

A Machine Learning Approach to Analyze Patterns and Comments in Java Code using Python

KRITI SHARMA¹, MEENAKSHI GUPTA^{2*}, RINKY AHUJA², PRAWAR³, MINA KUMARI³

¹School of Engineering and Technology,
K. R. Mangalam University,
Gurugram, Haryana,
INDIA

²School of Engineering and Technology,
Sushant University,
Gurugram, Haryana,
INDIA

³School of Basic and Applied Science,
K. R. Mangalam University,
Gurugram, Haryana,
INDIA

**Corresponding Author*

Abstract: - This study introduces a detailed methodology to enhance code comprehension by examining comments and blank spaces in the program code. Through the analysis of comments within Java source code, developers can understand if the documentation coverage is efficient. The choice between single-line and multi-line comment type gives an understanding of how programmers chose to communicate information. Analysis of length of comment lines assess the level of details in the documentation. Blank spaces and blank-lines help in evaluating how well the code is structured. Together, these features help in improving the overall coding experience. The integration of the Naive Bayes algorithm refines the system's capabilities, enabling accurate comment classification and length assessment. In conclusion, the proposed system, bolstered by the Naive Bayes algorithm, presents a sophisticated approach to code comprehension and documentation enhancement in software development.

Key-Words: - Code comprehension, comments, blank spaces, Java class files, comment length, whitespace analysis, software development, Naive Bayes algorithm.

Received: August 28, 2024. Revised: May 26, 2025. Accepted: June 23, 2025. Published: August 7, 2025.

1 Introduction

Developers frequently engage in activities connected to program understanding when working on tasks linked to software maintenance and evolution, [1]. To properly comprehend a software system, developers usually rely on its documentation. Numerous studies have demonstrated that developers are more inclined to consult code comments than other types of documentation when seeking answers to queries about programming comprehension. The importance of "code documentation" as the main source of information for problem-solving, feature

implementation, communication, and code review has also been highlighted in recent research, [2].

It is crucial to provide class documentation, notably through code commenting, to make maintenance responsibilities for programs easier, [3]. Class comments give developers a wealth of knowledge about class usage and implementation, which can help other developers comprehend programs and carry out their responsibilities for software maintenance. This project aims to demonstrate the significance of comments and patterns in maintaining and comprehending Java code. Looking at the comment lines can reveal a lot about documentation and code organization. The

overall objective is to increase the quality and maintainability of the code by giving developers comprehensive analyses and insights into comment lines.

Python's natural language processing (NLP) tools, such as NLTK and spaCy, perform sentiment analysis, keyword extraction, and topic modeling from code comments, [4]. Java code commonly embodies design patterns like Singleton, Factory, and Observer, along with recurring idiomatic code snippets, [5]. Regular expressions and machine learning algorithms aid pattern discovery, while Python's data analysis capabilities support categorization and visualization, enhancing developers' understanding and reuse of patterns, [6]. Source code comments play a vital role in software by enhancing code comprehension and aiding in maintenance, [7], [8]. Prior research has examined comments from diverse angles, such as comment-to-code ratios, [9], comment-code co-evolution, [10], and comment purposes, [11]. A large-scale investigation into code comments, potentially aiding practices in aspects like defining comment density benchmarks, identifying optimal commenting points, and even generating code comments is conducted, [12], [13], [14], [15], [16].

2 Literature Survey

Intriguing efforts have been made to categorize source code comments, as demonstrated by [17], groundbreaking investigation. The study, [18], presented a system encompassing seven distinct comment categories, [19], focusing on macro-level attributes like module headers and method descriptions. This framework served as a basis for the extended work of [20], who introduced hierarchical categories. The objective involves curating pairs of code and comments for future applications. While established industry norms dictate macroscopic comments, [21], the style of local comments often remains a subjective matter, subject to the programmer's judgment. Consequently, the analysis of local comments presents challenges, with only limited attempts made in this domain. While code comments are valued for expressing programmers' intent, [22], the notion of "self-documenting code" is advocated by certain voices (Williams, 2001). Knowing that programmers turn to comments when other methods fail to convey intent, [23], conducted a study on source code comments within operating systems.

The study, [24], creatively detected issues linked to locks by extracting specialized patterns from code comments. The consequences of outdated

comments on defects are explored in [25]. The relationship between "TODO" comments and software bugs is examined in [26], whereas, [27], investigated the role of "TODO" comments as tools for communication among programmers. An approach to identify current "TODO" comments was presented by [28]. Additionally, [29], explored the potential for software bugs arising from commented-out code segments.

Using text filtering techniques, [30], looked at the significance of code comments. With their presentation of a framework that generates Java method descriptions through program analysis, the field of automatic comment generation is gaining popularity. Additionally, [31], presented a technique for producing comments using programming question websites like Stack Overflow. Categorizing programmer expertise levels has been explored in prior studies. [32], employed biometric sensor data, including an EEG-collecting 16-channel V-amp and an SMI RED-mx eye-tracker, to assess 38 novice and expert programmers. They employed Support Vector Machine models for EEG and eye-tracking data, achieving high F1 scores for expert classification. Similarly, [33], designed an artificial neural network to classify student performance in linear programming using the Linear Programming Intelligent Tutoring System (LP-ITS).

Diverse endeavors focus on extracting insights from comments. Jiang and Hassan explored the repercussions of outdated comments on bugs, [34], and introduced a method to automatically detect lock-related bugs through comment patterns, and [35], examined "TODO" task comments as tools for inter-programmer communication. A connection was established between "TODO" comments and software bugs, [36], and developed an approach to identify current "TODO" comments. [37], investigated the potential for commented-out code to contribute to bugs, [38]. Regarding comment-reader dynamics, [39], found novices rely more on comments than pros, while, [40], assessed comment relevance via text filtering. Importantly, comments often remain detached from source code syntax trees, impacting automated refactoring. Kramer's case study highlighted Java programmers creating Javadoc comments, [41].

3 Proposed Approach

The suggested research procedure includes a highly planned workflow created to draw out useful information from a Java codebase via a series of methodical studies. The process begins with user input, where the user chooses a directory holding

Java code files and specifies a specific location for a CSV file that will hold the results of the next analytical work.

The process begins with user input, where the user selects a directory containing Java code files or specifies a single Java file for analysis. Additionally, users can specify filtering criteria, such as analyzing only .java files that exceed a certain line count or contain specific keywords. The selected files are then processed, and the extracted insights are stored in a specified CSV file for further analysis. For example, if a user provides the directory path: `/home/user/projects/java_code/`.

The Java codebase used for the analysis in this study is sourced from a public GitHub repository, [42].

The system will extract comment lines, whitespace usage, and other relevant patterns from the specified files and save the results in an output CSV file like: `/home/user/projects/output/code_analysis_results.csv`.

At this point, the Naive Bayes algorithm's strength is put to use. As shown in Table 1 (Appendix), Naïve Bayes offers computational efficiency suitable for text classification tasks.

The code comments are carefully classified into one of two sentiment orientations—single-line or double-line comments—using sentiment analysis. The textual content of comments is processed by the Naive Bayes algorithm, which is renowned for its efficiency in text classification tasks, and a probability score is given for each sentiment category, [48]. The system probabilistically finds the most likely sentiment label for each comment by taking the terms used in the comments and their correlation with recognized sentiment labels into account.

The complete set of results is saved within the selected CSV file, enabling future reference, reporting, and subsequent research endeavors, to ensure the longevity and accessibility of the study outcomes. Python, a flexible programming language, plays a key role in this process, enabling easy data extraction from the complex coding structures of the Java codebase. Figure 1 summarises the proposed research methodology.

A combination of quantitative and qualitative research methods has been adopted. The project intends to thoroughly discover both structured code patterns and the subtle insights hidden in comments by integrating both methodologies, developing a comprehensive grasp of coding practices and their ramifications.

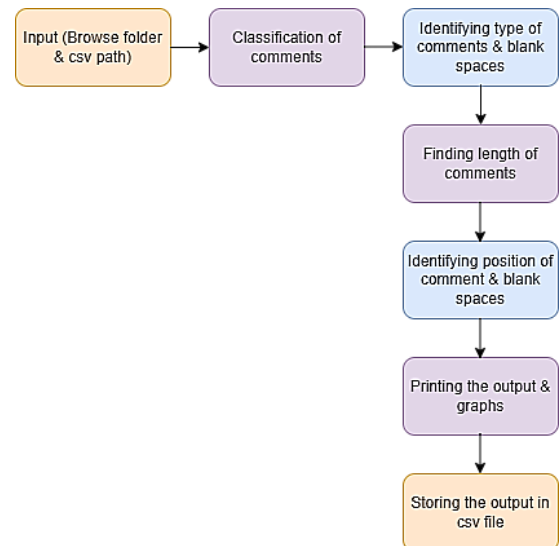


Fig. 1: Proposed system architecture

3.1 Methods and Procedure of Research

1) *Code Analysis Techniques*: Python libraries and tools were harnessed for in-depth code analysis. Employing techniques like regular expressions and parsing libraries, essential elements such as code patterns, function names, variable names, and comments were precisely identified. This comprehensive approach systematically categorized these components, enhancing the understanding of the code's core elements.

2) *Pattern Recognition*: The suggested model entails the creation of a Naïve Bayes algorithm intended to spot reoccurring patterns in code. The model attempts to find common coding practices by utilizing multiple techniques like classification, and sequence analysis. The model will use these methods, in particular, to find both single-line and multi-line comment patterns in the codebase. The proposed model will initially use `"/"` to denote the beginning of a single-line comment. The system will be able to identify situations where programmers provide quick notes on a single line of code. On the other hand, the model will use `"/"` to denote the beginning of a comment that spans many lines for multi-line comments. With the use of this indication, the model will be able to recognize annotations or explanations that span numerous lines of code and are more thorough.

3) *Data Visualization*: During the model implementation, visualizations were crafted to depict code patterns and comment insights. Utilizing graphs, these visual aids effectively conveyed the research findings to a diverse audience, encompassing both technical and non-technical stakeholders.

The proposed research paradigm starts the data collection process by engaging the user, who then

chooses a directory containing Java code files that will serve as the analysis's data source. The model effectively navigates the selected directory, identifying Java code files by their different file extensions, using the flexible Python libraries `os` and `glob`. It uses standard Java comment symbols like `//` for single-line comments or `/*` and `*/` for block comments. The model also stores useful information about each comment like line number etcetera. Comments are pre-processed before actual analysis. This includes steps like conversion to lowercase text and removing symbols. These comments are then assigned sentiment category-tags. These tags can assist with supervised machine learning as the model requires labelled data. Thus comment-dataset can lead to identify the purpose of each comment.

3.2 Data Analysis Methods

The techniques that help in finding useful information in Java code include both – code structure and language content. Following are these processes:

- 1) *RegEx (Regular Expression) Matching*: Program structures like variable declarations, variable names, function definitions etcetera were used to identify code patterns. This helps in maintaining consistency even when the code files are from different projects.
- 2) *Parsing tools*: To identify how different parts of the program code are related, Python tools and libraries were used for parsing. This helped in finding the usual and unusual coding patterns within the code.
- 3) *Text mining*: To detect common relationship, theme for a group of code. It requires natural language understanding, like, if the code is for error handling, comments etcetera.
- 4) *ML Algorithms*: Machine learning classifiers like Naïve Bayes were used for analysis. This helped in distinguishing the well written part of code from the unreliable one.
- 5) *Sequence Analysis*: Generalizations about coding habits and practices can be arrived at, by identifying flow of coding statements.
- 6) *Tools for Visualization*: Node-edge concept of graph representations help in recognizing loops or dependency. This further helps in gaining insights about code.

When applied on Java code, the above techniques of data analysis give useful information about coding style and comments type. This combinational analysis of both comment and code can help developers and managers take strategic

decisions on coding standards. This can go a long way in making project development more efficient.

4 Result and Analysis

The approach discussed above is capable of identifying the comment type, its location, as well as its length. Programmers and managers can use this model to write their comments more clearly and purposeful.

1) Total Number of Comment Lines & Tracking Comment Positions

The Java code was broken down using defined rules (parsing). Then the comment lines were identified and counted. The dataset was divided into a training set and a testing set, typically, a 70-30 split was used. This ensured the Naïve Bayes classifier was tested on unseen data, thus enabling a realistic estimate of its classification performance. To evaluate the performance of the Naive Bayes classifier, metrics were computed, including accuracy, recall, and precision, [49]. The classifier achieved an accuracy - of 85%, precision - of 82%, and recall - of 88%, indicating a high level of effectiveness. The model is able to calculate the comment length and placement, giving meaningful insights into documentation habits. Project development with high industry standards can be practiced through this approach.

2) Differentiating Comment Types

The above suggested model can recognize single-line comment (`//`) from a multi-line one (`/* ... */`). The software developers can thus enforce coding conventions for their projects. This helps in establishing a standard protocol across whole project.

3) Determining Comment Line Length

The suggested model can accurately count all lines between `/*` and `*/` symbols. This can help in validating the need or quality of multiline comments. This feature can make the code more readable and maintainable.

4) Determining blank spaces in code snippets

The blank lines in the Java code can be identified by this model. Strategic placement of blank lines can make the code appear more organized. Such a code is easier to modify for future developments. Team collaborations are easier and efficient for such projects.

5) Improving Code Comprehension and Maintainability

In situations where developers work with code not written by them, the above method can help in understanding the logic faster. Developers can grasp the flow of program faster and also add new

comments or modify obsolete ones. Identify. Table 2 presents the key metrics such as the distribution of various comment types (e.g., single-line, multi-line, Javadoc) and how long they are, providing insight into documentation quality. These insights assist in defining optimal commenting practices for future code development. By understanding these patterns, developers can establish a consistent and clear commenting style that aligns with both industry standards and the specific requirements of the codebase.

Table 2. Determining blank spaces and average length of comment lines

The average length of single-line comments:	39.78
The average length of multi-line comments:	86
Blank spaces count	210

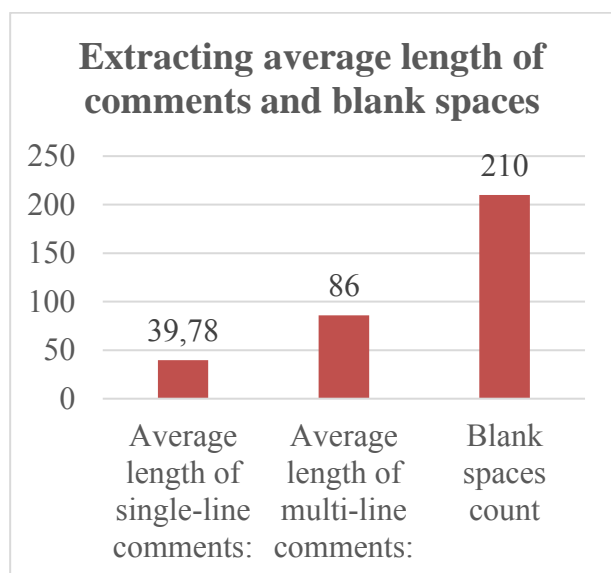


Fig. 2: Extracting the average length of comments and blank spaces in the proposed model

Figure 2 quantifies the number of blank lines or spaces and the average length of comments present within the code. While blank spaces do not contribute to the functional aspects of the code, they impact its visual structure and readability.

4.1 Output of the Proposed Model

Specifically, the following metrics are presented visually:

1) *Average length of single-line comments*: The proposed method generates a graph or histogram similar to the metric mentioned below in Figure 2 that graphically represents the distribution of comment lengths after automatically calculating the average length of single-line comments.

2) *Average length of multi-line comments*: The method determines the typical length of multi-line

comments and generates a graph or histogram showing the range of lengths, similar to the metric mentioned (Figure 2).

3) *Blank spaces count*: The method proposed creates a graph or histogram that shows how frequently blank spaces occur in the code. This visualization similar to the metric mentioned below as Figure 3 helps developers spot blank line patterns and comprehend how they affect the overall structure of the code.

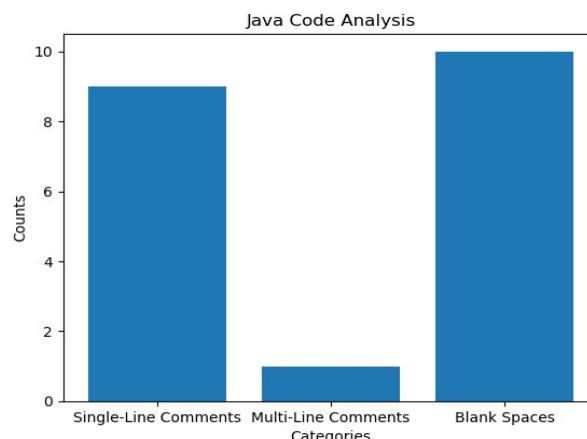


Fig. 3: Histogram produced by the suggested model

An automatic visualization module is integrated into the model, producing graphical outputs, for example - histograms. These graphs reflect patterns related to code comments and blank lines. These visual tools contribute towards better decision-making for code improvement thereby contributing to overall code readability and maintainability.

5 Discussion

5.1 Conclusion

In conclusion, the proposed model offers a robust solution for improving dimensions of software quality, like documentation, and code understanding. It provides detailed analysis by counting comment lines, classifying comment-types, calculating comment lengths, and studying blank spaces.

Also, this study highlights the implications of automated comment analysis in software development. Well-written comments are fundamental for code maintainability, team collaboration, and knowledge transfer. The proposed model not only enables the identification of documentation gaps, ensures consistency, and improves readability, but it also improves software quality. Its design can be integrated into automated

code review systems, IDEs, or CI/CD workflows, offering real-time insights into documentation quality.

The work also paves the way for future work. Advancements such as using deep learning models for better comment classification, integrating sentiment analysis to better capture developer intent, and scaling the model to large open-source projects hold considerable promise. The idea of cross-language applicability may also broaden the model's impact beyond Java. With code getting more complex with time, tools like this will be the key to writing well-structured, maintainable codebases and fostering effective collaborations.

5.2 Limitations

The current model has several limitations. It is specifically designed for Java class files, which restricts its use to other programming languages with different commenting conventions. Additionally, the dataset used for training the model is derived from a specific set of Java-based projects, which could introduce bias and limit the model's ability to generalize across codebases. The use of the Naive Bayes algorithm assumes independence of features, which may impact classification accuracy for complex comment structures. Finally, the model does not assess subjective elements of comments, such as clarity or relevance, which could be explored through further qualitative research.

5.3 Future Work

Future work on the proposed model can lead to significant enhancements in its applicability and performance. Expanding its capabilities to support multiple programming languages will enable broader usage across various coding environments. Additionally, leveraging larger and more diverse datasets can enhance the model's robustness and reduce bias in classification. Incorporating advanced deep learning techniques, like RNNs or transformer-based models like BERT, holds promise for improving classification accuracy. Furthermore, adopting a hybrid approach that combines a rule-based approach with machine learning could lead to improved pattern detection. Lastly, the exploration of automated comment generation and quality assessment mechanisms can improve real-time feedback, helping to establish consistent and effective documentation practices in software development teams.

Declaration of Generative AI and AI-assisted Technologies in the Writing Process

During the preparation of this work the authors used Grammarly in order to improve the readability and language of the manuscript. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

References:

- [1] Xia, X. *et al.* (2018) 'Measuring Program Comprehension: A Large-Scale Field Study with Professionals', *IEEE Transactions on Software Engineering*, 44(10), pp. 951–976. <https://doi.org/10.1109/TSE.2017.2734091>.
- [2] Yu, Y. *et al.* (2016) 'Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?', *Information and Software Technology*, 74, pp. 204–218. <https://doi.org/10.1016/j.infsof.2016.01.004>.
- [3] Rani, P., Panichella, S., *et al.* (2021) 'What do class comments tell us? An investigation of comment evolution and practices in Pharo Smalltalk', *Empirical Software Engineering*, 26(6). <https://doi.org/10.1007/s10664-021-09981-5>.
- [4] Desikan, B. (2018) *Natural Language Processing and Computational Linguistics: A practical guide*, Packt Publishing Ltd. ISBN: 978-1-78883-853-5.
- [5] Ellis, B., Stylos, J. and Myers, B. (2007) 'The factory pattern in API design: A usability evaluation', in *Proceedings - International Conference on Software Engineering*, pp. 302–311. <https://doi.org/10.1109/ICSE.2007.85>.
- [6] Hamed, B.A., Ibrahim, O.A.S. and Abd El-Hafeez, T. (2023) 'Optimizing classification efficiency with machine learning techniques for pattern matching', *Journal of Big Data*, 10(1). <https://doi.org/10.1186/s40537-023-00804-6>.
- [7] Tenny, T. (1988a) 'Program Readability: Procedures Versus Comments', *IEEE Transactions on Software Engineering*, 14(9), pp. 1271–1279. <https://doi.org/10.1109/32.6171>.
- [8] Woodfield, S.N., Dunsmore, H.E. and Shen, V.Y. (1981) 'The effect of modularization and comments on program comprehension', in *Proceedings - International Conference on Software Engineering*, pp. 215–223.

- <https://dl.acm.org/doi/abs/10.5555/800078.802534>.
- [9] Oman, P. and Hagemester, J. (1992a) 'Metrics for assessing a software system's maintainability', in *Proceedings - Conference on Software Maintenance, ICSM 1992*, pp. 337–344.
<https://doi.org/10.1109/ICSM.1992.242525>.
- [10] Fluri, B. *et al.* (2009) 'Analyzing the co-evolution of comments and source code', *Software Quality Journal*, 17(4), pp. 367–394.
<https://doi.org/10.1007/s11219-009-9075-x>.
- [11] Pascarella, L., Bruntink, M. and Bacchelli, A. (2019) 'Classifying code comments in Java software systems', *Empirical Software Engineering*, 24(3), pp. 1499–1537.
<https://doi.org/10.1007/s10664-019-09694-w>.
- [12] Chen, Q. and Zhou, M. (2018) 'A neural framework for retrieval and summarization of source code', in *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, Inc, pp. 826–831.
<https://doi.org/10.1145/3238147.3240471>.
- [13] Hu, X., Li, G., Xia, X., Lo, D. and Jin, Z. (2018) 'Deep code comment generation', in *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, pp. 200–210.
<https://doi.org/10.1145/3196321.3196334>.
- [14] Hu, X., Li, G., Xia, X., Lo, D., Lu, S., *et al.* (2018) 'Summarizing source code with transferred API knowledge', in *IJCAI International Joint Conference on Artificial Intelligence*, pp. 2269–2275.
<https://doi.org/10.24963/ijcai.2018/314>.
- [15] Iyer, S. *et al.* (2016) 'Summarizing source code using a neural attention model', in *54th Annual Meeting of the Association for Computational Linguistics, ACL 2016 - Long Papers*. Association for Computational Linguistics, pp. 2073–2083.
<https://doi.org/10.18653/v1/p16-1195>.
- [16] Wong, E., Yang, J. and Tan, L. (2013a) 'AutoComment: Mining question and answer sites for automatic comment generation', in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp. 562–567.
<https://doi.org/10.1109/ASE.2013.6693113>.
- [17] Padioleau, Y., Tan, L. and Zhou, Y. (2009) 'Listening to Programmers - Taxonomies and characteristics of comments in operating system code', in *Proceedings - International Conference on Software Engineering*, pp. 331–341.
<https://doi.org/10.1109/ICSE.2009.5070533>.
- [18] Steidl, D., Hummel, B. and Jürgens, E. (2013) 'Quality analysis of source code comments', in *IEEE International Conference on Program Comprehension*, pp. 83–92.
<https://doi.org/10.1109/ICPC.2013.6613836>.
- [19] Iammarino, M. *et al.* (2020) 'A Topic Modeling Approach to Evaluate the Comments Consistency to Source Code', in *Proceedings of the International Joint Conference on Neural Networks*.
<https://doi.org/10.1109/IJCNN48605.2020.9207651>.
- [20] Pascarella, L., Bruntink, M. and Bacchelli, A. (2019) 'Classifying code comments in Java software systems', *Empirical Software Engineering*, 24(3), pp. 1499–1537.
<https://doi.org/10.1007/s10664-019-09694-w>.
- [21] Ciurumelea, A. *et al.* (2023) 'Completing Function Documentation Comments Using Structural Information', *Empirical Software Engineering*, 28(4).
<https://doi.org/10.1007/s10664-022-10284-6>.
- [22] Shinyama, Y., Arahori, Y. and Gondow, K. (2018) 'Analyzing Code Comments to Boost Program Comprehension', in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pp. 325–334.
<https://doi.org/10.1109/APSEC.2018.00047>.
- [23] Padioleau, Y., Tan, L. and Zhou, Y. (2009) 'Listening to Programmers - Taxonomies and characteristics of comments in operating system code', in *Proceedings - International Conference on Software Engineering*, pp. 331–341.
<https://doi.org/10.1109/ICSE.2009.5070533>.
- [24] Tan, L. *et al.* (2007) 'iComment: Bugs or bad comments?', in *SOSP'07 - Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pp. 145–158.
<https://dl.acm.org/doi/abs/10.1145/1294261.1294276>.
- [25] Jiang, Z.M. and Hassan, A.E. (2006) 'Examining the evolution of code comments in PostgreSQL', in *Proceedings - International Conference on Software Engineering*, pp. 179–180.
<https://doi.org/10.1145/1137983.1138030>.
- [26] Storey, M.A. *et al.* (2008) 'Todo or to bug: Exploring how task annotations play a role in the work practices of software developers', in *Proceedings - International Conference on*

- Software Engineering*, pp. 251–260.
<https://doi.org/10.1145/1368088.1368123>.
- [27] Ying, A.T.T., Wright, J.L. and Abrams, S. (2005) ‘Source code that talks: An exploration of eclipse task comments and their implication to repository mining’, in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*.
<https://dl.acm.org/doi/abs/10.1145/1082983.1083152>.
- [28] Sridhara, G. *et al.* (2010) ‘Towards automatically generating summary comments for Java methods’, in *ASE’10 - Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 43-52.
<https://doi.org/10.1145/1858996.1859006>.
- [29] Aman, H. *et al.* (2014) ‘Empirical analysis of comments and fault-proneness in methods’, in Association for Computing Machinery (ACM), p.1.
<https://doi.org/10.1145/2652524.2652592>.
- [30] Hirata, Y. and Mizuno, O. (2011) ‘Do comments explain codes adequately? Investigation by text filtering’, in *Proceedings - International Conference on Software Engineering*, pp.242-245.
<https://doi.org/10.1145/1985441.1985482>.
- [31] Sridhara, G. (2016) ‘Automatically detecting the up-to-date status of TODO comments in Java programs’, in *ACM International Conference Proceeding Series*. Association for Computing Machinery, pp.16-25.
<https://doi.org/10.1145/2856636.2856638>.
- [32] Wong, E., Yang, J. and Tan, L. (2013b) ‘AutoComment: Mining question and answer sites for automatic comment generation’, in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp. 562–567.
<https://doi.org/10.1109/ASE.2013.6693113>.
- [33] Li, Z. *et al.* (2018) ‘VulDeePecker: A Deep Learning-Based System for Vulnerability Detection’, in *arxiv.orgZ Li, D Zou, S Xu, X Ou, H Jin, S Wang, Z Deng, Y ZhongarXiv preprint arXiv:1801.01681, 2018•arxiv.org*. Internet Society.
<https://doi.org/10.14722/ndss.2018.23158>.
- [34] Saccente, N. *et al.* (2019) ‘Project achilles: A prototype tool for static method-level vulnerability detection of Java source code using a recurrent neural network’, *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW 2019*, pp.114–121.
<https://doi.org/10.1109/ASEW.2019.00040>.
- [35] Huang, Y. *et al.* (2023) ‘A comparative study on method comment and inline comment’, *ACM Transactions on Software Engineering and Methodology*, 32(5).
<https://doi.org/10.1145/3582570>.
- [36] Ying, A.T.T., Wright, J.L. and Abrams, S. (2005) ‘Source code that talks: An exploration of eclipse task comments and their implication to repository mining’, in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*.
<https://dl.acm.org/doi/abs/10.1145/1082983.1083152>.
- [37] Horvath, A. *et al.* (2022) ‘Using Annotations for Sensemaking About Code’, in *UIST 2022 - Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, Inc.
<https://doi.org/10.1145/3526113.3545667>.
- [38] Mohayeji, H. *et al.* (2022) ‘On the Adoption of a TODO Bot on GitHub: A Preliminary Study’, in *Proceedings - 4th International Workshop on Bots in Software Engineering, BotSE 2022*. Institute of Electrical and Electronics Engineers Inc., pp.23-27.
<https://doi.org/10.1145/3528228.3528408>.
- [39] Sellitto, G. *et al.* (2022) ‘Toward Understanding the Impact of Refactoring on Program Comprehension’, in *Proceedings - 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022*, pp.731-742.
<https://doi.org/10.1109/SANER53432.2022.00090>.
- [40] Dramko, L. *et al.* (2023) ‘DIRE and its Data: Neural Decompiled Variable Renamings with Respect to Software Class’, *ACM Transactions on Software Engineering and Methodology*, 32(2).
<https://doi.org/10.1145/3546946>.
- [41] Sun, Z. *et al.* (2022) ‘On the Importance of Building High-quality Training Datasets for Neural Code Search’, in *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, pp. 1609–1620.
<https://doi.org/10.1145/3510003.3510160>.
- [42] 78meenakshi (2025). java-data. GitHub, [Online].
<https://github.com/78meenakshi/java-data> (Accessed Date: February 25, 2025).

- [43] J. D. Rennie, L. Shih, J. Teevan, and D. Koller, "Tackling the poor assumptions of Naive Bayes text classifiers," in Proc. 20th Int. Conf. Machine Learning (ICML 2003), 2003, pp. 616-623.
- [44] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [45] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.
- [46] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [47] Y. Goldberg, *Neural Network Methods for Natural Language Processing*, Synthesis Lectures on Human Language Technologies, vol. 10, no. 1, pp. 1–309, 2017.
- [48] Wang, C. *et al.* (2023) 'Suboptimal Comments in Java Projects: From Independent Comment Changes to Commenting Practices', *ACM Transactions on Software Engineering and Methodology*, 32(2), p.33. <https://doi.org/10.1145/3546949>.
- [49] Veziroğlu, M., Veziroğlu, E., & Bucak, İ. Ö. (2024). Performance comparison between Naive Bayes and machine learning algorithms for news classification. In *Bayesian Inference-Recent Trends*. IntechOpen. DOI: 10.5772/intechopen.1002778.

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

The authors equally contributed in the present research, at all stages from the formulation of the problem to the final findings and solution.

Sources of Funding for Research Presented in a Scientific Article or Scientific Article Itself

No funding was received for conducting this study.

Conflict of Interest

The authors have no conflicts of interest to declare.

Creative Commons Attribution License 4.0 (Attribution 4.0 International, CC BY 4.0)

This article is published under the terms of the Creative Commons Attribution License 4.0

https://creativecommons.org/licenses/by/4.0/deed.en_US

APPENDIX

Table 1. Comparison of Naïve Bayes with Other Text Classification Models for Java Code Analysis

Criteria	Naïve Bayes	Support Vector Machines (SVMs)	Decision Trees	Deep Learning (e.g., LSTMs, Transformers)
Suitability for Text Classification	Well-suited, widely used in NLP tasks, [43]	Effective for text but requires extensive tuning, [44]	Can be used, but often overfits textual data, [45]	Highly effective but requires a large dataset, [46]
Computational Efficiency	Fast, low resource consumption, [43]	Computationally expensive, especially with large data, [44]	Moderate, but grows exponentially with depth, [45]	Very high, requires GPUs and large memory, [46]
Assumption of Feature Independence	Assumes word/token independence, and works well for structured text, [43]	No independence assumption works well for high-dimensional data, [44]	No independence assumption, but requires pruning to avoid overfitting, [45]	No independence assumption, but highly complex modeling, [46]
Performance on Small Datasets	Performs well even with limited data, [43]	Requires large data for optimal performance, [44]	Tends to overfit with small datasets, [45]	Requires vast amounts of data for effective training, [46]
Ease of Implementation	Simple to implement with existing libraries, [43]	Requires parameter tuning (kernel selection, C value, etc.), [44]	Easy to implement but complex for text classification, [45]	Complex implementation requires a deep learning framework, [46]
Interpretability	Highly interpretable with probabilistic output, [43]	Moderate interpretability, [44]	Highly interpretable but can create overly complex trees, [45]	Low interpretability due to its black-box nature, [46]
Scalability for Large Codebases	Scales efficiently with minimal resource use, [43]	Slower with large-scale datasets, [44]	Struggles with deep trees on large datasets, [45]	Requires high-end computing infrastructure, [46]
Training Time	Very fast, [43]	Moderate to slow, [44]	Moderate but increases with tree complexity, [45]	Extremely slow, especially for large models, [47]
Suitability for Java Code Comment Analysis	Well-suited due to the structured nature of comments, [43]	Can be used, but with high computational overhead, [44]	Not ideal due to overfitting issues, [45]	Complex for this task, [47]