, designing a taxonomy for the categorization of source code comments, and implementing the different machine learning models and their results and evaluations. The datasets for this research study were not available. Therefore, they were extracted from the raw source code of student programming tasks. The machine learning models performed classification with a reasonably good accuracy of 85%, achieved by *Decision Tree*, and hence, outperformed the other two algorithms.

7. Future Work

This study provides a foundation for future directions in this area of research. As previously discussed, the research in this software engineering domain is ongoing and offers much potential for further study. This also opens more ways for pursuing research in this field of study. In the future, other machine learning models can be analyzed by their performance. Furthermore, the NLP approach to classifying source code comments can be carried out and compared with the machine learning approach as a comparison. Moreover, in the future, the idea related to predicting the contribution of each feature to the classification model using Random Forest variables with SHAP can be implemented as an extension to the current work with anticipation of further enhancing the effectiveness and accuracy of our research objective. We also aim to incorporate the code context and CodeBert in our intended extension of this research effort.

Author Contributions: Conceptualization, T.N. and S.W.; Methodology, T.N., T.D. and S.M.W.; Software, T.N. and S.K. (Sumra Khan); Validation, T.D.; Investigation, T.D., S.K. (Sumra Khan) and S.W.; Resources, G.A.; Data curation, G.A. and S.W.; Writing—original draft, S.M.W.; Writing—review & editing, S.K. (Suleman Khan) and A.A.A.; Supervision, G.A. and S.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Smit, M.; Gergel, B.; Hoover, H.J.; Stroulia, E. Maintainability and source code conventions: An analysis of open source projects. *Univ. Alta. Dep. Comput. Sci. Tech. Rep. TR11* **2011**, 6.
2. dos Santos, R.M.; Gerosa, M.A. Impacts of coding practices on readability. In Proceedings of the 26th Conference on Program Comprehension, Gothenburg, Sweden, 27–28 May 2018; pp. 277–285.
3. Tenny, T. Program readability: Procedures versus comments. *IEEE Trans. Softw. Eng.* **1988**, *14*, 1271. [CrossRef]
4. Tenny, T. Procedures and comments vs. the banker's algorithm. *Acm Sigcse Bull.* **1985**, *17*, 44–53. [CrossRef]
5. Rubio-González, C.; Liblit, B. Expect the unexpected: Error code mismatches between documentation and the real world. In Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Toronto, ON, Canada, 5–6 June 2010; pp. 73–80.
6. Subramanian, S.; Inozemtseva, L.; Holmes, R. Live API documentation. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 643–652.
7. Goffi, A.; Gorla, A.; Ernst, M.D.; Pezzè, M. Automatic generation of oracles for exceptional behaviors. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; pp. 213–224.
8. Pascarella, L.; Bacchelli, A. Classifying code comments in Java open-source software systems. In Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), Buenos Aires, Argentina, 20–28 May 2017; pp. 227–237.

9. Mohammadi-Aragh, M.J.; Beck, P.J.; Barton, A.K.; Reese, D.; Jones, B.A.; Jankun-Kelly, M. Coding the coders: A qualitative investigation of students' commenting patterns. In Proceedings of the 2018 ASEE Annual Conference Exposition, Salt Lake City, UT, USA, 23–27 July 2018.
10. Beck, P.; Mohammadi-Aragh, M.J.; Archibald, C. An Initial Exploration of Machine Learning Techniques to Classify Source Code Comments in Real-time. In Proceedings of the 2019 ASEE Annual Conference & Exposition, Tampa, FL, USA, 15 June–19 October 2019.
11. Hartzman, C.S.; Austin, C.F. Maintenance productivity: Observations based on an experience in a large system environment. In Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering, Toronto, ON, Canada, 22–25 November 1993; Volume 1, pp. 138–170.
12. Jiang, Z.M.; Hassan, A.E. Examining the evolution of code comments in PostgreSQL. In Proceedings of the 2006 International Workshop on Mining Software Repositories, Shanghai, China, 22–23 May 2006; pp. 179–180.
13. de Souza, S.C.B.; Anquetil, N.; de Oliveira, K.M. A study of the documentation essential to software maintenance. In Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information, Coventry, UK, 21–23 September 2005; pp. 68–75.
14. Oman, P.; Hagemeister, J. Metrics for assessing a software system's maintainability. In Proceedings of the Conference on Software Maintenance 1992, IEEE Computer Society, Orlando, FL, USA, 9–12 November 1992; pp. 337–338.
15. Garcia, M.J.B.; Granja-Alvarez, J.C. Maintainability as a key factor in maintenance productivity: A case study. In Proceedings of the Icsm, Monterey, CA, USA, 4–8 November 1996; p. 87.
16. Khamis, N.; Witte, R.; Rilling, J. Automatic quality assessment of source code comments: The JavadocMiner. In Proceedings of the International Conference on Application of Natural Language to Information Systems, Cardiff, UK, 23–25 June 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 68–79.
17. Steidl, D.; Hummel, B.; Juergens, E. Quality analysis of source code comments. In Proceedings of the 2013 21st International Conference on Program Comprehension (icpc), San Francisco, CA, USA, 20–21 May 2013; pp. 83–92.
18. Sun, X.; Geng, Q.; Lo, D.; Duan, Y.; Liu, X.; Li, B. Code comment quality analysis and improvement recommendation: An automated approach. *Int. J. Softw. Eng. Knowl. Eng.* **2016**, *26*, 981–1000. [CrossRef]
19. Tan, L.; Yuan, D.; Krishna, G.; Zhou, Y. comment: Bugs or bad comments? In Proceedings of the ACM Symposium on Operating Systems Principles: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, New York, NY, USA, 3–6 November 2007; Volume 14, pp. 145–158.
20. Ratol, I.K.; Robillard, M.P. Detecting fragile comments. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana-Champaign, IL, USA, 30 October–3 November 2017; pp. 112–122.
21. Das, T.; Penta, M.D.; Malavolta, I. A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps. In Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, IEEE Computer Society, Raleigh, NC, USA, 2–7 October 2016; pp. 443–447. [CrossRef]
22. Allamanis, M.; Peng, H.; Sutton, C. A convolutional attention network for extreme summarization of source code. In Proceedings of the International Conference on Machine Learning, New York City, NY, USA, 19–24 June 2016; pp. 2091–2100.
23. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation. In Proceedings of the 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), Gothenburg, Sweden, 27 May–3 June 2018; pp. 200–20010.
24. Haiduc, S.; Aponte, J.; Marcus, A. Supporting program comprehension with source code summarization. In Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, South Africa, 1–8 May 2010; Volume 2, pp. 223–226.
25. Haiduc, S.; Aponte, J.; Moreno, L.; Marcus, A. On the use of automated text summarization techniques for summarizing source code. In Proceedings of the 2010 17th Working Conference on Reverse Engineering, Washington, DC, USA, 13–16 October 2010; pp. 35–44.
26. Huang, Y.; Zheng, Q.; Chen, X.; Xiong, Y.; Liu, Z.; Luo, X. Mining version control system for automatically generating commit comment. In Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, Canada, 9–10 November 2017; pp. 414–423.
27. Lawrie, D.J.; Feild, H.; Binkley, D. Leveraged quality assessment using information retrieval techniques. In Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06), Athens, Greece, 14–16 June 2006; pp. 149–158.
28. Marcus, A.; Maletic, J.I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In Proceedings of the 25th International Conference on Software Engineering, Portland, OR, USA, 3–10 May 2003; pp. 125–135.
29. Chen, H.; Huang, Y.; Liu, Z.; Chen, X.; Zhou, F.; Luo, X. Automatically detecting the scopes of source code comments. *J. Syst. Softw.* **2019**, *153*, 45–63. [CrossRef]
30. Hata, H.; Treude, C.; Kula, R.G.; Ishio, T. 9.6 million links in source code comments: Purpose, evolution, and decay. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 27 May 2019; pp. 1211–1221.
31. Alghamdi, M.; Hayashi, S.; Kobayashi, T.; Treude, C. Characterising the Knowledge about Primitive Variables in Java Code Comments. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 460–470.

32. Haouari, D.; Sahraoui, H.; Langlais, P. How good is your comment? A study of comments in java programs. In Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement, Banff, AB, Canada, 22–23 September 2011; pp. 137–146.
33. Zhai, J.; Xu, X.; Shi, Y.; Tao, G.; Pan, M.; Ma, S.; Xu, L.; Zhang, W.; Tan, L.; Zhang, X. CPC: Automatically classifying and propagating natural language comments via program analysis. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 27 June–19 July 2020; pp. 1359–1371.
34. Vieira, C.; Magana, A.J.; Falk, M.L.; Garcia, R.E. Writing in-code comments to self-explain in computational science and engineering education. *ACM Trans. Comput. Educ. (TOCE)* **2017**, *17*, 1–21. [CrossRef]
35. Beck, P.J.; Mohammadi-Aragh, M.J.; Archibald, C.; Jones, B.A.; Barton, A. Real-time metacognition feedback for introductory programming using machine learning. In Proceedings of the 2018 IEEE Frontiers in Education Conference (FIE), Lincoln, NE, USA, 13–16 October 2018; pp. 1–5.
36. Pascarella, L. Classifying code comments in Java mobile applications. In Proceedings of the 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Gothenburg, Sweden, 27 May–3 June 2018.