Verifier Warnings Do Not Improve Comprehensibility Prediction

Nadeeshan De Silva
William & Mary
Williamsburg, Virginia, USA
kgdesilva@wm.edu

Martin Kellogg
New Jersey Institute of Technology
Newark, New Jersey, USA
martin.kellogg@njit.edu

Oscar Chaparro
William & Mary
Williamsburg, Virginia, USA
oscarch@wm.edu

Abstract

Proponents of software verification suggest that code simplicity is linked to the effort to verify code, hypothesizing that formal verifiers produce fewer false positive warnings and require less manual intervention when analyzing simpler code. A recent meta-analysis study found empirical support for this hypothesis: a small correlation between the sum of verifier warnings and human-derived code comprehensibility metrics. Based on this finding, we conjectured that using the sum of verifier tool (verifier) warnings to represent program semantic information as an input feature to machine learning (ML) models for code comprehensibility prediction can enhance their performance, when combined with traditional syntactic and developer features.

To test this conjecture, we performed a control-treatment experiment incorporating the *verifier warning sum* feature into machine learning models from the literature, and conducted a comparative analysis of their performance against models trained only on syntactic and developer features. We found no significant difference in the prediction performance of models with and without the warnings feature. Our findings suggest that while a correlation exists, the *verifier warning sum* offers limited discriminative power: combining syntactic and developer features is just as effective for predicting human-judged code comprehensibility.

ACM Reference Format:
Nadeeshan De Silva, Martin Kellogg, and Oscar Chaparro. 2026. Verifier Warnings Do Not Improve Comprehensibility Prediction. In *Proceedings of The 30th International Conference on Evaluation and Assessment in Software Engineering (EASE 2026).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In order to perform software engineering tasks such as feature development, code review, refactoring, or bug fixing, developers must deeply understand the code they are working with [10, 48, 52, 85]. Gaps in such understanding can lead to introducing defects, degrading code quality, and ultimately increasing code maintenance effort [25, 43, 79]. While understanding existing code is essential, it is often challenging and time-consuming; prior studies estimate that developers spend more than half of their time trying to understand existing code [52, 87].

EASE 2026, 9–12 June, 2026, Glasgow, Scotland, United Kingdom

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Researchers have proposed various mechanisms to measure code comprehension effort (*a.k.a.* code comprehensibility), to control for code that is hard to understand during project evolution. One mechanism is directly asking humans and collecting human judgments during user studies. However, measuring how well developers understand code is difficult: human judgments are subjective, vary widely, and are expensive to collect [21, 27, 73, 84]. To address this challenge, researchers introduced objective proxies [13, 68], such as structural measures (*e.g.*, lines of code and comment presence) and complexity metrics (*e.g.*, McCabe's [49] and Halstead's [34]). However, these metrics do not correlate strongly with comprehension measures collected from humans [26, 60, 68].

This mismatch led to researchers proposing machine-learning (ML) approaches that leverage combinations of metrics (*a.k.a.* features) to approximate human comprehensibility [13, 45, 68, 82]. For example, the most recent study [68] evaluated six ML models trained on 121 syntactic and developer-related features to predict human comprehensibility proxies such as *perceived binary understandability*. A current challenge in the research community is to understand what factors could make these models predict code comprehensibility more accurately. In this work, we contribute to this goal by reporting the results of our investigation on leveraging information produced by formal code verification tools (*verifiers*) to predict human comprehension effort.

Our work is motivated by a recent meta-analysis study by Feldman *et al.* [28], which investigated the relationship between formal code verification and human code comprehensibility. Using six existing comprehensibility datasets and four Java verifiers, which check for code correctness properties (*e.g.*, absence of null dereferences), they found evidence that false alarms from such tools may signal when the code is harder to understand for humans. In particular, they showed that such tools produce fewer false positives and require less human intervention on simpler code, reporting a small correlation between the *sum of verifier warnings* from the four verifiers and 20 human comprehensibility proxies.

Intuitively, when code becomes more complex, it makes it much harder for developers to understand it [6–8, 60, 68]. Formal verifiers are designed to handle a certain threshold of code complexity before issuing warnings. From an algorithmic perspective, performing any task requires a baseline level of inherent logic, known as essential complexity [12]. However, if the code can be simplified to eliminate a verification warning without altering its underlying functionality, it indicates that the original implementation contained unnecessary, or accidental, complexity (*i.e.*, it was unnecessarily complex). Complex code tends to obscure logical errors, safety issues, and subtle run-time risks, which formal verifiers aim to detect. As complexity increases, these tools tend to issue more (false positive) warnings and/or require more intervention from human operators (*e.g.*, type

annotations or loop invariants), reflecting the higher potential for subtle defects and verification challenges.

Based on Feldman *et al.*'s findings [28], in this work, we conjecture that verifier warnings could be leveraged as a semantic signal for automatically measuring code comprehensibility via machine learning (ML). To test this conjecture, we designed a control–treatment experiment [75] to investigate whether incorporating the *verifier warning sum* as a program semantic feature enhances the predictive performance of ML models used in prior work [68].

Specifically, in the **Control** setting, we trained six ML models (*e.g.*, Random Forests and Support Vector Machines) using only the syntactic and developer features extracted from two prior datasets [13, 68]. These are the largest comprehensibility datasets available, used by Feldman *et al.* [28]: they contain 150 Java code methods and corresponding 13,860 human comprehensibility measures for five proxies. Using a nested cross-validation approach [1], we evaluated the accuracy of the models in predicting these proxies. In the **Treatment** setting, we introduced the *verifier warning sum* as a new semantic feature and retrained the same models to predict the same comprehensibility proxies. This semantic feature was obtained by executing four state-of-the-art Java code verifiers (*e.g.*, OpenJML [81] and the Checker Framework [57]), also used in the prior study by Feldman *et al.*

Our comparative analysis of model predictive performance between control and treatment settings (section 4) found that adding the verifier warning sum does not significantly improve the predictive performance of code comprehensibility models. We further analyzed the impact of individual verifier-specific warnings (section 4.3) and observed that none of the model-metric-verifier combinations led to consistent, statistically significant improvements in model performance; this aligns with our observations regarding the verifier warning sum. Our findings highlight several promising directions for future work (see section 6), including the need to explore richer semantic representations and other factors that shape code comprehensibility to build more reliable prediction models.

We provide an online replication package [22] that contains our dataset, code, experimental infrastructure and results to facilitate verification and replication of our study.

2 Background and Related Work

This section reviews prior work on code comprehensibility proxies and predictive models of human understandability, and examines the relationship between formal verification and code comprehension, providing background for our work.

Empirical studies of code comprehensibility. Researchers have introduced various proxies to measure the difficulty to understand code, since directly measuring how humans understand code is difficult. The most commonly used proxies are related to *code correctness* (*e.g.*, number of questions about code answered correctly) [68], *subjective ratings* (*e.g.*, Likert scale of how well developers understand the code) [13], *time* (*e.g.*, time to read/understand code), and *physiological* metrics. These physiological measures include eye-tracking data (*e.g.*, gaze duration, fixation count) [4, 9, 31, 40, 58, 61], biometric sensory data [31, 32, 86], fMRI scanner measurements [60, 62, 74], and heart rate sensor data [76].

Prior studies have found that traditional code metrics, such as McCabe's cyclomatic complexity [49], Halstead volume [34], coupling/cohesion [77], and maintainability index [56], poorly correlate with these proxies [6, 26, 38, 39, 60, 68]. For example, Scalabrino *et al.* [68] found only a small correlation between code- and developer-related metrics collected from developers and students using open-source projects. Furthermore, the findings of Peitek *et al.* [60] suggest that code complexity metrics such as cyclomatic complexity are ineffective for measuring code comprehensibility.

Predictive models of understandability. The low correlation between traditional metrics and comprehensibility proxies has motivated the development of predictive models of comprehensibility, which directly predict human-judged proxies based on code properties (*e.g.*, lines of code or cyclomatic complexity) and developer information (*i.e.*, programming experience). These models aim to predict how difficult it is for a particular developer to understand a given code snippet.

Numerous studies have proposed machine learning models for predicting comprehensibility [13, 24, 45, 65, 68–70, 82], which is typically formulated as a classification problem (*e.g.*, predict whether the code is "easy" or "hard" to understand). Some studies have developed regression models [44, 45, 68, 82] (*e.g.*, predict comprehension time), but their accuracy is typically lower due to the inherent noise of continuous proxies, stemming primarily from hard to control factors during user studies (*e.g.*, different pace of participants when reading and understanding the code). Model inputs in these works include structural features (*e.g.*, loops, operators, blank lines) [13, 65], aggregated features such as visual matrix representations of code tokens and alignment-based metrics [24], and lexical features including comment readability and textual coherence introduced in more recent studies [69, 70].

Among the prior studies, Scalabrino *et al.* [68] present the current state of the art in the area, as they developed several ML models (*e.g.*, Random Forests and Support Vector Machines) that outperform earlier approaches [82]. In this work, we build upon Scalabrino *et al.*'s methodology to develop and evaluate ML models of comprehensibility, to test whether semantic information from code verifiers improves model accuracy.

With the emergence of LLMs for code generation, recent research has examined the readability of AI-generated code [20, 59, 71]. Dantas *et al.* [20] performed an analysis comparing code generated by ChatGPT with human-written code from Stack Overflow. They evaluated the code snippets using the SonarQube tool to identify and analyze potential code readability issues. Sergeyuk *et al.* [71] performed a comparison against existing code readability models (*e.g.*, [51, 64, 68]) on AI-generated Java snippets and evaluated the same snippets with a human study, finding that existing models poorly correlate with human judgments—making those models unsuitable as proxies for human evaluation of AI-generated code. Patel *et al.* [59] compare AI-generated (Microsoft Copilot) and human-written Java solutions to LeetCode problems using static analysis tools (Understand, SpotBugs, PMD—none of which are verifiers), finding that AI code tends to be longer but has lower bug density than human code. None of these latest studies introduces a predictive model of code comprehensibility, and hence we decided to work with models introduced by Scalabrino *et al.* [68]. However, these studies emphasize the importance of code comprehensibility

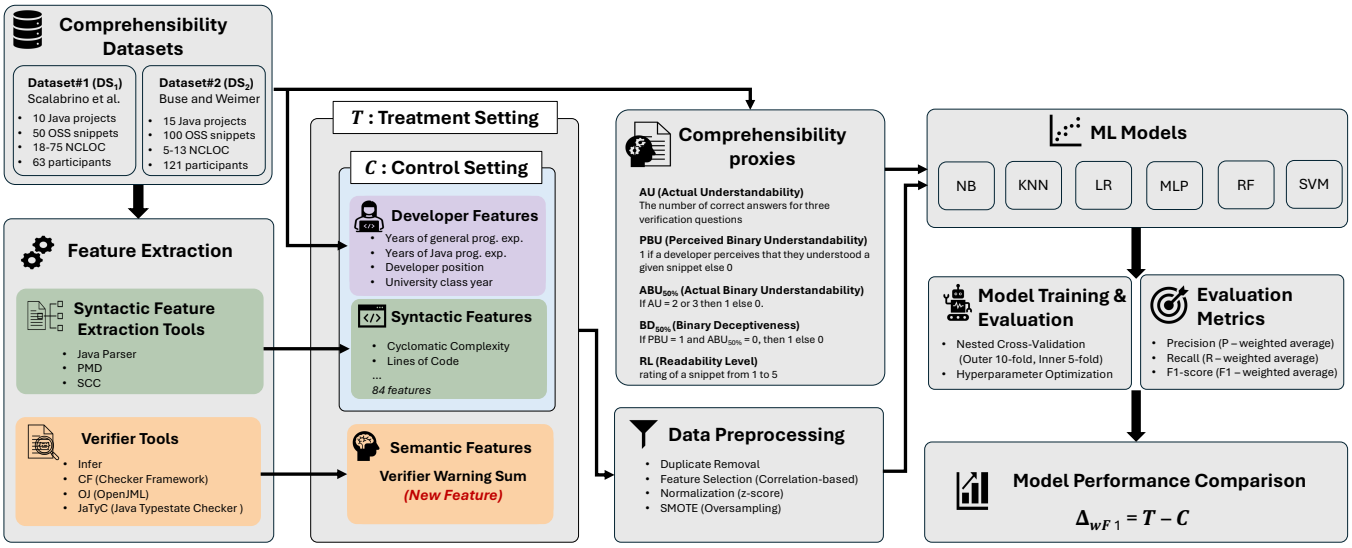

Figure 1: Overview of our methodology for evaluating the impact of verifier warnings on code comprehensibility prediction. We compare two model variants: a *control* model trained on syntactic and developer features, and a *treatment* model trained on the same features plus the verifier warning sum as a semantic feature. Both models are evaluated using nested cross-validation.

prediction models and the need to improve them to better estimate human comprehension of code, thereby supporting practical software maintenance tasks more effectively.

Formal code verification and comprehensibility. Recent research by Feldman *et al.* [28] investigates the relationship between code verifiability and human understandability. Our work is directly motivated by their work, as they concluded that *"our work has implications for the users and designers of verification tools and for future attempts to automatically measure code comprehensibility"*. They conducted a meta-analysis study to measure the correlation between human-judged code comprehensibility and the difficulty of formally verifying code (*i.e.*, verifiability). Using six comprehensibility datasets from prior studies, spanning 18,005 human-collected comprehensibility measurements for 211 Java methods, the study compared the number of "false positive" warnings generated by four state-of-the-art static verifiers against 20 human-based comprehensibility proxies of different categories (proxies related to correctness, time, and physiological measurements). The study established a measurable but small correlation (Pearson's $r = 0.23$) between the two domains, implying that code which is easier/difficult for formal tools to verify is easier/difficult for human developers to comprehend. Verifiers inherently must reason about program semantics to prove safety properties; therefore, when they fail and issue "false positive" warnings due to convoluted logic or missing specifications, these warnings can act as a proxy for semantic code complexity. The study speculated that verification warnings might capture a semantic dimension of cognitive burden that syntactic metrics miss, suggesting that verifiability could be a valuable input for automated code understandability models.

A correlation between code verifiability and human comprehensibility may mean that both humans and formal verifiers struggle with the same underlying factors that cause code complexity. Syntactic metrics often assign different scores to code snippets that look different in structure but implement the same algorithm (*e.g.*, the same sorting algorithm can be implemented with imperative loops or in a functional style), but verifier warnings could capture the "accidental complexity" and unstated assumptions that confuse human developers.

For example, consider accessing an array using a potentially out-of-bounds index in a Java-like language. A simple bounds check might use an if statement to ensure the index falls within the array's valid range. A more complex variant with the same semantics might access the array inside a try block, relying on a catch statement to intercept the resulting exception if the index proves invalid. This more convoluted approach introduces accidental complexity and might fail verification: the out-of-bounds access actually occurs, but it is intercepted before crashing the program. To avoid issuing a false positive warning, a verifier would need to accurately model this exceptional control flow (and, in practice, few verifiers do). Alternatively, a verifier might flag code that relies on unstated assumptions. For instance, accessing an array without a prior bounds check inherently assumes the index has already been validated elsewhere. A verifier might warn that this operation is unsafe unless a developer provides an explicit specification guaranteeing the index's validity for that specific array. With this specification in place, the verifier can approve the array access, but it must then ensure that every integer value assigned to the index variable strictly adheres to that valid range. Thus, a warning triggered by a missing specification often highlights underlying code complexity: a human reader would similarly need to reason through the program's logic to determine exactly why that specific array access is safe.

Table 1: Datasets and Code comprehensibility metrics. NCLOC = Non-Comment/blank Lines of Code

Dataset	Metric	Definition	Class Distribution
DS_1 (Scalabrino *et al.* [68]) 50 OSS snippets of 18 - 75 NCLOC; 50 students and 13 developers	AU (Actual Understandability)	The number of correct answers for three verification questions about the code. Possible values are 0,1,2,3. (higher number implies higher understandability)	0 - 153 (34.7%) 1 - 72 (16.4%) 2 - 138 (31.4%) 3 - 77 (17.5%)
	PBU (Perceived Binary Understandability)	1 if a developer perceives that they understood a given snippet; 0 otherwise. (higher number implies higher understandability)	0 - 136 (30.9%) 1 - 304 (69.1%)
	$ABU_{50\%}$ (Actual Binary Understandability)	If AU = 2 or 3 then 1 else 0. (higher number implies higher understandability)	0 - 225 (51.1%) 1 - 215 (48.9%)
	$BD_{50\%}$ (Binary Deceptiveness)	If PBU = 1 and $ABU_{50\%}$ = 0, then 1 else 0 (higher number implies lower understandability)	0 - 351 (79.8%) 1 - 89 (20.2%)
DS_2 (Buse and Weimer [13]) 100 OSS snippets of 5 - 13 NCLOC; 121 students	RL (Readability Level)	Readability rating of a snippet from 1 to 5 (higher value implies higher readability)	1 - 889 (7.3%) 2 - 2481 (20.5%) 3 - 3240 (26.8%) 4 - 3290 (27.2%) 5 - 2200 (18.2%)

We define the *verifier warning sum* as the summation of false positive warnings generated by the four verifiers used in Feldman *et al.*'s prior study [28] during the analysis of a code snippet.

3 Methodology

This study focuses on ML-based classification models of human-judged code comprehensibility, aiming to assess their effectiveness when *verifier warning sum* is included as a complementary feature. With this in mind, we define this research question:

RQ: *How much more effective are ML models at predicting code comprehensibility when the verifier warning sum is included?*

To answer this question, we designed the *control–treatment* experiment [75] depicted in fig. 1. In the *control setting*, the models are trained using only syntactic and developer features, as in prior work [68]. In the *treatment setting*, the models are trained on the same datasets, feature extraction methods, model architectures, evaluation protocols, and the same feature set augmented with the newly introduced semantic feature: *verifier warning sum*. This setup enables a controlled comparison between the two model variants and allows us to isolate and evaluate the added contribution of verifier warnings to predictive performance.

3.1 Comprehensibility Data Sources

We reused the two largest Java code comprehensibility datasets from prior studies [13, 68] (see table 1), both of which trained ML models on hand-crafted features for code comprehensibility prediction:

- **Dataset #1 (DS_1)**, from Scalabrino *et al.* [68], contains 50 Java Open Source (OSS) methods evaluated by 50 CS students (with intermediate to high programming experience) and 13 professional developers. Participants judged each method's understandability, answered three comprehension questions, and had their task-completion time recorded. OSS methods were extracted from 10 open source Java projects, including OpenCMS, Jenkins, Spring Framework, Weka, and K-9 Mail.
- **Dataset #2 (DS_2)**, from Buse and Weimer [13], consists of simplified snippets from 100 Java OSS methods. A total of 121 CS students rated snippet readability on a 5-point Likert scale. These snippets were purposely stripped (size of the snippets were in the range of 5-13 NCLOC) of contextual and algorithmic complexity to mitigate confounding factors and isolate low-level readability attributes. These snippets were taken from 15 OSS Java projects, including jUnit, jEdit, SoapUI, Hibernate, Azureus/Vuze, and JasperReports.

These two datasets were also used in Feldman *et al.*'s study [28] and were the largest contributors to their meta-analysis results. Given their dominant influence on the findings and their substantially larger size compared to the other datasets considered in such a study [28], which were relatively small, we selected them for this work to ensure robustness and comparability of results.

All comprehensibility measurements were collected for each individual. DS_1 had 6 snippets evaluated by 8 programmers (*a.k.a.* developers) and 44 snippets by 9 individuals. DS_2 had 121 programmers evaluate each of the 100 snippets. In total DS_1 includes 440 human measurements for each of the four code comprehensibility metrics, while DS_2 includes 12,100 measurements for the single metric. These measurements were collected with the code comprehensibility metrics described in section 3.2.

3.2 Code Comprehensibility Metrics

We used the code comprehensibility metrics (*a.k.a.* proxies) introduced in the above two prior studies as prediction targets for our models. These metrics capture different aspects of the participants' code comprehension and fall into three categories: subjective *ratings* (*e.g.*, perceived understandability), the *correctness* of a developer's understanding of program behavior (*e.g.*, based on participants' answers to verification questions), and the *time* required to read or understand the code.

We used the four *correctness* metrics from DS_1 [68] and the single metric from DS_2 [13], all defined in table 1:

- Actual Understandability (**AU**) reflects the participant's understanding of the code's behavior based on how correct are their answers to three verification questions posed by the researchers.
- Perceived Binary Understandability (**PBU**) captures subjective judgment of comprehensibility: whether a programmer considered they understood a code snippet.
- Actual Binary Understandability ($\mathbf{ABU_{50\%}}$), based on AU, provides a stricter measure of comprehensibility as it considers

Table 2: Syntactic features categorized by type.

Category	Examples	# Features
Complexity	Cyclomatic complexity, # of nested blocks, # of loops, # of comparisons, entropy, …	10
Size	LOC, # of parameters, # of statements, # of literals, # of assignments, # of numbers, …	17
Lexicon	# of identifiers, # of keywords, Identifier length, # of operators, text coherence, …	27
Format	# of blank lines, # of spaces, # of parentheses, # of commas, # of periods, line length, …	18
Documentation	# of comments, comment readability, comments and identifier consistency, …	12
Total		**84**

a snippet easy to understand only when the programmer correctly answers at least two of three verification questions.
- Binary Deceptiveness (**$BD_{50\%}$**), based on PBU and $ABU_{50\%}$, highlights potentially misleading or deceptive code as it considers a snippet hard to understand when the programmer perceived they understood the code (PBU = 1) but answer only one or none of the three verifications correctly ($ABU_{50\%}$ = 0).
- Readability Level (**RL**) reflects participants' subjective assessment of how easy the code is to read and understand based on a 5-point Likert scale.

We excluded the two *time* metrics from DS_1 as we focused on classification rather than regression of continuous time values. Also, these metrics have high variability (*e.g.*, the TNPU metric ranges from 3 to 1,649 seconds), influenced by individual differences such as cognitive speed, code familiarity, and task complexity, making them less reliable for modeling purposes.

Table 1 summarizes the metrics definitions and class distributions. All metrics yield discrete comprehensibility scores. In total, DS_2 includes 12,100 RL measurements, while DS_1 includes 440 measurements for each of the other four metrics.

Although the datasets may be relatively small, they are well suited for this study because they contain carefully curated code samples paired with controlled and validated human comprehension measurements. Moreover, our experimental design, including data preprocessing steps and nested cross-validation, described later in this section, help mitigate potential biases and increase the likelihood that meaningful patterns can be learned from the data. Prior studies [13, 68] have successfully trained predictive models and conducted correlation analyses [28, 68] on the same datasets, indicating that they contain sufficient signal for modeling code comprehensibility. While the dataset size may limit generalizability, it is adequate for the comparative analysis between control and treatment settings, which is the primary goal of this study.

3.3 Input Features for Models

Both prior studies [13, 68] leveraged features capturing various aspects of code and developer characteristics to train models for predicting code comprehensibility. In this study, we reused the same set of features from these works and additionally introduce a new feature: *verifier warning sum*, used in the prior meta-analysis study [28]. These features can be categorized into three groups: *syntactic*, *developer*, and *semantic features*.

3.3.1 Syntactic Features. We began with the 115 features defined by Scalabrino *et al.* [68]. Upon reviewing their data, definitions, and implementations, we excluded 38 features that were ambiguously defined (*e.g.*, unclear term "word" in "# of words"), inapplicable to the available snippets (*e.g.*, "# of aligned blocks" for constructors, of which none were present), or missing for more than 30% of snippets (also excluded in the prior study [68]). We then added seven complementary features to ensure consistency (*e.g.*, adding total counts where only normalized counts existed).

In total, we used 84 code features, including cyclomatic complexity, lines of code, and number of loops, parameters, Java keywords, and comments. Table 2 shows examples of the features across five categories: *complexity*, *size*, *lexicon*, *format*, and *documentation*. *Complexity* features include traditional metrics like loop count and cyclomatic complexity, while *size* features account for parameters, statements, and lines of code. *Format* features capture stylistic elements such as blank lines and parentheses, whereas *lexicon* features capture vocabulary, including identifiers and keywords. Finally, *documentation* features account for comments and their readability (*e.g.*, via the Flesch reading-ease test [29]). See our replication package [22] for the full list of features and their definitions.

3.3.2 Developer Features. We reused all developer-related features from prior work [13, 68], which capture aspects of a programmer's background and experience. In DS_1, *years of general programming experience* and *years of Java experience* are modeled as discrete variables, each taking integer values from 1 to 10 to represent increasing levels of experience. The dataset also includes *participant role*, an ordinal variable with four possible values: bachelor student, master student, Ph.D. student, and professional developer. DS_2 includes a single feature, *university class year/academic level*, which represents the participant's college year. This feature is also ordinal, with four levels: first year, second year, third/fourth year, and graduate level. The original data does not distinguish third- and fourth-year participants [13].

3.3.3 Semantic Feature. We consider *verifier warning sum* as a *semantic feature*. Formal verifiers assess the logical correctness of code and its adherence to semantic rules, rather than merely its syntactic form. The warnings they generate indicate potential violations of correctness properties, such as null dereferences, dead code, or run-time errors. By aggregating these warnings as counts, we obtain a quantitative measure that captures underlying program semantics, reflecting how the code behaves or could behave.

We used the replication package from previous meta-analysis study [28] to extract the verifier warning sum from DS_1 and DS_2. We utilized the same four state-of-the-art Java formal verifiers and same released versions to analyze the Java code snippets from the two datasets:

- The Checker Framework (**CF**) [57] is a platform for extending Java's type system to verify properties that the compiler does not reason about by default. By integrating custom type

qualifiers into the Java compiler, it enables developers to enforce properties like nullability or regular expression validity at compile time. We re-used the same nine typecheckers used by Feldman *et al.* [28]: checkers for nullness, interning, object construction, resource leaks, array bounds, signature strings, format strings, regular expressions and optionals.

- OpenJML (**OJ**) [81] is a Java verifier that uses the Java Modeling Language (JML) [46] to check whether program's implementation satisfies its formal specifications. Developers use JML to specify the intended behavior of classes and methods, including preconditions, postconditions, and class invariants. The tool translates the Java code together with its JML specifications into logical statements called verification conditions (VCs). SMT solvers then attempt to mathematically prove that these VCs are valid. OpenJML verifies the absence of common programming errors, including out-of-bounds array accesses, null pointer dereferences, and integer overflows and underflows, even without any programmer-written specifications.
- The Java Typestate Checker (**JaTyC**) [53] implements a formal typestate analysis [78] that extends Java's static type system to monitor the lifecycle states of objects. Unlike standard type systems, JaTyC tracks state transitions (*e.g.*, the sequence of closed, open, and closed for a File object) to ensure protocol adherence. This verifier identifies potential null dereferences, protocol violations in object lifecycles (*e.g.*, sockets or files), and unauthorized memory accesses.
- **Infer** [14] is a bug-finding tool developed by and deployed within Meta, written in OCaml [80], that implements separation logic [55] and bi-abduction [15] to detect common defects such as null pointer dereferences, data races, and resource leaks. Separation logic facilitates scalable, modular reasoning of state mutations, while bi-abduction automates this inference process. Though Infer itself is not *technically* a verifier—its design intentionally omits warnings that are heuristically judged to be likely false positives—we include it because its reasoning is based on a sound core, unlike truly-heuristic bug-finders like SpotBugs [3].

For each snippet, we recorded the total number of warnings issued by each tool and totaled them across all tools; we refer to this aggregated value as the *verifier warning sum*. Feldman *et al.* [28] checked that these warnings are really false positives: they indicate *possible* defects detected by the tools that cannot actually occur if the code is executed. To ensure consistency, we re-ran the analysis using the data provided in their replication package and identified several discrepancies. A major portion of these discrepancies stemmed from warnings generated in artificial stubs rather than in the actual methods under analysis. By re-executing the analysis pipeline end to end, we ensured accuracy and replicability instead of relying solely on the published results in the prior study [28]. We documented these discrepancies in this study's replication package [22].

3.4 Machine Learning Models

We used the same six ML models as in prior work [68]:

- Naïve Bayes (**NB**) [83] predicts the most probable class using Bayes' theorem under a feature-independence assumption
- K-Nearest Neighbors (**KNN**) [63] predicts by taking the majority vote among the k nearest instances
- Logistic Regression (**LR**) [17] predicts class probabilities using a weighted linear combination of features
- Multilayer Perceptron (**MLP**) [66] predicts using a feedforward neural network with one or more hidden layers
- Random Forest (**RF**) [67] predicts through majority voting based on an ensemble of decision trees combined via bagging
- Support Vector Machine (**SVM**) [36] identifies a maximum-margin separating hyperplane to distinguish instances, via linear or nonlinear kernels.

The models are trained to predict a comprehensibility metric or proxy (*e.g.*, PBU) given a code snippet c inspected by programmer p, as represented by a vector of (concatenated) features: syntactic and developer features in the ***Control*** setting, and the same features supplemented with the semantic feature in ***Treatment*** setting. We do this for each model, proxy, and corresponding dataset.

3.5 Data Preprocessing

We applied three standard preprocessing steps: duplicate removal, normalization, and class balancing.

First, duplicates were removed from the training data to prevent overfitting. Duplicate instances in the validation/test sets were preserved to account for realistic model usage. Second, feature values were standardized using z-score normalization [5], transforming values x into $z = \frac{x-\mu}{\sigma}$ to ensure comparability across scales. Finally, to address class imbalance (see table 1), we applied Synthetic Minority Over-sampling (SMOTE) [18] to generate synthetic samples for minority classes in the training data only.

Because the number of features exceeded the number of data instances, we applied correlation-based feature selection (as in [68]) before oversampling. Features were ranked by Kendall's τ [41] correlation with a target comprehensibility metric, and we evaluated models using the top 10%, 20%, …, 100% of features. Due to their small number, all developer features were retained.

In the treatment setting, we augmented the previously selected features with the verifier warning sum.

3.6 Model Training and Evaluation

To reduce overfitting and bias in relatively small datasets, we used nested cross-validation (CV) [1], which tunes and evaluates models iteratively across folds, providing better generalization [16].

Nested CV consists of two levels: the **outer** 10-fold CV splits data into train/test sets, maintaining class balance. Each fold serves once as the test set. The **inner** 5-fold CV optimizes model hyperparameters within each outer training set. We tested multiple hyperparameter combinations (see the replication package [22]), starting from values used in prior work [68] and refining them based on established guidelines [11, 33]. Optimal parameters were selected by the highest weighted F1-score (see section 3.7). We selected models with multiple optimal configurations across folds as this minimizes bias compared to selecting a single best set, as done in prior work [68].

To balance accuracy and training time, we chose 10 outer folds and 5 inner folds. More folds reduce test set size, making results less reliable, while fewer folds risk overfitting. After determining the best hyperparameter sets across all 10 outer training sets, we trained the models using each optimal set and evaluated them on every test

fold. Unlike prior work [68], which selects a single best hyperparameter set, our approach reduces bias by testing multiple optimal configurations, leading to a more reliable performance estimates.

3.7 Evaluation Metrics

Model performance was evaluated using standard classification metrics. We began by computing *precision (P)*, *recall (R)*, and *F1-score (F1)* [37, 54]. Scores were computed per class (P_i, R_i, $F1_i$) and aggregated into weighted averages (wP, wR, $wF1$) based on the class distribution of each comprehensibility proxy. We computed global metrics by summing confusion matrix entries across folds before calculation [30], avoiding distortions from across-fold averaging. Additionally we used the Area Under the Receiver Operating Characteristic Curve (AUC-ROC) [35], which measures a model's ability to classify instances across all classification thresholds, providing a comprehensive measure of predictive power. As with the other metrics, AUC-ROC is computed per classes and aggregated into weighted AUC-ROC ($wAUC$), based on the class distribution of each proxy. For space reasons, we only report summary metrics ($wF1$ and $wAUC$) in tables 3 and 4, but the results based on all the metrics are found in our replication package [22].

Since we compare the absolute metric differences between control and treatment settings, across the six model types, five comprehensibility metrics, and different sets of model hyperparameters, we employed the non-parametric, unpaired Mann-Whitney U test [50] to assess statistical significance (evaluated at a confidence level of 95%). Given a particular Metric ($wF1$ or $wAUC$), the null hypothesis (H_0) posits that $C_{Metric} \geq T_{Metric}$ and, hence, the alternative hypothesis (H_a) posits that $T_{Metric} > C_{Metric}$. C_{Metric} and T_{Metric} represent model performance obtained from the control and treatment settings, respectively.

4 Results and Analysis

Given the combinatorial nature of our experimental design, we ended up training 1,228 classifiers in the control setting (*i.e.*, models trained solely on syntactic and developer features) and 935 in the treatment setting (*i.e.*, models trained on the augmented semantic feature). These total encompass models trained across six machine learning types, ten feature sets, five comprehensibility metrics, and multiple hyperparameter combinations optimized via nested cross-validation (see section 3.6).

The C ("***Control***") columns in table 3 report the aggregated performance of models trained solely on syntactic and developer features, and T ("***Treatment***") columns display the results when using the semantic feature alongside syntactic and developer features. Results are grouped by metric and model type. Model performance is expressed as the average weighted F1-score ($wF1$) and weighted AUC ($wAUC$) across all trained classifiers. We focus our analysis of $wF1$ results as we obtained similar trends with the $wAUC$ metric.

4.1 Analysis across Metrics and Model Types

We first discuss the results across comprehensibility metrics and models types for both control and treatment settings (see table 3).

Among the five comprehensibility metrics, PBU (Perceived Binary Understandability), $ABU_{50\%}$ (Actual Binary Understandability) and $BD_{50\%}$ (Binary Deceptiveness) are the easiest to predict in both control and treatment settings, consistently achieving the highest scores across all model types. When using only the syntactic and developer features, model $wF1$ scores range from 0.489 to 0.702, and supplementing these features with the semantic feature leads to 0.46 to 0.699 $wF1$ scores.

Notably, AU (Actual Understandability) and RL (Readability Level) are more difficult to predict than the other three metrics. Between these two, RL is the most challenging metric to predict regardless of features, obtaining $wF1$ scores ranging from 0.162 to 0.202 in control models, and 0.168 to 0.201 in treatment models. This is confirmed by $wAUC$ results, which shows model performance lower for RL than for AU. We attribute this to two main factors. (1) *Multi-Class Classification Challenge:* Both metrics involve multiple predictive classes (4 for AU and 5 for RL), which are inherently more difficult to predict than binary metrics [23]. (2) *Subjectivity and Noise:* RL relies on subjective ratings provided directly by developers, introducing greater variability and noise compared to objective measures like AU. The decision boundaries between adjacent ratings (*e.g.*, 2 vs. 3) might be subtle, leading to inconsistencies even in human judgement.

Overall, Random Forests (RF) achieved the highest scores across most metrics in both control and treatment settings, with a few exceptions. In the control setting using only syntactic and developer features, RF was slightly below the top-performing models: about 0.5% lower than LR on $ABU_{50\%}$ and 25% lower than MLP on RL. A similar pattern appears when augmenting verifier warnings, where RF trailed LR by roughly 1% on $ABU_{50\%}$ and achieved 0.168 on RL approximately 20% lower than the best-performing MLP. As an ensemble model combining multiple decision trees, RF is capable of capturing complex, non-linear patterns in the data, making it better-suited for the diverse nature of code comprehensibility metrics.

As shown in table 3, both control and treatment models demonstrate some capacity to learn from the data, as their AUC scores outperform the naïve random baseline ($wAUC = 0.5$). At the same time, 47% of models fail to effectively learn meaningful patterns and underperform even the naïve baseline, showing mixed results across models and metrics. Additionally, both $wF1$ and $wAUC$ scores remain generally low across models and evaluation metrics, suggesting that these models provide limited reliability as predictors of code comprehensibility. Our results are in line with Scalabrino *et al.* [68], who concluded that these ML models are far from being practical.

4.2 Analysis of Control vs Treatment Models

Table 3 shows that incorporating the aggregated count of verifier warnings as a semantic feature does not yield any meaningful improvement. Across the 30 model–metric combinations, only 10-13 exhibit any gain depending on the evaluation metric, and even those are marginal and mostly not statistically significant ($\Delta wF1 = +0.002$ to $+0.042$, and $\Delta wAUC = +0.001$ to $+0.009$). 56.6% to 66.7% of all configurations show either no benefit or an outright decline when verifier warnings are added, although with most of the degradation being not statistically significant either.

The largest improvement, observed for $BD_{50\%}$ with the MLP model, reaches only $\Delta wF1 = +0.042$, a small 7.2% relative increase over the control models (trained solely with syntactic and developer features). Although every metric except $ABU_{50\%}$ shows a slight improvement in at least one isolated configuration, no metric exhibits consistent gains across models. RL is the only metric that shows

Table 3: Code comprehensibility prediction results based on wF1 and wAUC, averaged across trained models. C (Control setting): Syntactic + Developer features; T (Treatment setting): Syntactic + Developer + Semantic features; $\Delta = T - C$; A positive difference Δ is highlighted in Green. Statistically significant differences ($p < 0.05$) are in bold.

(a) $wF1$ results for each metric and model

Metric	NB			KNN			LR			MLP			RF			SVM		
	C	T	Δ_{wF1}	C	T	Δ_{wF1}	C	T	Δ_{wF1}	C	T	Δ_{wF1}	C	T	Δ_{wF1}	C	T	Δ_{wF1}
AU	0.273	0.273	0.000	0.278	0.278	0.000	0.324	0.319	-0.004	0.255	0.227	-0.028	0.340	0.344	0.004	0.304	0.296	-0.008
PBU	0.556	0.555	-0.001	0.537	0.524	-0.013	0.593	0.595	0.002	0.516	0.510	-0.005	0.666	0.674	0.008	0.584	0.588	0.004
$ABU_{50\%}$	0.600	0.599	-0.001	0.559	0.553	-0.006	0.614	0.611	-0.003	0.489	0.460	-0.030	0.611	0.605	-0.006	0.589	0.580	-0.009
$BD_{50\%}$	0.579	0.598	**0.018**	0.567	0.542	-0.025	0.572	0.564	-0.008	0.573	0.614	0.042	0.702	0.699	-0.003	0.565	0.557	-0.008
RL	0.179	0.187	**0.007**	0.199	0.194	-0.005	0.178	0.183	0.005	0.202	0.201	**-0.001**	0.162	0.168	**0.007**	0.165	0.169	0.004

(b) $wAUC$ results for each metric and model

Metric	NB			KNN			LR			MLP			RF			SVM		
	C	T	Δ_{wAUC}	C	T	Δ_{wAUC}	C	T	Δ_{wAUC}	C	T	Δ_{wAUC}	C	T	Δ_{wAUC}	C	T	Δ_{wAUC}
AU	0.552	0.548	-0.004	0.535	0.534	-0.001	0.575	0.572	-0.003	0.514	0.503	-0.011	0.586	0.583	-0.003	0.560	0.552	-0.009
PBU	0.511	0.506	-0.005	0.528	0.526	-0.002	0.528	0.529	0.001	0.502	0.500	-0.002	0.556	0.558	0.001	0.526	0.528	0.001
$ABU_{50\%}$	0.497	0.498	0.001	0.498	0.498	0.000	0.497	0.497	0.000	0.499	0.500	**0.001**	0.496	0.496	0.000	0.498	0.498	0.000
$BD_{50\%}$	0.460	0.455	-0.005	0.462	0.462	0.000	0.471	0.477	**0.006**	0.512	0.484	-0.028	0.437	0.438	0.001	0.490	0.495	0.006
RL	0.490	0.494	**0.004**	0.522	0.518	-0.003	0.503	0.512	**0.009**	0.508	0.519	**0.011**	0.489	0.492	**0.003**	0.507	0.514	0.007

improvement in four out of six models, although the gains are small ($\Delta wF1$ ranging from +0.004 to +0.007). These results are consistent with the correlation findings reported by Feldman *et al.* [28], which showed that the *verifier warning sum* exhibits a small correlation with RL, whereas PBU and AU show no correlation. While using features that correlate with the target metric is not the only factor enabling models to learn underlying patterns, it plays a significant role. Here, the correlation of RL with the *verifier warning sum* may explain the consistent improvements in RL observed across models when this feature is added.

Overall, these findings suggest that while verifier-warning counts may occasionally capture patterns related to code comprehensibility when models happen to be well-calibrated, their predictive contribution is generally weak and largely negligible. As such, **adding *verifier warning sum* does not improve ML models' ability to predict code comprehensibility** in a meaningful way.

4.2.1 Analysis across model types and metrics. We analyze the results across model types and metrics to understand this overall finding. RF, SVM, LR and NB remain the most robust performers, achieving $wF1$ and $wAUC$ values up to 0.699 and 0.583 (for treatment), respectively. Having improvements in two to three metrics in these models indicate that adding verifier warnings provides improvements, but marginally. The inclusion of warnings yields limited but occasionally positive effects, suggesting that while they may capture certain comprehensibility patterns, their overall predictive contribution remains weak.

The metrics sensitive to user perception (PBU and RL), show small but consistent improvements across three or four models (RF, SVM, LR, and NB) when verifier warnings are included. This suggests that metrics driven by user ratings and developers' perceived understanding may benefit more from semantic information about the code than purely objective metrics. For example, when code contains issues that verifiers detect, such as pointer dereferences without null checks or array accesses that are not obviously in-bounds, developers may perceive these as more severe, which may influence their judgments of code comprehensibility. Such information is not fully captured by syntactic features alone. Furthermore, the RL metric originates from the DS_2 dataset, which contains simplified code snippets designed to reduce complexity. In this setting, syntactic features alone may be insufficient to capture the subtle factors influencing developers' ratings in these simplified contexts. The addition of semantic features like verifier warnings may help bridge this gap.

In order to understand why incorporating *verifier warning sum* as an additional semantic feature does not improve model performance, we conducted a statistical analysis between this feature and each comprehensibility metric. Rather than relying on a single statistical indicator, we examined the feature from three complementary perspectives: (i) rank-based association using Kendall's τ [42], (ii) general statistical dependency using Mutual Information (MI) [19, 72], and (iii) model-level contribution using SHAP (SHapley Additive exPlanations) values [47].

Kendall's rank correlation tests whether a monotonic relationship exists between the feature and each comprehensibility metric. A near-zero Kendall's τ coefficient indicates the absence of a consistent directional association, irrespective of the underlying data distribution. Mutual Information captures statistical dependencies that extend beyond monotonic relationships, including non-linear and threshold-based effects. A low MI score indicates that the feature and a metric are largely statistically independent, providing no meaningful discriminative information. SHAP value analysis was conducted as a post-hoc model explanation technique to evaluate how the trained classifiers utilized the feature at the individual prediction level. Unlike the preceding statistical measures, SHAP examines the learned model directly, attributing a feature contribution value for each data instance.

All three analyses yielded consistently weak results. Kendall's τ coefficients were really small (between -0.19 and 0.03), indicating no monotonic association of the *verifier warning sum* feature with each metric. MI scores were similarly low, suggesting statistical near-independence between the feature and the metrics. SHAP

Table 4: Code comprehensibility prediction results per verifier. Model performance differences are shown: $\Delta = T - C$; C (Control setting): Syntactic + Developer features; T (Treatment setting): Syntactic + Developer + Semantic features; The semantic feature is the warning count of each individual verifier. A positive difference Δ is highlighted in Green. Statistically significant differences ($p < 0.05$) are in bold.

(a) $\Delta wF1$ results for each verifier, metric, and model

Verifier	Metric	NB	KNN	LR	MLP	RF	SVM
Infer	AU	0.006	-0.001	-0.002	-0.029	0.007	-0.003
	PBU	0.000	-0.003	0.000	-0.022	0.000	0.005
	$ABU_{50\%}$	-0.009	-0.009	-0.001	-0.030	-0.003	-0.017
	$BD_{50\%}$	0.001	-0.009	0.000	0.038	0.001	0.010
	RL	0.000	-0.003	0.000	-0.011	0.000	0.014
CF	AU	-0.005	-0.006	-0.003	-0.028	0.007	-0.004
	PBU	-0.002	-0.003	0.001	-0.022	0.001	**0.010**
	$ABU_{50\%}$	-0.003	-0.008	-0.002	-0.028	-0.005	-0.001
	$BD_{50\%}$	0.000	-0.011	-0.004	0.050	-0.003	**0.011**
	RL	**0.006**	-0.004	**0.008**	**0.002**	**0.005**	**0.022**
OJ	AU	-0.004	-0.006	-0.003	-0.027	0.003	0.002
	PBU	0.007	-0.024	0.003	-0.009	**0.008**	0.007
	$ABU_{50\%}$	0.001	-0.005	0.002	-0.026	-0.005	-0.003
	$BD_{50\%}$	-0.005	-0.012	-0.005	0.050	-0.006	0.003
	RL	0.002	0.000	0.004	-0.002	0.004	**0.015**
JaTyC	AU	0.000	0.000	-0.002	-0.029	**0.011**	-0.001
	PBU	0.000	-0.006	0.002	-0.002	-0.005	**0.014**
	$ABU_{50\%}$	0.003	-0.007	-0.003	-0.028	0.000	-0.001
	$BD_{50\%}$	**0.012**	-0.020	-0.014	0.040	-0.008	-0.001
	RL	**0.008**	-0.002	**0.006**	0.002	**0.014**	**0.019**

(b) $\Delta wAUC$ results for each verifier, metric, and model

Verifier	Metric	NB	KNN	LR	MLP	RF	SVM
Infer	AU	-0.006	-0.004	-0.002	-0.010	-0.001	-0.010
	PBU	0.000	0.000	-0.001	-0.005	0.000	-0.002
	$ABU_{50\%}$	**0.000**	0.000	0.000	**0.001**	0.000	0.000
	$BD_{50\%}$	**0.005**	0.002	0.004	-0.024	0.002	0.006
	RL	0.000	-0.001	0.000	0.003	-0.001	-0.001
CF	AU	-0.006	-0.010	0.000	-0.010	0.000	-0.008
	PBU	-0.005	-0.006	-0.001	-0.005	-0.002	0.001
	$ABU_{50\%}$	0.000	0.000	0.000	**0.001**	**0.000**	0.000
	$BD_{50\%}$	0.002	0.003	0.004	-0.028	**0.003**	0.004
	RL	**0.012**	0.002	**0.021**	**0.014**	**0.013**	**0.018**
OJ	AU	-0.005	-0.005	-0.002	-0.009	-0.004	-0.006
	PBU	-0.001	-0.002	0.000	-0.003	0.001	0.000
	$ABU_{50\%}$	0.000	0.000	0.000	**0.001**	0.000	0.000
	$BD_{50\%}$	0.004	-0.006	0.005	-0.029	0.002	0.006
	RL	0.000	-0.003	0.000	0.003	-0.001	-0.001
JaTyC	AU	-0.002	0.001	-0.004	-0.010	0.002	-0.006
	PBU	-0.005	0.002	0.000	-0.004	-0.001	0.002
	$ABU_{50\%}$	0.000	0.000	0.000	**0.001**	0.000	0.000
	$BD_{50\%}$	-0.007	-0.003	**0.008**	-0.029	**0.004**	**0.011**
	RL	**0.004**	-0.001	**0.007**	**0.012**	**0.017**	0.005

Note: For improved readability, individual results for both Control and Treatment settings are omitted but can be found in our replication package [22].

value analysis further confirmed this, with the feature exhibiting near-zero SHAP magnitudes, inconsistent directionality across data instances, and unstable importance rankings across classes and cross-validation folds. For AU classes 0 and 2, the SHAP values are approximately zero, and *verifier warning sum* is ranked last among the top 10% features (i.e., twelfth place). In contrast, for classes 1 and 3, the SHAP values are slightly higher but still remain close to zero; *verifier warning sum* is ranked fifth among the same 12 features, with SHAP values ranging between -0.02 and 0.02. This suggests each class instance in the metric is influenced differently inside the model when other features are present. The convergence of these analyses provides evidence that *verifier warning sum* carries no reliable or learnable signal for comprehensibility prediction, and its inclusion does not contribute meaningfully to model performance. Please refer to our replication package [22] for detailed results of these analyses, including the full set of Kendall's τ coefficients, MI scores, and SHAP value distributions for all features across all models and metrics.

4.3 Verifier-Specific Warning Analysis

To examine whether modeling verifier-specific semantic information yields better predictive performance than aggregating warnings across all tools, we conducted the same control–treatment comparison using the warning count of each verifier. In other words, rather than combining warnings from all four verifiers into a single aggregated feature, we treated each tool's warning count as an individual semantic feature to assess whether tool-specific signals improve prediction accuracy.

We focus our analysis in this subsection on the $wF1$ metric. According to the results presented in table 4, among the four tools, OpenJML (OJ) and Java Typestate Checker (JaTyC) exhibit at least one $wF1$ improvement across all six models and all five metrics. Specifically, for each metric, the treatment values obtained using OJ and JaTyC warnings outperform the control setting in at least one of the evaluated models, with $\Delta wF1$ ranging from **+0.001** to **+0.05**.

Similarly, PBU, $BD_{50\%}$, and RL are the comprehensibility metrics that benefit most from adding individual verifier warnings, showing $wF1$ improvements in at least two models whenever any individual verifier warning count is used as a feature (with one exception: when use warning count from Infer, PBU improves in only one treatment model), with $\Delta wF1$ scores ranging from **+0.001** to **+0.05**.

When examining the predictive power of individual verifiers, JaTyC and the Checker Framework (CF) yield the highest number of statistically significant improvements in $\Delta wF1$, whereas the Infer verifier yields none. Within these statistically significant combinations, the RL metric emerges as the most robust, driving improvements across multiple models (most notably NB, LR, and SVM) for both the CF and JaTyC verifiers. While these significant gains for the RL metric are largely mirrored in the ΔAUC results, ΔAUC also exposes a distinct trend absent from the $\Delta wF1$ data: a minor but statistically significant improvement for the $ABU_{50\%}$ metric when paired with the MLP model, which occurs consistently across all four verifiers.

However, no single verifier consistently outperforms the others across all metrics and models. This suggests that the predictive value of warnings depends on the specific categories of issues each tool detects in the code and how those issues relate to different dimensions of code comprehensibility. It is worth mentioning that, using individual verifier warnings from CF, OJ, and JaTyC leads

to improvements across almost all models, although the gains are minimal. This is explained by the fact that CF, OJ, and JaTyC issue comparatively more warnings (CF issues 51 and 70 warnings for DS_2 and DS_1, respectively. The corresponding numbers for OJ are 605 and 236, and for JaTyC, 327 and 522), suggesting they capture a broader range of semantic information, whereas Infer issues fewer warnings (0 for DS_2 and only 5 for DS_1), which may explain its smaller improvements.

When using individual verifier warning counts as semantic features, RF and SVM remain the most robust performers, achieving $\Delta wF1$ values of up to 0.022. RF leads to improvements across all the verifiers for AU and RL (except for Infer), although the improvements are minimal ($\Delta wF1$ **0.001** to **0.014**). SVM shows improvements for PBU and RL across all verifiers, with $\Delta wF1$ values ranging from **+0.005** to **+0.022**.

Overall, as observed with the *verifier warning sum*, incorporating individual verifier warnings produces limited but occasionally positive effects. In most cases, the improvements are modest, typically not exceeding $\Delta wF1 = +0.05$ (when using the MLP model with OpenJML and Checker Framework warnings to predict $BD_{50\%}$). These results confirm that the earlier conclusion derived from the aggregated warning sum also holds when individual verifier warnings are used as separate semantic features.

5 Threats to Validity

Construct Validity. We rely on human-judged comprehensibility metrics (*e.g.*, perceived understandability and readability) as proxies for the complex, multifaceted cognitive process of code comprehension. It is possible that these proxies do not fully capture all dimensions of code comprehensibility, such as the influence of domain knowledge or individual differences in cognitive styles. To minimise this, threat we used a wider range of comprehensibility proxies, including both subjective and objective measures, to capture different aspects of code comprehensibility.

We use the counts of verifier warnings as a proxy for the actual information processed by the verifiers. It is possible that warning counts do not perfectly encapsulate the semantic nuances that affect code complexity and human cognitive load. However, to ensure methodological robustness, we employed the same proxy metric successfully used and validated in prior work [28].

Internal Validity. The choice of models and their hyperparameters, our feature selection procedure, and our cross-validation methodology are validity threats. We also built our dataset of code features (section 3.1) from the definitions of Scalabrino *et al.* [68]. Some definitions were ambiguous, and we found discrepancies between the data generated by their code and the data provided in their replication package. We discarded a few code features and implemented the rest using the Java 8 Language Specification [2] to resolve ambiguities.

Although the datasets we used in this study are the largest available Java comprehensibility datasets used in prior work, the limited number of instances could impact the power of our statistical analyses and the ability of the models to learn patterns from the data. However, the *wAUC* results show that the models are indeed learning from the data, although not consistently across models and comprehensibility metrics.

External Validity. All the snippets from DS_1 and DS_2 datasets are written in Java. The results may not generalize to code written in other programming languages. To our best knowledge, comprehensibility datasets for other languages are scarce, or contain made-up code not representative of real-world code. In contrast, the selected datasets are extracted from OSS projects, thus representing real-life code. The majority of the study participants from whom both comprehensibility data were obtained in prior work are mostly students. Only DS_2 included some professional developers. The results may not generalize to broader populations of professional developers. Further, the results may not generalize to larger code snippets.

6 Implications, Conclusions, and Future work

Our results showed mostly no performance improvement of verifier-augmented ML models compared to baseline models that solely rely on syntactic and developer features, with only a small handful of cases exhibiting marginal and inconsistent gains.

These negative findings highlight **two important insights**:

(1) Developers construct a rich cognitive model when understanding code, one that is not fully captured by the syntactic, semantic, and developer-specific features used in this study, which are also commonly used in the current literature [13, 64, 68]. This suggests that additional, yet-unidentified factors play a substantial role in shaping human perceptions of code comprehensibility.
(2) Combining syntactic and developer features is sufficient for predicting code comprehensibility at the state-of-the-art accuracy level we obtained, which is consistent with the accuracy reported in prior studies [68]. This could mean that their combination may be enough to capture code complexity, that *none* of the features are useful to predict comprehensibility, or that the semantic feature we studied (verifier warning sum) does not fully capture nuanced aspects of code semantics and complexity.

Future work holds several promising avenues for researchers. Efforts should focus on conducting more comprehensive user studies to uncover new influential factors that could be used as model features. As warning counts do not improve mode performance significantly, this feature may not be an appropriate way to represent code complexity. Our future work will experiment with alternative representations, such as the types of warnings issued by the verifiers or the types of facts about the code that verifiers prove during code analysis, which can provide a more nuanced view of code semantics and complexity. For example, richer semantic representations from program analysis (*e.g.*, control-flow or data-flow relations) may yield superior results compared to warning counts. For practitioners, our findings suggest that existing machine learning models are not yet reliable enough for automated code comprehensibility estimation in real-world development. Before such tools can be confidently applied, further methodological advances and program semantic modeling will be necessary.

Acknowledgments

This work is supported in part by NSF grants 2414110, 2414111, and 2239107. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] 2024. K Fold Cross Validation. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html/.

[2] 2026. Java Language Specification. https://docs.oracle.com/javase/specs/jls/se8/html/index.html.

[3] 2026. SpotBugs. https://spotbugs.github.io/

[4] Amine Abbad-Andaloussi, Thierry Sorg, and Barbara Weber. 2022. Estimating Developers' Cognitive Load at a Fine-grained Level Using Eye-tracking Measures. In *Intl. Conf. on Prog. Compr. (ICPC)*. 111–121.

[5] Herve Abdi, Lynne J Williams, et al. 2010. Normalizing data. *Encyclopedia of research design* 1 (2010), 935–938.

[6] Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. 2019. Syntax, predicates, idioms — what really affects code complexity? *Emp. Soft. Eng.* 24, 1 (2019), 287–328.

[7] Vard Antinyan. 2020. Evaluating Essential and Accidental Code Complexity Triggers by Practitioners' Perception. *IEEE Soft.* 37, 6 (2020), 86–93.

[8] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. 2017. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Emp. Soft. Eng.* 22, 6 (2017), 3057–3087.

[9] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Emp. Soft. Eng.* 18, 2 (2013), 219–276.

[10] Jürgen Börstler, Kwabena E Bennin, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, Bonnie MacKellar, Rodrigo Duran, Harald Störrle, Daniel Toll, et al. 2023. Developers talking about code quality. *Empirical Software Engineering* 28, 6 (2023), 128.

[11] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[12] Frederick Brooks and H Kugler. 1987. *No silver bullet.* April.

[13] Raymond Buse and Westley Weimer. 2009. Learning a metric for code readability. *Trans. on Soft. Eng. (TSE)* 36, 4 (2009), 546–558.

[14] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symp.* Springer, 3–11.

[15] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Principles of Programming Languages (POPL)*. 289–300.

[16] Gavin C Cawley and Nicola LC Talbot. 2010. On over-fitting in model selection and subsequent selection bias in performance evaluation. *The Journal of Machine Learning Research* 11 (2010), 2079–2107.

[17] S le Cessie and JC Van Houwelingen. 1992. Ridge estimators in logistic regression. *Journal of the Royal Statistical Society Series C: Applied Statistics* 41, 1 (1992), 191–201.

[18] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[19] Thomas M Cover. 1999. *Elements of information theory.* John Wiley & Sons.

[20] Carlos Dantas, Adriano Rocha, and Marcelo Maia. 2023. Assessing the readability of chatgpt code snippet recommendations: A comparative study. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. 283–292.

[21] Nadeeshan De Silva, Martin Kellogg, and Oscar Chaparro. 2025. Relative Code Comprehensibility Prediction. *arXiv preprint arXiv:2510.03474* (2025).

[22] Nadeeshan De Silva, Martin Kellogg, and Oscar Chaparro. 2026. Online replication package. https://github.com/sea-lab-wm/warning-comprehensibility.

[23] Pablo Del Moral, Sławomir Nowaczyk, and Sepideh Pashami. 2022. Why is multiclass classification hard? *IEEE Access* 10 (2022), 80448–80462.

[24] Jonathan Dorn. 2012. A general software readability model. *MCS Thesis available from (http://www. cs. virginia. edu/weimer/students/dorn-mcs-paper. pdf)* 5 (2012), 11–14.

[25] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. 2002. Does code decay? assessing the evidence from change management data. *IEEE transactions on software engineering* 27, 1 (2002), 1–12.

[26] Janet Feigenspan, Sven Apel, Jorg Liebig, and Christian Kastner. 2011. Exploring Software Measures to Assess Program Comprehension. In *Intl. Symp. on Emp. Soft. Eng. and Meas. (ESEM)*. 127–136.

[27] Dror G Feitelson. 2021. Considerations and pitfalls in controlled experiments on code comprehension. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 106–117.

[28] Kobi Feldman, Martin Kellogg, and Oscar Chaparro. 2023. On the Relationship between Code Verifiability and Understandability. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 211–223.

[29] Rudolf Flesch. 1979. *How to write plain English: A book for lawyers and consumers.* Vol. 76026225. Harper & Row New York.

[30] George Forman and Martin Scholz. 2010. Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement. *Acm Sigkdd Explorations Newsletter* 12, 1 (2010), 49–57.

[31] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using psycho-physiological measures to assess task difficulty in software development. In *Intl. Conf. on Soft. Eng. (ICSE)*. 402–413.

[32] Davide Fucci, Daniela Girardi, Nicole Novielli, Luigi Quaranta, and Filippo Lanubile. 2019. A Replication Study on Code Comprehension and Expertise using Lightweight Biometric Sensors. In *Intl. Conf. on Prog. Compr. (ICPC)*. 311–322.

[33] Amy GrabNGoInfo. 2022. Support Vector Machine (SVM) Hyperparameter Tuning In Python. https://medium.com/grabngoinfo/support-vector-machine-svm-hyperparameter-tuning-in-python-a65586289bcb/

[34] Maurice H. Halstead. 1977. *Elements of Soft. Science.* Elsevier.

[35] James A Hanley and Barbara J McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* 143, 1 (1982), 29–36.

[36] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. 1998. Support vector machines. *IEEE Intelligent Systems and their applications* 13, 4 (1998), 18–28.

[37] Mohammad Hossin and Md Nasir Sulaiman. 2015. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process* 5, 2 (2015), 1.

[38] Ahmad Jbara and Dror G. Feitelson. 2017. How programmers read regular code: a controlled experiment using eye tracking. *Emp. Soft. Eng.* 22, 3 (2017), 1440–1477.

[39] Cem Kaner, Senior Member, and Walter P. Bond. 2004. Software Engineering Metrics: What Do They Measure and How Do We Know?. In *Intl. Soft. Metrics Symp. (METRICS)*.

[40] Zachary Karas, Aakash Bansal, Yifan Zhang, Toby Li, Collin McMillan, and Yu Huang. 2024. A tale of two comprehensions? analyzing student programmer attention during code summarization. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–37.

[41] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1-2 (1938), 81–93.

[42] Maurice G. Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.

[43] Amy J Ko and Brad A Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing* 16, 1-2 (2005), 41–84.

[44] Luigi Lavazza, Abedallah Zaid Abualkishik, Geng Liu, and Sandro Morasca. 2023. An empirical evaluation of the "Cognitive Complexity" measure as a predictor of code understandability. *Journal of Systems and Software* 197 (2023), 111561.

[45] Luigi Lavazza, Sandro Morasca, and Marco Gatto. 2023. An empirical study on software understandability and its dependence on code characteristics. *Empirical Software Engineering* 28, 6 (2023), 155.

[46] Gary T Leavens, Albert L Baker, and Clyde Ruby. 1998. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA 1998)*. Citeseer, 404–420.

[47] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

[48] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *Trans. on Soft. Eng. and Methodology (TSEM)* 23, 4 (2014), 1–37.

[49] T.J. McCabe. 1976. A Complexity Measure. *Trans. on Soft. Eng. (TSE)* SE-2, 4 (1976), 308–320.

[50] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[51] Qing Mi, Yiqun Hao, Liwei Ou, and Wei Ma. 2022. Towards using visual, semantic and structural features to improve code readability classification. *Journal of Systems and Software* 193 (2022), 111454.

[52] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *Intl. Conf. on Prog. Compr. (ICPC)*. 25–35.

[53] João Mota, Marco Giunti, and António Ravara. 2021. Java typestate checker. In *Intl. Conf. on Coord. Lang. and Models*. Springer, 121–133.

[54] Gireen Naidu, Tranos Zuva, and Elias Mmbongeni Sibanda. 2023. A review of evaluation metrics in machine learning algorithms. In *Computer science on-line conference*. Springer, 15–25.

[55] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *Intl. Workshop on Computer Science Logic*. Springer, 1–19.

[56] Paul Oman and Jack Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*. IEEE, 337–344.

[57] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. 2008. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 201–212.

[58] Kang-il Park, Jack Johnson, Cole S Peterson, Nishitha Yedla, Isaac Baysinger, Jairo Aponte, and Bonita Sharif. 2024. An eye tracking study assessing source code readability rules for program comprehension. *Empirical Software Engineering* 29, 6 (2024), 160.

[59] Abhi Patel, Kazi Zakia Sultana, and Bharath K. Samanthula. 2024. A Comparative Analysis between AI Generated Code and Human Written Code: A Preliminary Study. In *2024 IEEE International Conference on Big Data (BigData)*. 7521–7529. https://doi.org/10.1109/BigData62323.2024.10825958

[60] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program comprehension and code complexity metrics: An fMRI study. In *Intl. Conf. on Soft. Eng. (ICSE)*. 524–536.

[61] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What Drives the Reading Order of Programmers? An Eye Tracking Study. In *Intl. Conf. on Prog. Compr. (ICPC)*. 342–353.

[62] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kästner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2018. A look into programmers' heads. *Trans. on Soft. Eng. (TSE)* 46, 4 (2018), 442–462.

[63] Leif E Peterson. 2009. K-nearest neighbor. *Scholarpedia* 4, 2 (2009), 1883.

[64] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*. 73–82.

[65] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2021. Reflections on: A Simpler Model of Software Readability. *ACM SIGSOFT Soft. Eng. Notes* 46, 3 (2021), 30–32.

[66] Hassan Ramchoun, Youssef Ghanou, Mohamed Ettaouil, and Mohammed Amine Janati Idrissi. 2016. Multilayer perceptron: Architecture optimization and training. (2016).

[67] Steven J Rigatti. 2017. Random forest. *Journal of Insurance Medicine* 47, 1 (2017), 31–39.

[68] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2019. Automatically assessing code understandability. *Trans. on Soft. Eng. (TSE)* 47, 3 (2019), 595–613.

[69] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30, 6 (2018), e1958.

[70] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2016. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

[71] Agnia Sergeyuk, Olga Lvova, Sergey Titov, Anastasiia Serova, Farid Bagirov, Evgeniia Kirillova, and Timofey Bryksin. 2024. Reassessing java code readability models with a human-centered approach. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 225–235.

[72] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[73] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *Intl. Conf. on Soft. Analysis, Evolution, and ReEng. (SANER)*, Vol. 5. 13–20.

[74] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Intl. Conf. on Soft. Eng. (ICSE)*. 378–389.

[75] Dag IK Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, N-K Liborg, and Anette C Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE transactions on software engineering* 31, 9 (2005), 733–753.

[76] Ryo SOGA, Takatomi KUBO, Takashi ISHIO, Yuna NUNOMURA, Takahiro KINOSHITA, Hideyuki KANUKA, and Kenichi MATSUMOTO. 2025. Your heart foretells your performance: Analysis of pre-task heart rate in program comprehension tasks. *IEICE Transactions on Information and Systems* (2025).

[77] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. 1974. Structured design. *IBM systems journal* 13, 2 (1974), 115–139.

[78] Robert E Strom and Shaula Yemini. 2012. Typestate: A programming language concept for enhancing software reliability. *IEEE transactions on software engineering* 1 (2012), 157–171.

[79] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? an exploratory study in industry. In *Symp. on the Found. of Soft. Eng. (FSE)*. 1–11.

[80] The OCaml Developers. 2026. OCaml. https://ocaml.org/.

[81] The OpenJML Developers. 2022. OpenJML. https://www.openjml.org/.

[82] Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. 2018. "Automatically assessing code understandability" reanalyzed: combined metrics matter. In *Intl. Conf. on Mining Soft. Repositories (MSR)*. 314–318.

[83] Geoffrey I Webb, Eamonn Keogh, and Risto Miikkulainen. 2010. Naïve Bayes. *Encyclopedia of machine learning* 15, 1 (2010), 713–714.

[84] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2023. 40 years of designing code comprehension experiments: A systematic mapping study. *Comput. Surveys* 56, 4 (2023), 1–42.

[85] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *Trans. on Soft. Eng. (TSE)* 44, 10 (2018), 951–976.

[86] Martin K.-C. Yeh, Dan Gopstein, Yu Yan, and Yanyan Zhuang. 2017. Detecting and comparing brain activity in short program comprehension using EEG. In *Frontiers in Education Conf. (FIE)*. 1–5.

[87] H. Zuse. 1993. Criteria for program comprehension derived from software complexity metrics. In *Workshop on Prog. Compr.* 8–16.