electronics

MDPI

Article

On the Use of an Extreme Learning Machine for GitHub Repository Popularity Prediction Based on Static Software Metrics

Emin Borandağ, Fatih Yücalar *, Yusuf Özçevik and Osman Altay

Department of Software Engineering, Manisa Celal Bayar University, 45400 Manisa, Turkey; emin.borandag@cbu.edu.tr (E.B.); yusuf.ozcevik@cbu.edu.tr (Y.Ö.); osman.altay@cbu.edu.tr (O.A.)
* Correspondence: fatih.yucalar@cbu.edu.tr

Abstract

Software data is widely used to predict attributes of software systems; however, obtaining reliable datasets from commercial companies remains challenging due to confidentiality constraints. GitHub has emerged as a data source, offering access to diverse applications and development statistics. Nevertheless, concerns about the reliability and representativeness of public repositories persist. Star count is a widely accepted indicator of repository popularity, and existing studies mainly rely on time-dependent platform metrics. In this study, we propose using static software metrics extracted from source code, along with GitHub statistics. To our knowledge, this study is among the first to use ELM for popularity prediction with static software metrics. Repositories from different application domains are selected to ensure dataset diversity and representativeness. An automated tool has been developed to collect data via the GitHub API and SourceMonitor CLI. In addition, several baseline machine learning models, including LR, SVM, RF, and LSBoost, are evaluated for comparison. Experimental results show that ELM achieves competitive performance across datasets. In terms of R^2 scores, ELM performs best in four datasets, RF in three, and LR in one. These results indicate that ELM is an effective method for popularity prediction and highlight the potential of incorporating static software metrics into GitHub-based predictive modeling.

Keywords: static software metrics; popularity prediction; GitHub; extreme learning machine

1. Introduction

Software developers have recently shown increasing interest in conducting data-driven analyses of software projects under development in order to improve various aspects of software repositories, including software quality requirements [1]. In this context, studies such as [2,3] focus on recommendation systems for identifying influential repositories contributed by multiple developers, while [4,5] investigate developer profiles for popularity analysis. It is also observed that many developers consider the popularity of a repository before deciding to use or contribute to it [6]. Therefore, software influence analysis and popularity prediction, based on indicators such as star count, fork count, and follower count, have become prominent research areas for learning-based models [7–9]. GitHub is widely utilized as one of the most popular platforms for hosting software repositories, containing numerous public repositories across diverse software technologies [10–12]. Existing studies attempt to predict software influence, popularity, and maintenance using metrics provided by the GitHub Application Programming Interface (API) [13,14]. GitHub also enables

Academic Editor: George Angelos Papadopoulos

Received: 7 April 2026
Revised: 30 April 2026
Accepted: 11 May 2026
Published: 14 May 2026

users to label projects with topics [15] and provides access to repository-level data such as commits, stars, and contributors [16]. Although these approaches achieve satisfactory predictive performance, their scope is inherently limited by the finite and predominantly time-dependent nature of GitHub-specific metrics.

Beyond platform-based indicators, the source code itself contains valuable information in the form of static software metrics, which can be extracted using dedicated metric collection tools [17,18]. These metrics can be directly computed from source code without requiring compilation or execution [19], making software repositories a rich and practical source for their extraction [20]. Static software metrics have been extensively studied in the literature, particularly for software quality assessment and defect prediction, using both rule-based and learning-based approaches [21–24]. Although static software metrics have been widely used in defect prediction studies, their applicability to popularity prediction is not straightforward. Defect prediction primarily focuses on identifying fault-prone components based on internal code quality characteristics, whereas popularity prediction is concerned with external, community-driven indicators such as GitHub stars. Unlike defects, repository popularity is influenced not only by code quality but also by factors such as project visibility, usability, and developer engagement. Therefore, it cannot be assumed that static metrics used for defect prediction would directly translate to popularity prediction tasks. In this study, we aim to bridge this gap by investigating the extent to which static software metrics can contribute to predicting repository popularity and by empirically evaluating their effectiveness in this new context. Accordingly, the following research question is addressed in this study: *Can machine learning models accurately predict repository popularity using static software metrics?*

To address this gap, this study proposes a novel and integrative framework that combines static software metrics with GitHub-specific indicators for repository popularity prediction. A dataset is constructed by merging GitHub metrics obtained via the GitHub API with static software metrics extracted using the SourceMonitor command-line interface (CLI). To ensure diversity and representativeness, repositories are systematically selected from different types of applications based on popular GitHub topics. For this purpose, a Windows-based console application is developed in C# to automate the data collection and preprocessing pipeline.

Although the Extreme Learning Machine (ELM) has been widely applied to various prediction tasks in software engineering and data mining, its use in GitHub repository popularity prediction has not been extensively explored. In particular, existing studies on GitHub repository popularity prediction predominantly rely on time-dependent platform metrics, such as temporal star growth, fork evolution, and repository activity patterns over time. To the best of our knowledge, no prior study has directly employed ELM for GitHub repository popularity prediction based on static software metrics. Therefore, this study aims to fill this gap by investigating the effectiveness of ELM in this context.

ELM is adopted to model the relationship between input features and repository star counts, offering a fast and efficient learning mechanism based on a single-layer feedforward neural network. The implementation is carried out in MATLAB R2024b, and the predictive performance of the proposed model is evaluated using standard performance metrics. The experimental results demonstrate that incorporating static software metrics significantly enhances prediction capability, highlighting their effectiveness as complementary indicators for popularity analysis. The main contributions of this study can be summarized as follows:

- The construction of a diverse and comprehensive dataset that integrates GitHub metrics with static software metrics collected from repositories belonging to different application domains;

- The development of an automated and modular data collection framework that utilizes the GitHub API and SourceMonitor CLI is developed to ensure reproducibility and scalability, enabling the construction of diverse datasets across different application domains;
- The introduction and evaluation of the ELM method for GitHub repository popularity prediction, providing a novel methodological perspective;
- A systematic comparison of the proposed ELM-based approach with several baseline machine learning methods, including Linear Regression (LR), Support Vector Machine (SVM), Random Forest (RF), and Least Squares Boosting (LSBoost), to evaluate its relative predictive performance across different datasets;
- Ablation analysis and multi-model comparisons, to systematically analyze the individual and combined effects of GitHub-based and static software metrics.

The remainder of this paper is organized as follows: Section 2 presents a comprehensive literature review on GitHub metrics and static software metrics. Section 3 describes the dataset construction methodology, while Section 4 introduces the proposed learning-based model. Section 5 presents and discusses the experimental results, and finally, Section 6 concludes the paper and outlines future research directions.

2. Literature Review

The existing literature employs a wide range of abbreviations for static software metrics, which are predominantly utilized in software defect prediction and quality assessment studies. However, the inconsistent use of these abbreviations across different studies often leads to ambiguity and reduces the comparability of results. To address this issue and ensure clarity, a comprehensive nomenclature of commonly used static software metrics is presented in Table 1. It can be observed that multiple abbreviations are frequently used to represent the same metric in different studies, which may complicate interpretation. Therefore, the abbreviations are systematically organized in alphabetical order, and their corresponding meanings are explicitly defined. In addition, the table highlights the specific abbreviations adopted in the reviewed studies, thereby providing a standardized reference framework that facilitates consistency and improves the readability of the literature review.

Table 1. Nomenclature of common static software metrics.

Abbreviation	Meaning
AMC	Average method complexity
AVG_CC	Average McCabe's complexity
CC	Cyclomatic complexity
CD	Comment density
CLOC	Comment lines of code
CMC	Class method complexity
DIT	Depth of the inheritance tree
LOC	Lines of code
LLOC	Logical lines of code
MAX_CC	Maximum McCabe's complexity
NFi, NOF	Number of files
NOA	Number of attributes
NOC	Number of classes
NOI	Number of interfaces
NM, NOM	Number of methods
NV	Number of variables per class
WMC	The number of methods per class

In the last decade, several studies have extensively explored both learning-based and statistical approaches by leveraging data obtained from software repositories as well as static software metrics. In particular, GitHub repositories are widely utilized to construct datasets and develop predictive models for software-related tasks such as defect prediction, influence analysis, and popularity estimation [2,6,7,21,22,24–32]. While GitHub-based metrics such as star count, fork count, and commit activity are widely used for analyzing repository influence and popularity trends [2,6,7,25,27,29,31], static software metrics are primarily investigated in the context of software quality evaluation and defect prediction [21,22,24,26,28,30,32]. Despite this distinction, both types of metrics provide complementary perspectives on software systems. A comprehensive summary of the existing literature is presented in Table 2, where the selected works are organized chronologically to illustrate the evolution and diversity of research in this domain. The table outlines the publication year, the purpose, methodology, and key attributes of each study, enabling a structured comparison of existing approaches and highlighting the diversity in data sources, modeling techniques, and application domains.

Table 2. A summary of learning-based studies in the last decade that utilize social repositories and static software metrics.

Reference	Year	Purpose	Methodology	Key Attributes
[7]	2021	Analyzing metadata factors driving dataset and software popularity.	Artificial neural networks-based popularity prediction combined with statistical analysis	GitHub metrics, including forks, stars, contributors, releases, and project size growth
[21]	2022	Software fault prediction	A deep learning architecture for software fault prediction	Static software metrics including AVG_CC, AMC, DIT, LOC, MAX_CC, WMC
[24]	2022	Analyzing ML-based defect prediction in mobile applications	Systematic Literature Review (SLR) of 47 studies	Static software metrics, including CMC, DIT, NOC, NM, NV, WMC
[2]	2023	Analyzing the impact of COVID-19 on GitHub development trends	Time-series trend analysis and stationarity tests of GitHub events	GitHub metrics, including forks, issues, comments, commits, and pushes
[22]	2023	Comparing ML and DL for software fault prediction	ML and DL classifiers (including RNNs) on multiple software fault datasets	Static software metrics, including CBO, DIT, LOC, WMC, and historical change metrics.
[25]	2024	Improved software defect prediction against imbalanced datasets	Bi-LSTM model combined with oversampling (SMOTE/random) on benchmark datasets.	Static software metrics including AMC, AVG_CC, CBO, LOC, MAX_CC, NOC
[26]	2024	Analyzing GitHub activity dynamics and influencing factors	Discrete-Time Markov Chains, and model checking with probabilistic Computation Tree Logic	GitHub metrics, including repository states, pull requests, and branches
[27]	2024	Dataset construction and method-level bug prediction for practical use	Empirical evaluation and comparison on multiple datasets, including the one constructed with improvement strategies	Static software metrics, including method-level CC, HCPL, LOC, NOM
[28]	2024	Improved method-level bug prediction with a dependency-aware approach	Machine learning (Random Forest) with a dependency tracking algorithm	Static software metrics including method-level metrics and code dependencies CC, CD, LOC, NL, HCPL, NUMPAR

Table 2. *Cont.*

Reference	Year	Purpose	Methodology	Key Attributes
[6]	2025	Comparing software quality, identifying differences, and best practices	Dataset creation from GitHub, metric computation, and in-depth code examination	Static software metrics, including complexity, coupling, size, cohesion, and maintainability metrics
[29]	2025	Analyzing fork diversity (fork entropy) and its impact on public GitHub repository contributions	Empirical analysis, correlation study, ARMAX, and Transformer-based prediction models	GitHub metrics, including fork count, pull requests, bugs, and historical contribution data
[30]	2025	Predicting design-stage defects using ML/DL and software metrics	Dataset creation from class diagrams, ML/DL classification, cross-dataset evaluation	Static software metrics including size-inharitance-coupling metrics DIT, Nesting, NOC, NOA,
[31]	2026	Improving test-case prioritization using multimodal and explainable AI	Deep learning–based multimodal framework jointly uses bug reports, source code changes, and test metadata with SHAP explanations	GitHub metrics, including lines added, lines deleted, the number of files modified, and the entropy of change in a commit
[32]	2026	Improving defect prediction using deep learning with instance similarity	Image-based DL model with custom loss function (ISRL)	Static software metrics, including LOC, CC, and other project or language-specific metrics

It is evident from the existing literature that GitHub repositories are extensively used to obtain software data and construct representative datasets. However, most of the datasets either contain only static software metrics collected by metrics gathering tools or only git metrics obtained through the GitHub API when the literature given in Table 2 is examined. The ones using static software metrics focus on software defects, whereas the others utilizing GitHub metrics investigate the influence and popularity. Only [29] includes both types of metrics, but this time the study focuses only on bug prediction by ignoring the popularity analysis. Moreover, it is also inferred that complexity and size-related static software metrics are utilized for defect prediction models, whereas fork count and star count GitHub metrics are used for influence analysis and popularity prediction. To the best of our knowledge, there is no comprehensive study that constructs an automated dataset integrating both GitHub metrics and static software metrics specifically for repository popularity prediction. Furthermore, it is hypothesized that there exists a meaningful relationship between different types of metrics and that static software metrics can contribute to the impact analysis typically performed using GitHub metrics. In other words, we expect to infer information about software popularity by utilizing static software metrics since they are used for retrieving information about software defects by implication of the quality. Thus, we offer to investigate the effect of the inclusion of static software metrics for the learning model of star count-based popularity analysis of GitHub repositories. Accordingly, we conducted a thorough analysis of ELM on the datasets constructed in the study and discussed the evaluation results.

3. Data Acquisition

Traditional approaches to software data acquisition often involve labor-intensive processes, regardless of whether the data source consists of public repositories or static software metrics. Such manual procedures are not only time-consuming but also prone to human errors, which may negatively affect the reliability and reproducibility of learning-

based studies [33]. Therefore, minimizing human intervention in the data acquisition pipeline has become an important requirement in recent research. In this study, a fully automated data acquisition process is proposed to collect both GitHub-based metrics and static software metrics. For this purpose, the GitHub REST API [13,14,34] is utilized to retrieve repository-level information, while the SourceMonitor command-line interface (CLI) tool [35,36] is employed to compute static software metrics directly from source code. These tools are selected due to their practicality, scalability, and platform-independent operation, enabling efficient data collection without manual effort. As a result, a novel dataset is constructed by integrating GitHub metrics with static software metrics in a unified framework. This automated pipeline not only improves data reliability but also ensures consistency and reproducibility. The resulting dataset provides a comprehensive basis for investigating the relationship between source code characteristics and repository popularity, thereby supporting the development of more effective learning-based prediction models.

3.1. Software Data Acquisition Tool

In this study, a software data acquisition tool is developed to automatically construct datasets for subsequent analysis. The development environment and application stack of the proposed tool are illustrated in Figure 1. The development environment is based on Windows 10 Pro (x64) as the operating system, Visual Studio 2022 as the integrated development environment, and .net framework 7.0 as the runtime platform. The application stack is implemented as a C# console application. The GitHub REST API (version 2022-11-28) is utilized to retrieve repository-related information, while the SourceMonitor executable is integrated into the system to compute static software metrics via its command-line interface (CLI) on locally cloned repositories.

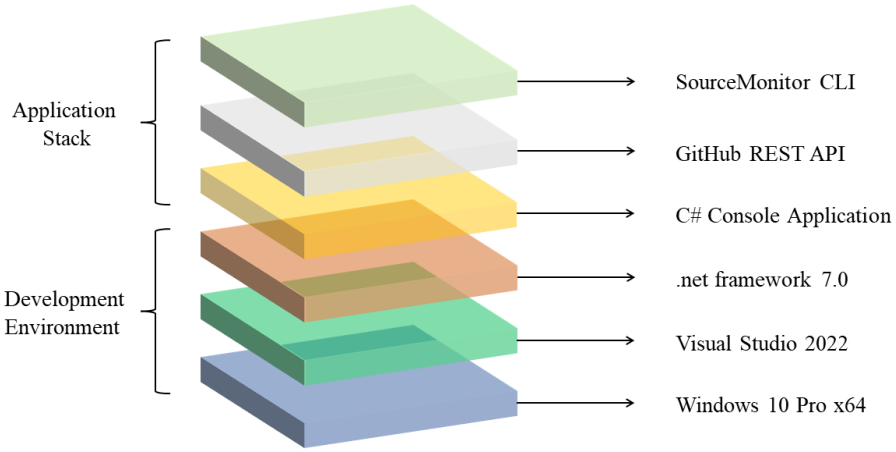

Figure 1. Architecture of the proposed software data acquisition tool.

The interaction between the components of the data acquisition tool is presented in Figure 2. The tool accepts the repository topic and programming language as input parameters. Based on these inputs, the GitHub REST API is queried using a chunked and batch-based strategy to mitigate API rate-limiting constraints, enabling incremental retrieval of repository metadata while respecting request limitations. The API returns information for repositories that may include multiple topics and programming languages, providing data such as creation date, last update date, and GitHub-specific metrics, including star count, fork count, and project size. Subsequently, each retrieved repository is cloned to local storage and processed using the SourceMonitor CLI to extract static software metrics. During this process, repositories are selected based on predefined topic categories and programming language constraints using the GitHub REST API. No minimum star count, popularity threshold, or activity-based filtering criteria are applied during the ini-

tial selection process. This approach was adopted to avoid bias toward highly popular repositories and to ensure diversity in repository characteristics. Following data collection, repositories with missing source code or those that cannot be successfully processed due to extraction or build errors were removed to ensure data quality and consistency for analysis. The output of the SourceMonitor tool is generated in XML format, containing the computed metric values. These static software metrics are then integrated with GitHub metrics, and the combined data is exported to a CSV file. As a result, a unified dataset comprising both GitHub metrics and static software metrics is obtained in an automated manner. After the filtering process, the final dataset consists of 3377 unique GitHub repositories.

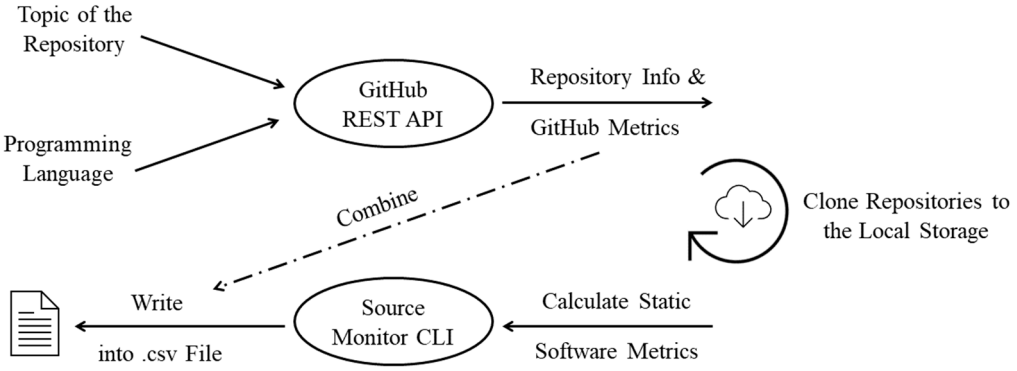

Figure 2. Workflow of the software data acquisition process.

3.2. Dataset Properties

The list of metrics, their descriptions, and their corresponding data sources in the constructed dataset are summarized in Table 3. The dataset includes three metrics obtained from the GitHub API and eleven metrics computed using the SourceMonitor CLI. Although a larger number of attributes can be extracted from both sources, only those metrics that are relevant to popularity analysis based on insights from the literature are selected. Furthermore, the selected GitHub metrics are consistently available across repositories, while the chosen static software metrics can be reliably computed for various types of software projects using metric extraction tools. Therefore, the resulting dataset provides a novel and comprehensive representation that extends beyond traditional GitHub-based datasets by incorporating intrinsic source code-level characteristics.

Table 3. Attributes included in the dataset, their descriptions, and data sources.

Attribute	Description	Acquisition Source
Star Count	The number of stars voted by GitHub users/developers	GitHub
Fork Count	The number of forks of a repository	
Project Size	The size of the projects in KB	
Lines	The total number of lines	SourceMonitor CLI
Statements	The number of statements	
Percent Comment Lines	The percentage of comment lines	
Percent Documentation Lines	The percentage of documentation lines	
Classes, Interfaces, Structs	The number of units, i.e., classes, interfaces, or structs, depending on the programming language	
Methods per Class	The number of methods per class	
Statements per Method	The number of statements per method	
Maximum Complexity	The maximum complexity measure of software units	
Average Complexity	The average complexity measure of software units	
Maximum Block Depth	The maximum block depth measure of software units	
Average Block Depth	The average block depth measure of software units	

Descriptive statistics of the dataset attributes are presented in Table 4, including the mean, standard deviation, median, minimum, and maximum values for each metric. These statistics provide an overview of the data distribution and highlight the variability across different software projects. Such variations are taken into account during the learning process and are also considered in the evaluation of the prediction performance of the ELM model, as the attribute values span a wide range due to the diversity of the collected repositories.

Table 4. Statistical summary of dataset attributes.

	Mean	Std. Dev.	Median	Min.	Max.
Star Count	664.76	1878.88	65	12	30,352
Fork Count	164.60	673.46	23	0	25,082
Project Size	33,746.43	135,286.75	2344	4	2,567,842
Lines	39,032.07	269,359.40	4018	5	10,590,806
Statements	16,565.04	110,514.26	1895	2	3,937,322
Percent Comment Lines	2.43	4.18	0.2	0	66.5
Percent Documentation Lines	3.57	6.82	0	0	67.3
Classes, Interfaces, Structs	381.84	2382.76	56	0	98,094
Methods per Class	4.56	3.35	3.99	0	59.91
Statements per Method	5.22	5.65	4.48	0	226.5
Maximum Complexity	42.61	621.59	15	0	35,864

4. Proposed Methodology

The proposed methodology aims to predict the popularity of software repositories by employing a learning-based model on the constructed dataset. In this context, the GitHub star count is considered the target variable representing repository popularity. The dataset, which integrates both GitHub-based metrics and static software metrics, serves as the input to the prediction model. To model the relationship between these features and repository popularity, the ELM method is utilized. The selection of ELM is motivated by its fast learning capability and effectiveness in handling nonlinear relationships, which makes it suitable for the proposed prediction task. Prior to model training, a data preprocessing stage is applied, in which normalization is performed to ensure that all features contribute proportionally to the learning process. Following the training phase, the performance of the model is evaluated using a set of standard evaluation metrics. These metrics provide a quantitative basis for assessing prediction accuracy and enable a comprehensive interpretation of the model's effectiveness. The overall framework of the proposed methodology thus integrates data preprocessing, model training, and performance evaluation stages to systematically investigate the relationship between software metrics and repository popularity.

4.1. Extreme Learning Machine

Feedforward neural networks have been widely employed for classification and regression tasks across various domains due to their strong generalization capability [37]. However, conventional training approaches for artificial neural networks often require significant computational time, as they iteratively update all network parameters, including weights and biases across multiple layers. This dependency among parameters leads to slow convergence and increases the complexity of the learning process. In contrast, the ELM offers a more efficient alternative for training single-layer feedforward neural networks. In the ELM framework, the input weights and hidden layer biases are randomly assigned and remain fixed during training, eliminating the need for iterative parameter tuning. As a result, the learning process is significantly accelerated compared to traditional gradient-based methods. Conventional approaches that rely on gradient descent are often computationally expensive and

may suffer from issues such as slow convergence and susceptibility to local minima, requiring multiple iterations to achieve satisfactory performance [37]. By randomly determining the hidden-layer parameters, the feedforward network can be reformulated as a linear system. Consequently, the output weights can be analytically computed using a generalized inverse of the hidden layer output matrix. This closed-form solution enables rapid training while maintaining strong generalization performance. Owing to these characteristics, ELM has been demonstrated to be both computationally efficient and effective compared to conventional artificial neural network training methods.

4.1.1. Gradient-Based Solutions

Gradient-based solutions have historically been employed to train single-hidden-layer feedforward neural networks by iteratively optimizing network parameters, including input weights, biases, and output weights [38]. In this framework, the learning process aims to minimize the discrepancy between the predicted outputs and the target values $\widetilde{w}_i$, $\widetilde{b}_i$, $\widetilde{\beta}$,$(i = 1, \ldots, \widetilde{N})$, as formulated in Equation (1):

$$\|H(\widetilde{w}_1, \ldots, \widetilde{w}_{\widetilde{N}}, \widetilde{b}_1, \ldots, \widetilde{b}_N)\beta - T\| = \min w_i b_i \beta \, \|H(\widetilde{w}_1, \ldots, \widetilde{w}_{\widetilde{N}}, \widetilde{b}_1, \ldots, \widetilde{b}_N)\beta - T\| \quad (1)$$

The equation above relates to the minimal value of the cost function. Specifically, the objective is to minimize the cost function defined over the hidden layer output matrix H, as expressed in Equation (2), the error is computed as the sum of squared differences between the network output and the target values.

$$E = \sum_{j=1}^{N} \sum_{i=1}^{\widetilde{N}} \left(\beta_i g\left(w_i \times x_j + b_i\right) - t_j\right)^2 \quad (2)$$

In gradient-based learning, the optimization process is typically performed using iterative update rules, as shown in Equation (3), where the parameter vector is updated based on the gradient of the cost function with respect to the model parameters, and n denotes the learning rate.

$$W_k = W_{k-1} - n\frac{\partial E(W)}{\partial W} \quad (3)$$

Among these methods, the backpropagation algorithm is the most widely used technique, which propagates the error gradients from the output layer to the input layer to adjust the network parameters efficiently [37]. Despite their widespread use, gradient-based approaches exhibit several limitations:

- First, the convergence speed is highly sensitive to the choice of the learning rate; a small learning rate leads to slow convergence, whereas a large learning rate may cause instability or divergence;
- Second, these methods are prone to getting trapped in local minima, which prevents the model from reaching the global optimum;
- Third, an artificial neural network might be overtrained and exhibit poor generalization performance when trained using the back-propagation learning process. Therefore, the approach for reducing the cost function must include legitimate and acceptable halting methods;
- Finally, the iterative nature of gradient-based optimization often leads to high computational cost, making these methods less suitable for large-scale or time-sensitive applications.

In the ELM algorithm suggested to overcome the aforementioned problems of gradient-based algorithms, these concerns can be avoided, and a more efficient learning scheme for feedforward neural networks is created.

4.1.2. Least Squares Norm

In contrast to conventional function approximation approaches, which require iterative adjustment of input weights and hidden layer biases, the ELM framework allows these parameters to be randomly assigned and fixed, provided that the activation function is sufficiently nonlinear. Under this assumption, the hidden layer output matrix H remains unchanged during training, and the learning problem reduces to solving a linear system of equations of the form $H\beta = T$, where β represents the output weights and T denotes the target matrix. The optimal solution can be obtained using the least squares norm $\hat{\beta}$, as formulated in Equation (4).

$$\left\|H\left(w_1,\ldots,w_{\tilde{N}},b_1,\ldots,n_{\hat{N}}\right)\hat{\beta}-T\right\| = \min_{\beta}\left\|H\left(w_1,\ldots,w_{\tilde{N}},b_1,\ldots,n_{\hat{N}}\right)\beta-T\right\| \quad (4)$$

Input weight vectors w_i and hidden bias values b_i can be assigned randomly if N number of hidden nodes equals $\hat{N}$ number of samples, and matrix H is square and translatable. However, in most practical applications, the number of hidden neurons is significantly smaller than the number of training samples, resulting in a non-square H matrix. In such cases, an exact solution to the equation $H\beta = T$ may not exist. To address this issue, the least squares solution is employed to obtain an approximate solution that minimizes the error between the predicted and target outputs, as expressed in Equation (5):

$$\beta = H^{*}T \quad (5)$$

Specifically, the Moore–Penrose generalized inverse of the matrix $\boldsymbol{H}$, denoted as $\boldsymbol{H}^{*}$, is utilized to compute the output weights analytically [39]. This approach enables efficient and stable training of the network without the need for iterative optimization procedures.

The basic structure of the ELM, consisting of an input layer, a single hidden layer, and an output layer, is illustrated in Figure 3. Given a training dataset $\aleph = \{(x_i, t_i) \parallel x_i \in R^n, t_i \in R^m, i = 1,\ldots,N\}$, an activation function $g(x)$, and a predefined number of hidden nodes $\tilde{N}$; the ELM algorithm can be implemented through the following main steps:

- Step 1: Randomly assign the input weights w_i and hidden layer bias values b_i for $i = 1,\ldots,\tilde{N}$;
- Step 2: Compute the hidden layer output matrix $\boldsymbol{H}$ using the activation function based on the assigned weights and biases;
- Step 3: Determine $\beta = \boldsymbol{H}^{*}\boldsymbol{T}$, in which $\boldsymbol{T} = [\boldsymbol{t_1},\ldots,\boldsymbol{t_N}]^{\boldsymbol{T}}$, and $\boldsymbol{H}^{*}$ denotes the Moore–Penrose generalized inverse of matrix $\boldsymbol{H}$.

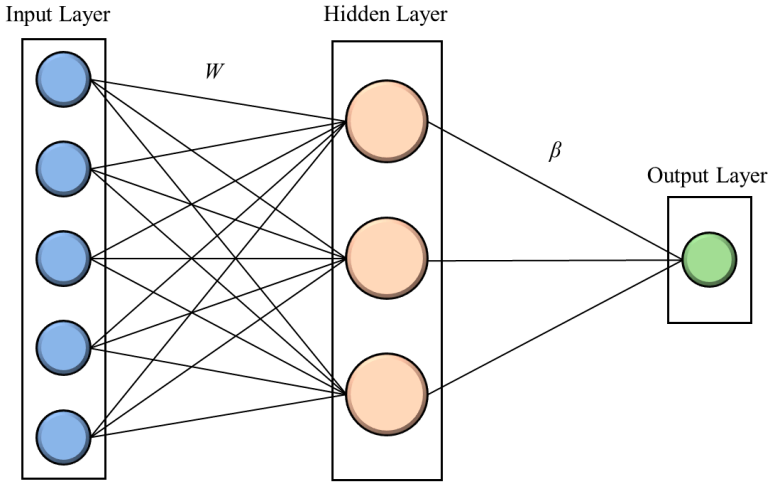

Figure 3. Basic structure of the ELM, including input, hidden, and output layers.

4.2. Data Preprocessing

Machine learning (ML) methods aim to construct generalizable prediction models; however, their performance is highly dependent on the quality and distribution of the input data. Therefore, data preprocessing plays a crucial role in enhancing the effectiveness of learning-based approaches. Among preprocessing techniques, normalization is commonly employed to improve data quality by scaling feature values into a comparable range, ensuring that each attribute contributes proportionally to the learning process [40].

In the dataset used in this study, most attributes do not have a predefined upper bound, and there exists a significant disparity between their minimum and maximum values, as presented in Table 4. Such variations may negatively affect the learning process by causing certain features to dominate others. To mitigate this issue, all input features were scaled using *min–max* normalization according to Equation (6) to map values into a fixed range. This approach was preferred to ensure that features with different scales contribute uniformly to the learning process and to maintain numerical stability, particularly in the ELM model.

$$scale\ data\ using\ min-max;\ x'_{i,n} = \frac{x_{i,n} - min(x_i)}{max(x_i) - min(x_i)} \tag{6}$$

Although some variables, such as star count, may exhibit skewed distributions, no logarithmic transformation was applied in order to preserve the original data distribution and evaluate model performance under realistic conditions. This decision prevents potential transformation-induced bias in the learning process and allows a more direct assessment of the model's robustness to raw data distributions.

4.3. Evaluation Metrics

To evaluate the performance of the developed models and to enable a comprehensive comparison among different configurations, three widely used evaluation metrics are employed in this study. These metrics are Mean Absolute Error (*MAE*), Mean Squared Error (*MSE*), and the coefficient of determination (R^2), each capturing different aspects of prediction performance [41]. *MAE* provides a straightforward measure of the average magnitude of prediction errors without considering their direction, offering an interpretable indication of model accuracy. *MSE*, on the other hand, emphasizes larger errors due to the squaring operation, making it particularly useful for identifying models with significant deviations. In addition, the R^2 metric evaluates the proportion of variance in the dependent variable that is explained by the model, thereby reflecting the overall goodness-of-fit. The mathematical formulations of these metrics are given as follows:

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|t_j - o_j| \tag{7}$$

$$MSE = \left(\frac{1}{n}\right) \times \sum_{j}|t_j - o_j|^2 \tag{8}$$

$$R^2 = 1 - \left(\frac{\sum_j (t_j - o_j)^2}{\sum_j (t_j - \hat{t})^2}\right) \tag{9}$$

By jointly considering these complementary metrics, a more robust and reliable assessment of model performance is achieved, allowing both error-based evaluation and explanatory capability to be analyzed.

5. Results and Discussion

In this study, the star count of public GitHub repositories is predicted by utilizing a total of 13 input features, consisting of 2 GitHub-based metrics and 11 static software metrics extracted from the source code. The inclusion of both metric types enables a more comprehensive representation of software projects for popularity prediction. The ELM method is applied to the constructed datasets, and the prediction performance is evaluated through a predefined evaluation methodology. Moreover, LR, SVM, LSBoost, and RF are also evaluated and compared against the ELM model. The obtained results are systematically presented and analyzed to assess the effectiveness of the proposed approach. In particular, the contribution of static software metrics to the prediction of repository popularity, represented by the star count, is investigated in detail. Through this analysis, the study aims to reveal whether incorporating static code-related features can enhance the predictive capability beyond traditional GitHub metrics. Overall, the results provide evidence that integrating static software metrics can improve predictive performance compared to relying solely on GitHub-based metrics.

5.1. Evaluation Scheme

The evaluation scheme employed in this study is illustrated in Figure 4. Initially, the proposed data acquisition tool is utilized to generate multiple datasets based on different programming languages and repository topics. Specifically, Java and C# are selected as the programming languages, and seven commonly used software development domains are defined as input topics: continuous integration and continuous deployment (CI/CD), database, desktop, learning-based, mobile, service, and web. Accordingly, seven distinct datasets are automatically constructed, each containing unique repositories associated with the corresponding topic. In addition, a combined dataset is created by aggregating all topic-specific datasets. Following dataset construction, data preprocessing is performed through normalization, and ML models are applied to predict the star count of the repositories. All models were trained and tested under identical experimental conditions to ensure a fair comparison. The experimental results are presented in this section, enabling a comprehensive evaluation of the relative performance of ELM against widely used alternative methods. The prediction performance is evaluated using standard evaluation metrics.

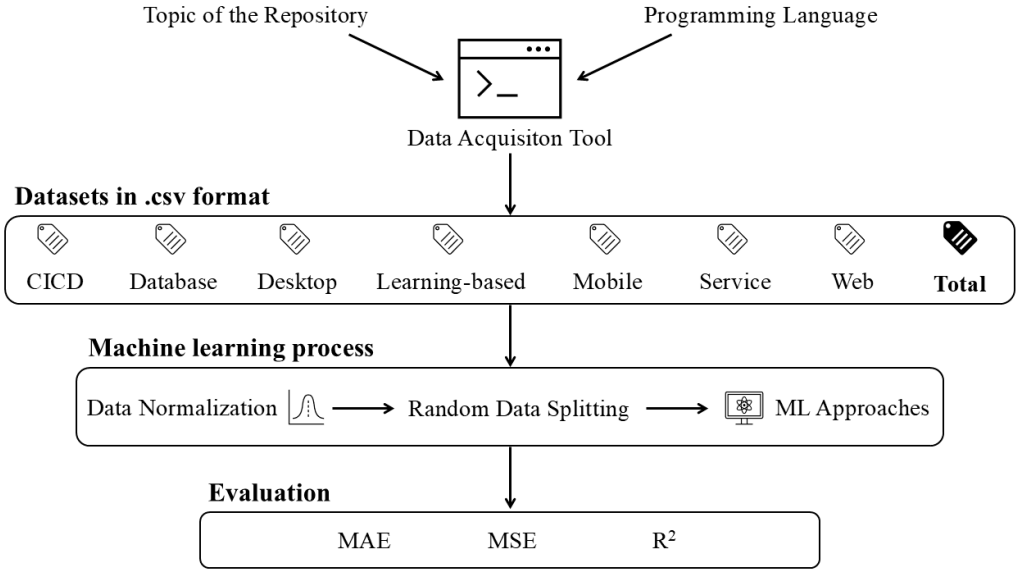

Figure 4. Evaluation methodology applied in the study.

The characteristics of the datasets used for ELM training are summarized in Table 5, including the total number of samples as well as the distribution of training and test data for each topic. The SVM model was implemented with an RBF kernel, RF was trained with 100 regression trees, and LSBoost was configured with 100 learning cycles, while the parameter settings of the other baseline methods were defined accordingly. For the combined dataset, duplicate entries (arising from repositories labeled under multiple topics) are carefully identified and removed, resulting in the elimination of 1063 redundant samples and ensuring data consistency. The final datasets are evaluated using 10-fold cross-validation in order to provide a more robust and reliable assessment of model performance. In this approach, each dataset is randomly partitioned into ten equal folds, where in each iteration nine folds are used for training and the remaining fold is used for testing. This process is repeated ten times, ensuring that each instance is used both for training and testing exactly once. Predictions from all test folds were aggregated, and the final performance metrics were computed over the combined out-of-fold predictions. During the evaluations, the sigmoid function is adopted as the activation function due to its widespread use and proven effectiveness in nonlinear modeling tasks. The number of hidden neurons in the ELM model was determined through a sensitivity analysis. A range of hidden neuron values between 10 and 300 was evaluated in preliminary experiments to assess their impact on model performance and stability. Based on these experiments, 40 hidden neurons were selected as the final configuration, as it provided a favorable trade-off between predictive accuracy and stability. In all experiments, the sigmoid activation function was used, which is commonly adopted in ELM-based models due to its effectiveness and compatibility with the random weight assignment mechanism of the hidden layer. This configuration allows the ELM model to capture underlying patterns in the data while maintaining fast training capability.

Table 5. Properties of the generated datasets and ELM-specific parameters used in the evaluation.

Dataset	Data Samples	Activation Function	Number of Neurons
CICD Apps	220	Sigmoid	40
Database Apps	374		
Desktop Apps	466		
Learning-based Apps	160		
Mobile Apps	1159		
Service Apps	678		
Web Apps	1383		
Total	3377		

It should be noted that the topic-specific datasets used in this study vary considerably in size, ranging from 160 to 1383 samples. This variation may influence the stability and reliability of the performance metrics, particularly in smaller datasets. Although lower performance is observed in some smaller datasets, the results indicate that performance differences are not solely driven by sample size, but are also affected by domain-specific characteristics of the repositories. However, no formal statistical significance tests were conducted in this study; therefore, the observed differences should be interpreted with caution.

5.2. Results

The evaluation results are summarized in Table 6 for each individual dataset as well as for the combined dataset. When Table 6 is examined, the highest performance on the CICD Apps dataset is achieved by the Linear Regression model (R^2 = 0.8171). The ELM model exhibits a moderate performance with an R^2 value of 0.6639, outperforming SVM, LSBoost, and RF; however, it does not yield the best result. This suggests that linear

relationships may be more dominant in the CICD dataset, allowing simpler models to perform more effectively.

Table 6. Performance evaluation of ELM and other ML models for star count prediction based on the evaluation metrics.

CICD Apps				Database Apps			
Method	**RMSE**	**MAE**	**R^2**	**Method**	**RMSE**	**MAE**	**R^2**
LR	**1104.9262**	**431.7647**	**0.8171**	LR	886.3236	**369.9072**	0.8175
SVM	2542.2554	703.3277	0.0319	SVM	2031.2968	594.2901	0.0416
LSBoost	2422.6669	901.6331	0.1209	LSBoost	1593.3826	530.8455	0.4103
RF	2144.1646	589.5706	0.3114	RF	1494.5386	392.6650	0.4812
ELM	1497.9450	792.3538	0.6639	ELM	**849.1388**	428.4873	**0.8325**
Desktop Apps				**Learning-Based Apps**			
Method	**RMSE**	**MAE**	**R^2**	**Method**	**RMSE**	**MAE**	**R^2**
LR	1092.9208	518.4462	0.4808	LR	1564.6021	676.6005	0.4160
SVM	1123.7339	470.1709	0.4512	SVM	1927.8392	617.4400	0.1134
LSBoost	1180.9284	607.4464	0.3939	LSBoost	1507.1669	589.6497	0.4581
RF	**911.1043**	**443.5975**	**0.6392**	RF	1630.8948	**507.0670**	0.3655
ELM	984.6494	542.1970	0.5786	ELM	**1374.4634**	725.5094	**0.5493**
Mobile Apps				**Service Apps**			
Method	**RMSE**	**MAE**	**R^2**	**Method**	**RMSE**	**MAE**	**R^2**
LR	1868.4418	421.5051	0	LR	883.1411	304.0469	0.7353
SVM	1453.7169	452.0446	0.1404	SVM	1588.9056	480.0818	0.1433
LSBoost	1037.3303	355.4495	0.5623	LSBoost	1359.2577	458.1832	0.3730
RF	**929.4019**	**277.3370**	**0.6487**	RF	1154.5136	**321.6246**	0.5477
ELM	1209.5501	381.7863	0.4049	ELM	**672.8992**	325.9593	**0.8463**
Web Apps				**Total**			
Method	**RMSE**	**MAE**	**R^2**	**Method**	**RMSE**	**MAE**	**R^2**
LR	2317.6540	660.2378	0	LR	1712.5227	537.3132	0.1692
SVM	2174.7013	681.4602	0.1007	SVM	1731.1929	546.7153	0.1510
LSBoost	1794.0643	635.2489	0.3880	LSBoost	1435.2235	509.7410	0.4165
RF	1527.0361	**450.8754**	0.5566	RF	**1165.5877**	**340.7816**	**0.6152**
ELM	**1513.4826**	637.2216	**0.5644**	ELM	1245.7634	469.0849	0.5604

Bold values represent the best results for the corresponding evaluations.

For the Database Apps dataset, the best performance is obtained by ELM (R^2 = 0.8325). ELM provides higher explanatory power compared to Linear Regression (R^2 = 0.8175) and demonstrates clearly superior performance over the other methods. This indicates that nonlinear relationships in the dataset are better captured by ELM.

In the Desktop Apps dataset, the most successful model is Random Forest (R^2 = 0.6392). The ELM model achieves an R^2 value of 0.5786, showing competitive performance but not reaching the best result. This suggests that complex structures within the dataset are better captured by ensemble-based methods.

For the Learning-based Apps dataset, the best performance is achieved by ELM (R^2 = 0.5493). LSBoost ranks second with an R^2 value of 0.4581, while the remaining methods show lower performance. The superiority of ELM in this case indicates its effectiveness in modeling complex and nonlinear relationships within the dataset.

In the Mobile Apps dataset, the highest performance is achieved by Random Forest (R^2 = 0.6487). The ELM model shows a lower performance with an R^2 value of 0.4049. This

result indicates that ELM is relatively limited in this dataset, while ensemble methods are more suitable.

For the Service Apps dataset, the best performance is obtained by ELM (R^2 = 0.8463). This value is significantly higher than all other methods. Linear Regression ranks second with an R^2 value of 0.7353, while the remaining methods show lower performance. These results demonstrate that ELM is a highly effective model for this dataset.

In the Web Apps dataset, the best performance is achieved by ELM (R^2 = 0.5644), closely followed by Random Forest (R^2 = 0.5566). The marginal difference between the two methods indicates that both approaches are highly competitive in this dataset.

Finally, for the Total dataset, the best performance is obtained by Random Forest (R^2 = 0.6152). The ELM model ranks second with an R^2 value of 0.5604, showing competitive performance. This suggests that ensemble methods may provide an advantage in heterogeneous data structures.

When evaluated in terms of R^2 values, the ELM model achieves the best performance in four out of eight datasets (Database, Learning-based, Service, and Web Apps). The Random Forest model yields the best results in 3 datasets (Desktop, Mobile, and Total), while the Linear Regression model performs best in 1 dataset (CICD Apps). This distribution clearly demonstrates that ELM produces strong and competitive results across multiple datasets; however, its performance may vary depending on the characteristics of the dataset. Overall, ELM demonstrates strong predictive capability, achieving top performance in multiple datasets, although its effectiveness is dataset-dependent. Nevertheless, the results also indicate that model performance is highly dependent on dataset-specific characteristics and may vary accordingly.

The comparison between the predicted star counts obtained from the ML models and the corresponding actual values is illustrated in Figures 5–12 for each dataset. As shown in the figure, the prediction error tends to increase for samples that can be considered outliers within their respective datasets. Despite the inherent difficulty in modeling such extreme values, the ELM approach demonstrates a strong capability to approximate the target values across most samples. In particular, for several datasets, the model produces predictions that are relatively close to the actual values even for certain outliers. Moreover, the presence of outliers becomes more pronounced in the prediction plot of the combined dataset, where a wider variation in star counts is observed due to the aggregation of multiple application domains. Nevertheless, even in this more challenging setting, the ELM model maintains a consistent prediction behavior, producing values that are generally close to the ground truth for both typical samples and a subset of outliers. Overall, these results indicate that the model preserves its generalization capability when exposed to more diverse and complex data distributions, thereby supporting its applicability to real-world software repository popularity prediction tasks.

The distribution of GitHub star counts in the dataset is highly skewed, ranging approximately from 10 to 30,000. This indicates the presence of extreme values, where a small number of highly popular repositories may disproportionately influence error-based evaluation metrics. To mitigate this effect, a threshold-based treatment was applied to reduce the impact of extreme values during evaluation. This approach was adopted to ensure a more balanced assessment of model performance across repositories with varying popularity levels. Nevertheless, more advanced approaches such as robust regression techniques and logarithmic transformations were not applied in this study, which represents a limitation of the current work.

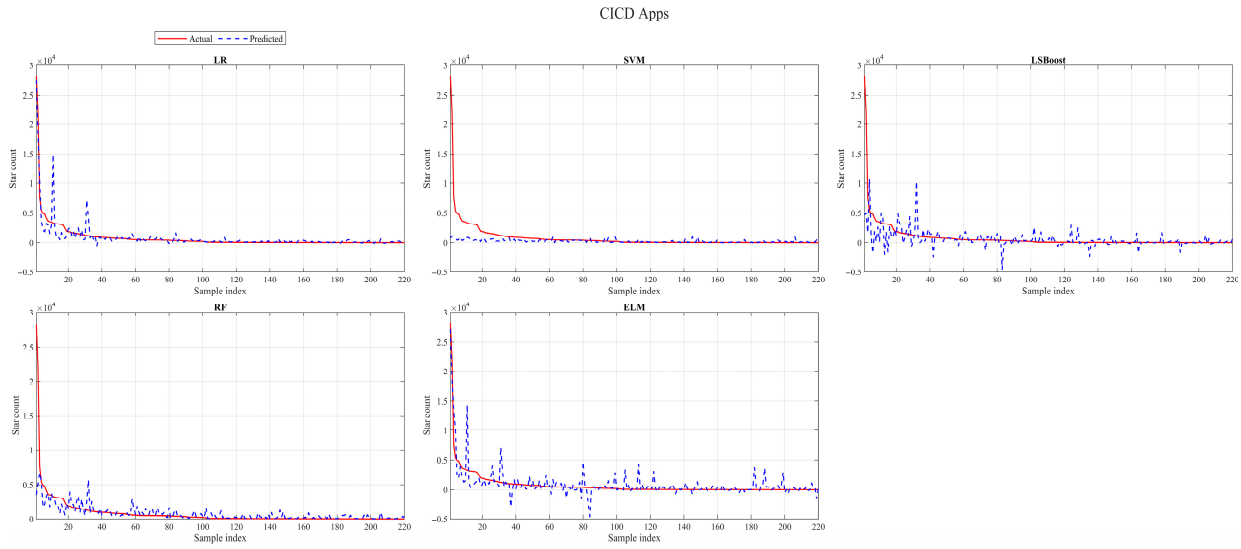

Figure 5. Comparison of ML models for actual and predicted star counts in CICD apps.

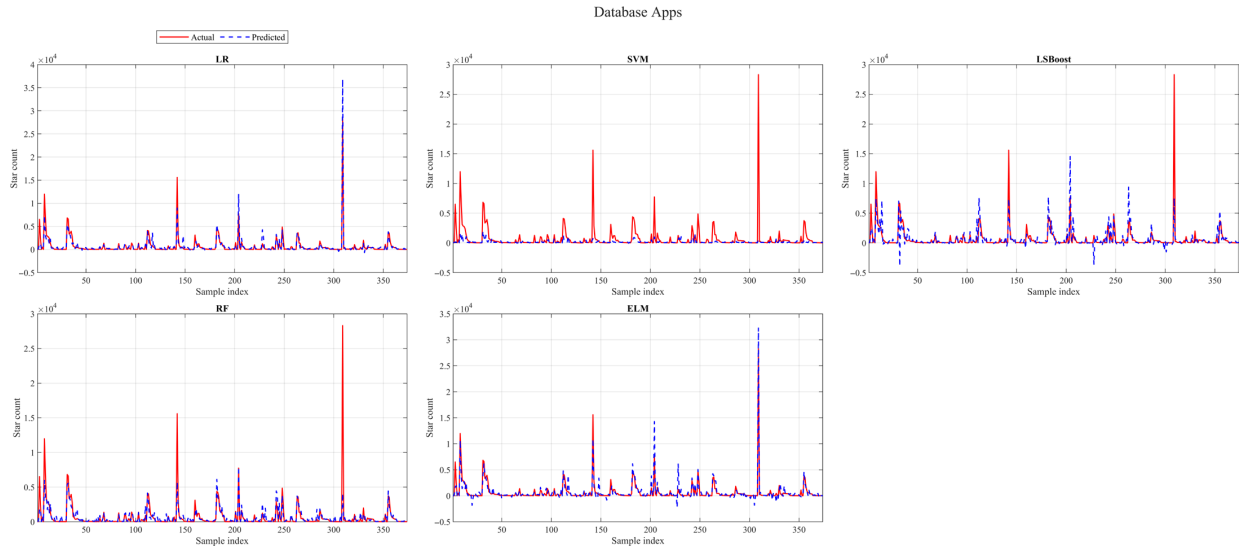

Figure 6. Comparison of ML models for actual and predicted star counts in Database apps.

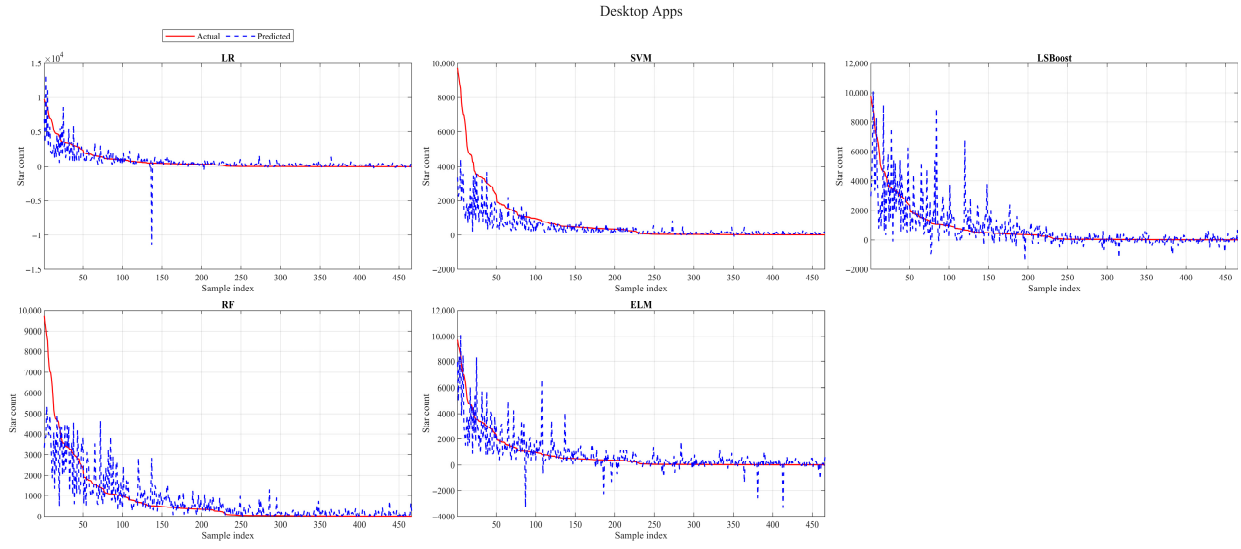

Figure 7. Comparison of ML models for actual and predicted star counts in Desktop apps.

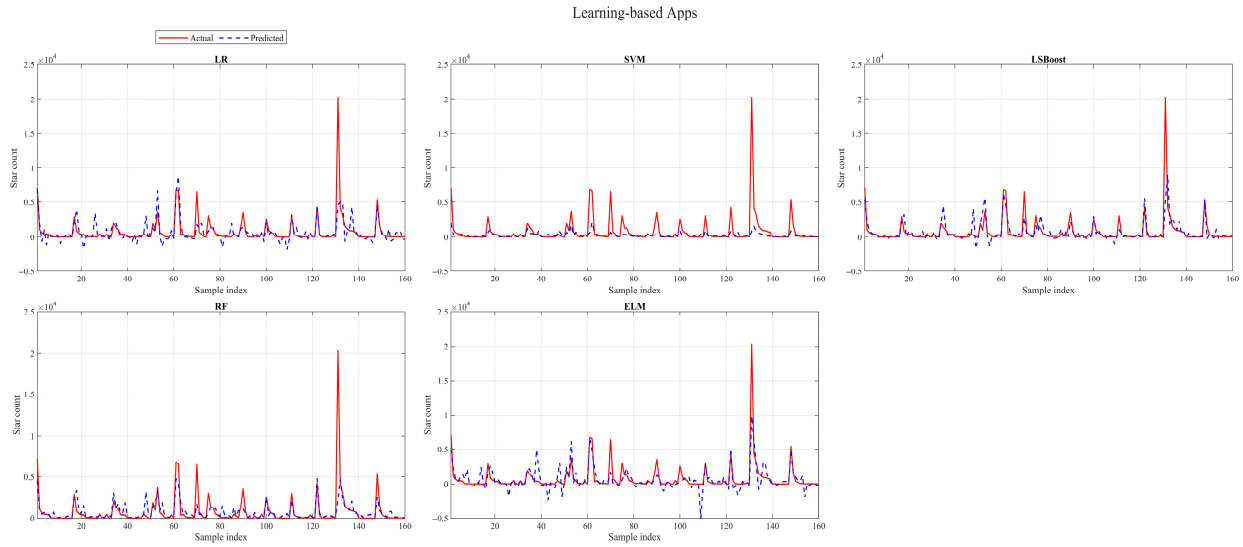

Figure 8. Comparison of ML models for actual and predicted star counts in Learning-based apps.

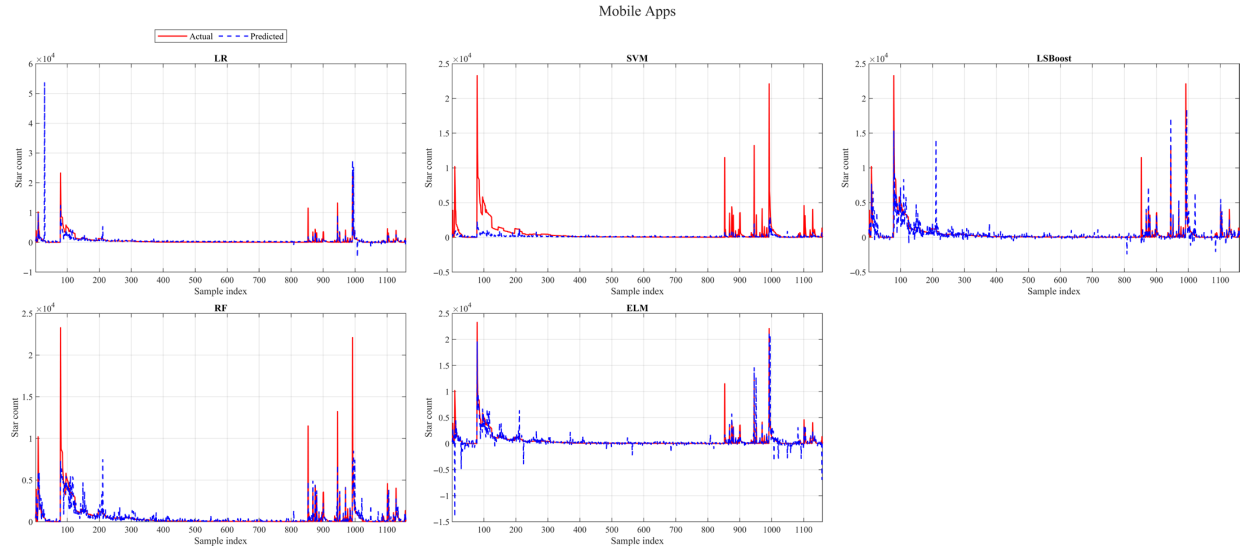

Figure 9. Comparison of ML models for actual and predicted star counts in Mobile apps.

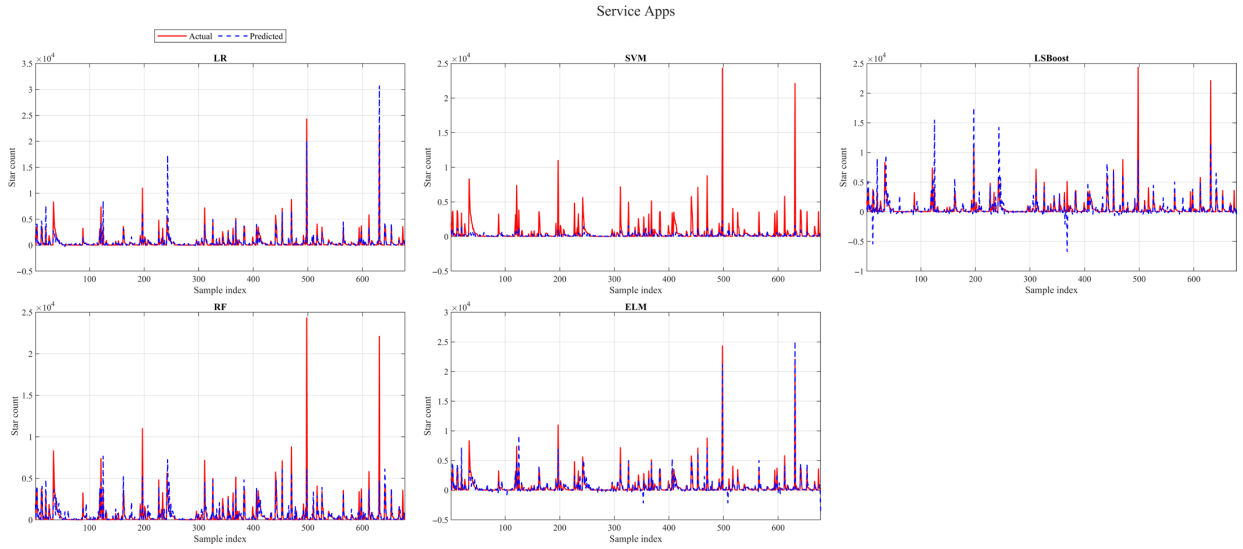

Figure 10. Comparison of ML models for actual and predicted star counts in Service apps.

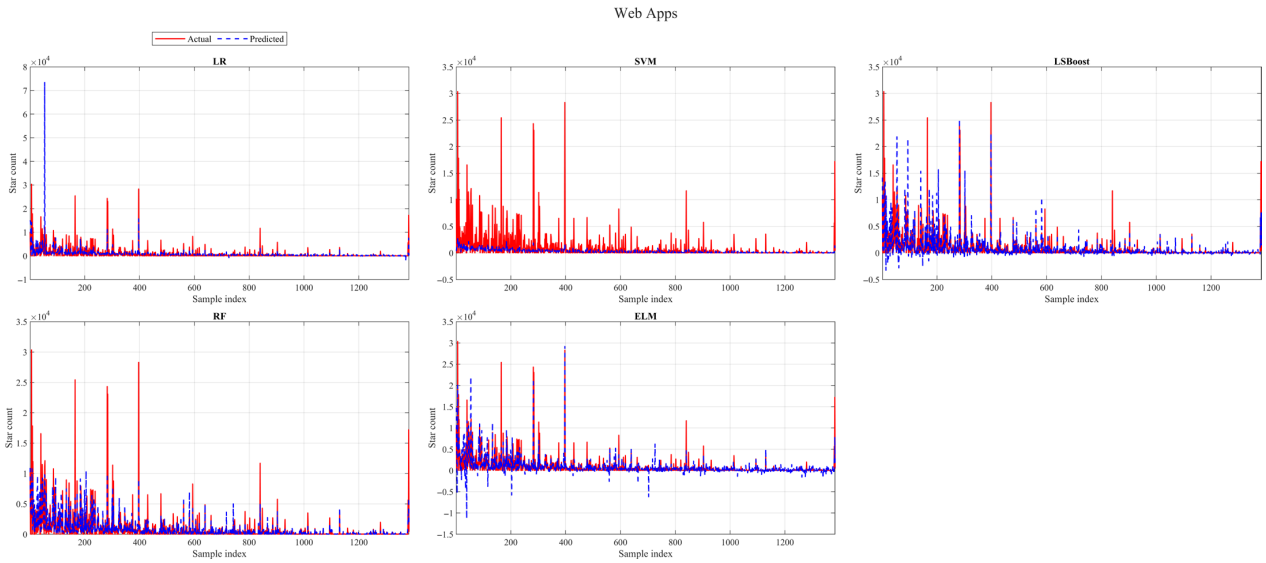

Figure 11. Comparison of ML models for actual and predicted star counts in Web apps.

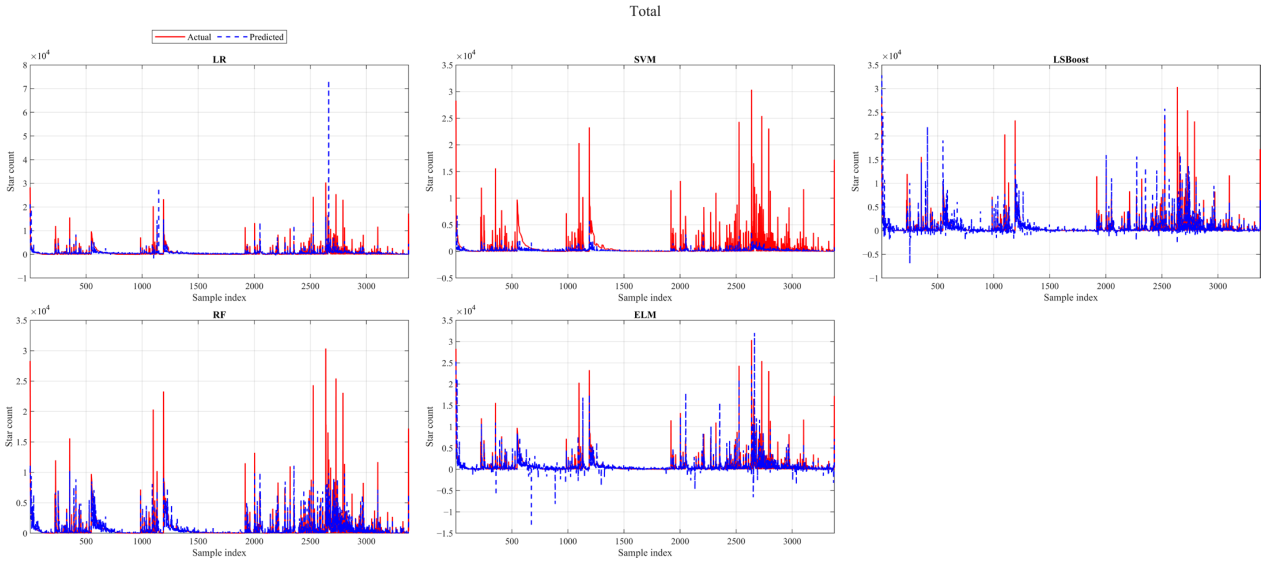

Figure 12. Comparison of ML models for actual and predicted star counts in the entire dataset.

In addition, the scatter plots presented in Figures 13–20 provide further insight into the relationship between predicted and actual star counts. The distribution of data points demonstrates a strong positive correlation, with the majority of samples aligning closely along a diagonal trend extending from the lower-left to the upper-right corner. This alignment indicates that the ELM model is capable of effectively capturing the underlying mapping between the input features and the target variable. The concentration of points around the ideal regression line suggests that the model produces consistent and reliable predictions for a substantial portion of the data. A more detailed examination of the scatter distribution reveals that deviations from the ideal line are primarily associated with high-value star count samples, which correspond to outliers in the dataset. These deviations are expected due to the highly skewed nature of the target variable and the inherent difficulty of accurately modeling extreme values in regression problems. Nevertheless, the overall dispersion remains limited, indicating that prediction errors are generally controlled and do not significantly distort the overall performance trend. Furthermore, the observed correlation pattern highlights the strong generalization capability of the ELM approach, as the model maintains a stable

predictive relationship across datasets with varying characteristics and distributions. The consistency of this pattern across both individual and combined datasets suggests that the proposed feature set, which integrates GitHub metrics with static software metrics, provides a meaningful representation for popularity prediction. This reinforces the claim that incorporating static software metrics contributes positively to the learning process.

Overall, the scatter plot analysis confirms that the ELM-based model achieves a high level of predictive accuracy while preserving robustness against variations in data distribution. Although certain limitations are observed in handling extreme outliers, the model consistently demonstrates reliable performance, thereby validating its applicability to the popularity prediction of software repositories across diverse contexts.

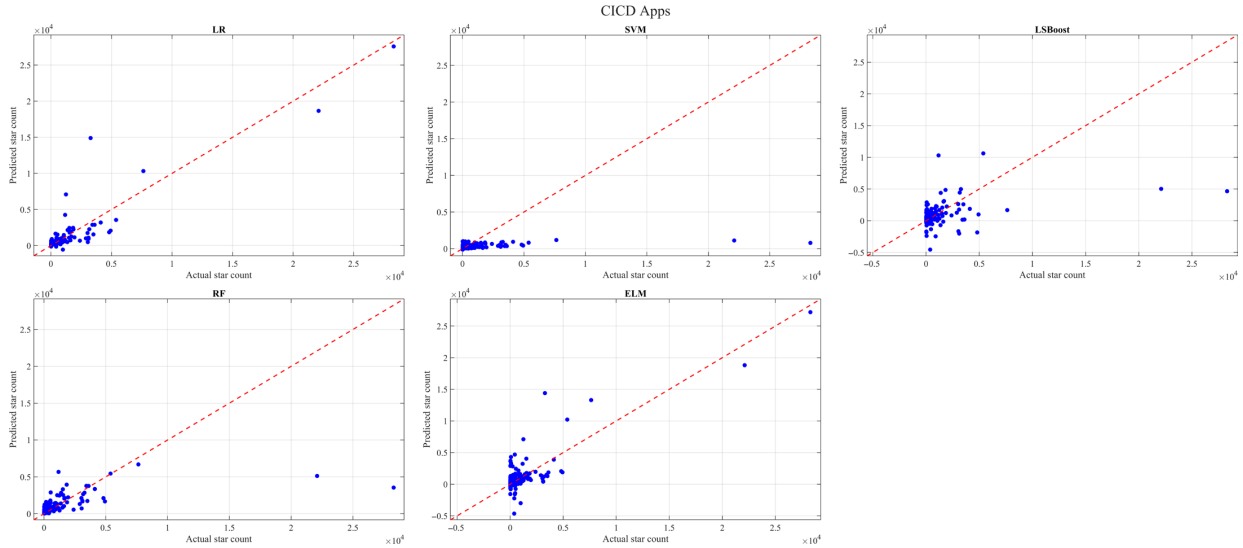

Figure 13. Scatter plots of ML predictions versus actual star counts for the CICD apps dataset.

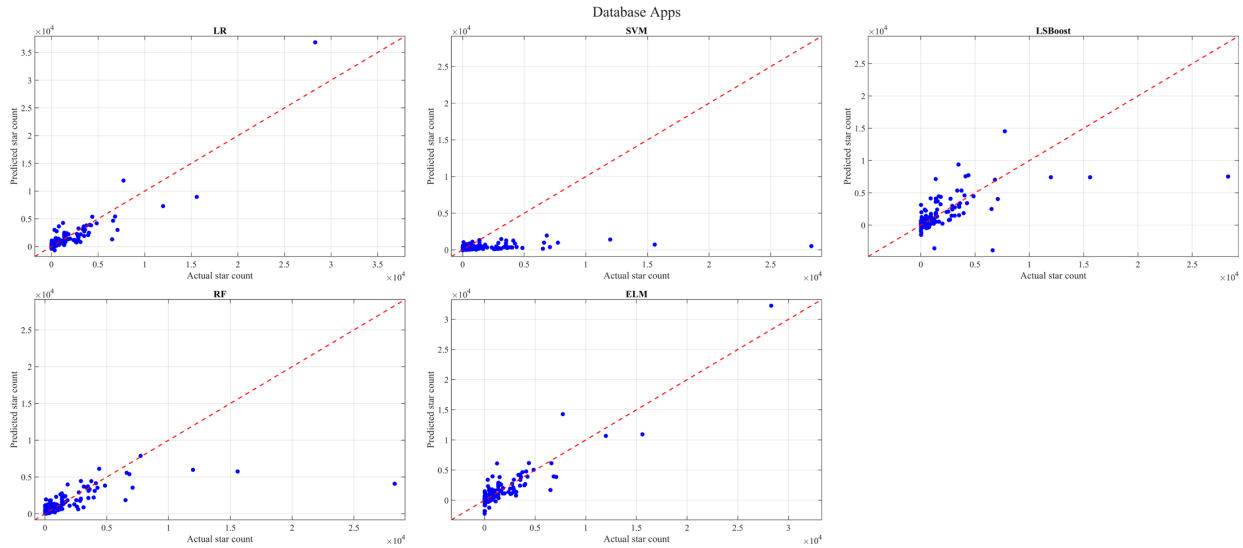

Figure 14. Scatter plots of ML predictions versus actual star counts for the Database apps dataset.

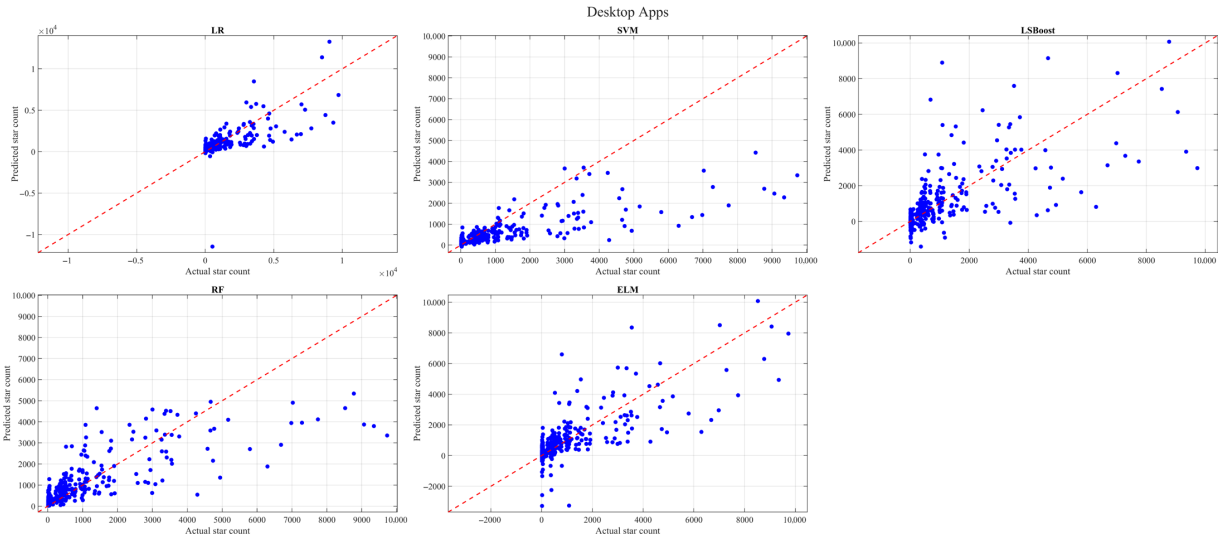

Figure 15. Scatter plots of ML predictions versus actual star counts for the Desktop apps dataset.

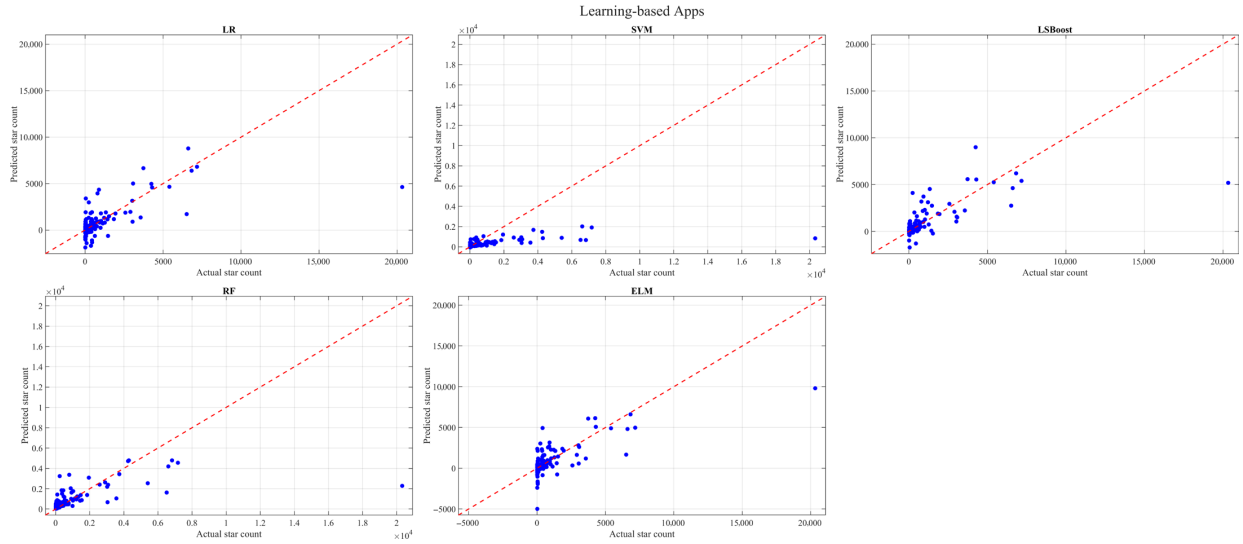

Figure 16. Scatter plots of ML predictions versus actual star counts for the Learning-based apps dataset.

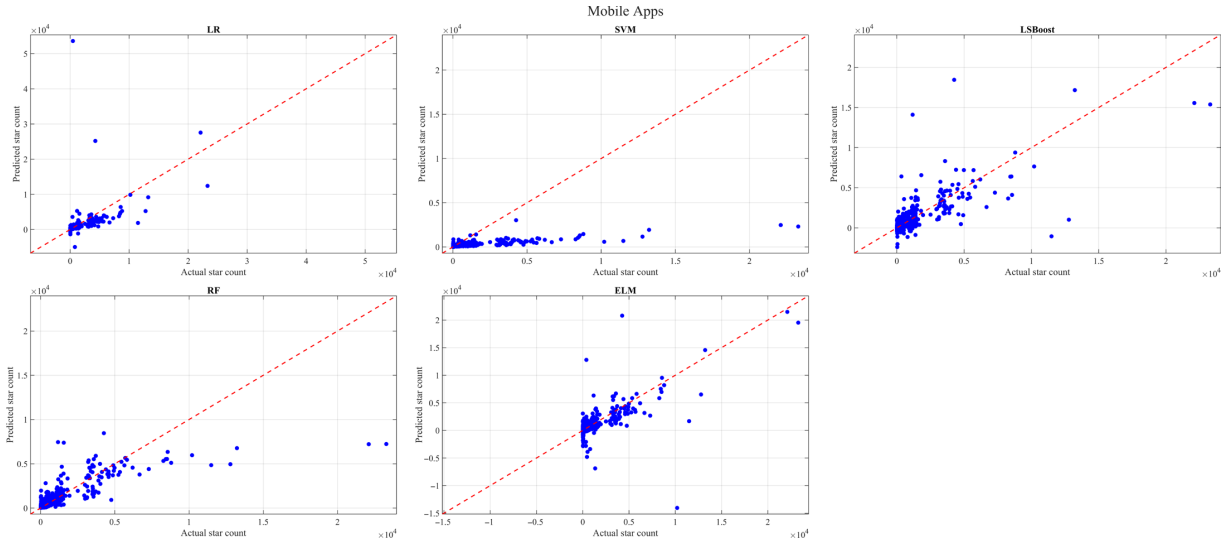

Figure 17. Scatter plots of ML predictions versus actual star counts for the Mobile apps dataset.

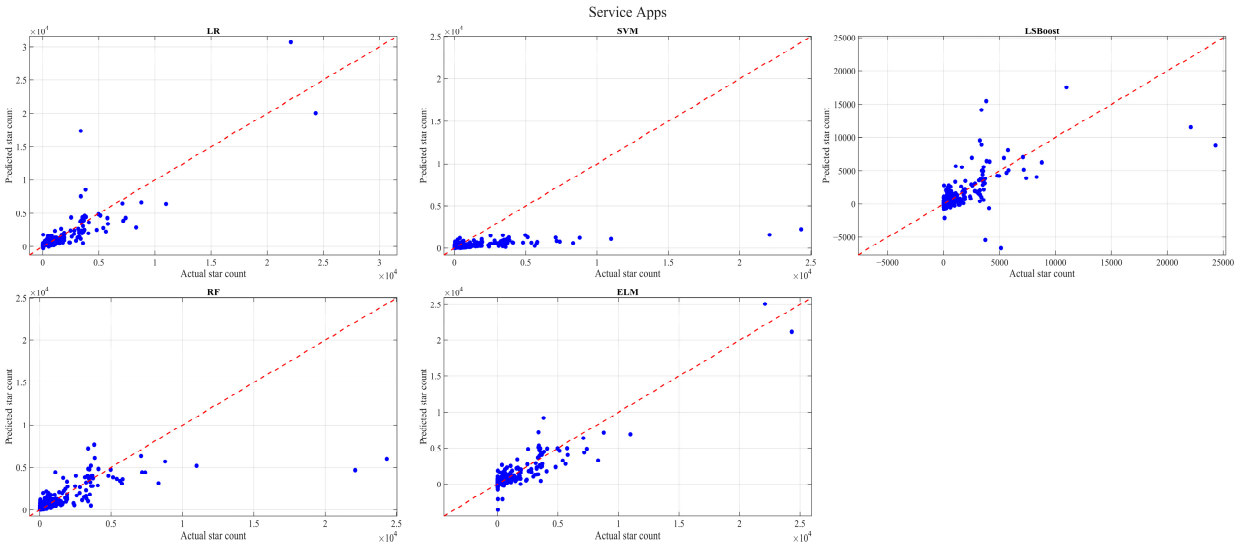

Figure 18. Scatter plots of ML predictions versus actual star counts for the Service apps dataset.

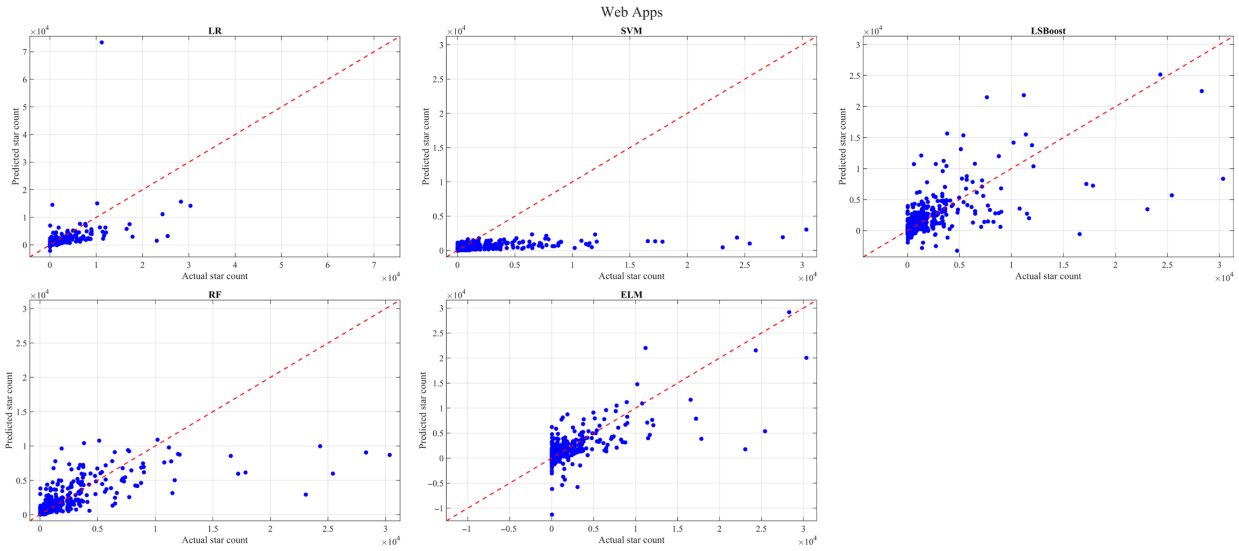

Figure 19. Scatter plots of ML predictions versus actual star counts for the Web apps dataset.

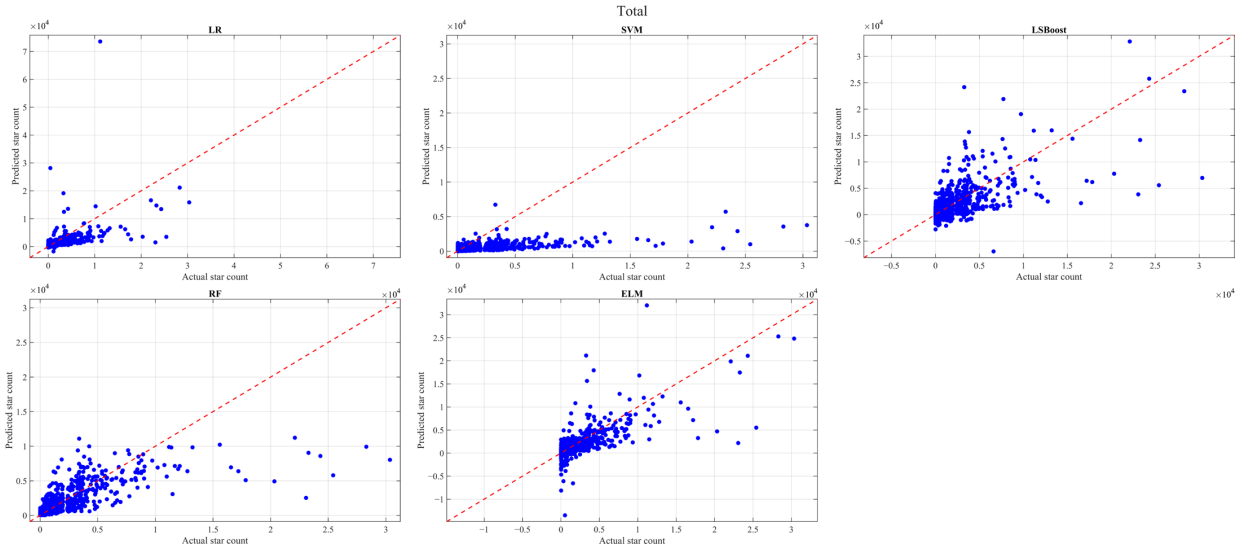

Figure 20. Scatter plots of ML predictions versus actual star counts for the entire dataset.

5.3. Discussion

In this study, a comprehensive approach is proposed by incorporating not only commonly used GitHub metrics but also static software metrics for star-count-based popularity prediction of public GitHub repositories. While existing studies predominantly rely on repository-level indicators such as fork count and repository size, this work extends the feature space by integrating 11 different static software metrics obtained via the SourceMonitor CLI tool. This integration enables a more holistic representation of software projects by capturing both repository activity and intrinsic code characteristics. Accordingly, an automatically generated and topic-oriented dataset is constructed that covers several popular software development domains on GitHub, and different ML models, including ELM, LR, SVM, LSBoost, and RF, are evaluated on the dataset.

To evaluate the contribution of different feature groups, an ablation study was conducted using three configurations: (i) GitHub-only metrics, (ii) static software metrics only, and (iii) the combined feature set. The results indicate that the GitHub-only configuration yields very poor predictive performance, with R^2 values close to zero across all datasets. The static-only configuration shows slight improvement, with R^2 values ranging approximately between 0.01 and 0.11. In contrast, the combined feature set significantly improves performance, achieving R^2 values of 0.8463 (Service_Apps), 0.8325 (Database_Apps), and 0.6639 (CICD_Apps), while other datasets show values between 0.40 and 0.56. These findings suggest that no single feature group dominates the prediction process and that the integration of heterogeneous metrics leads to a more informative representation of repository popularity. This confirms the complementary nature of GitHub-based and static software metrics in predicting repository popularity.

As a consequence, this study can be considered a pioneering effort in establishing a link between static software metrics and popularity analysis in GitHub repositories. Although the present work focuses on star count as an indicator of repository popularity, future studies may extend this approach to other popularity-related attributes such as forks, watchers, or contributor growth. In addition, static software metrics may also be leveraged in different predictive tasks, including software maintenance estimation, contributor behavior analysis, and long-term support prediction. The automated dataset construction tool developed in this study further provides a practical foundation for future research, enabling the generation of scalable, consistent, and reproducible datasets for learning-based software analytics.

5.4. Limitations and Threats to Validity

To discuss the limitations of the study, it should be noted that this study considers GitHub star count as the sole indicator of repository popularity. Although star count is widely used and provides an intuitive measure of user interest and visibility, it may be influenced by external factors such as social media exposure, recency effects, and project promotion. Therefore, it may not fully capture all dimensions of repository popularity. Other indicators, such as fork count, number of watchers, and contributor activity, may provide complementary insights into different aspects of popularity and project impact. Accordingly, relying solely on star count represents a limitation of this study, and the results should be interpreted in this context. In addition, another limitation of this study is that the dataset is restricted to repositories written in C# and Java due to the use of SourceMonitor CLI for static metric extraction. While this ensures consistency in metric computation, it may limit the generalizability of the findings across different programming languages and paradigms. Extending the proposed framework with alternative static analysis tools supporting languages such as Python, TypeScript, and Go is considered an important direction for future work.

To discuss the threats to validity, it should be noted that the dataset construction process may introduce certain biases. Specifically, the use of GitHub topic labels as a selection criterion may favor repositories that are more actively maintained, better documented, or more mature, while potentially excluding repositories without assigned topics. This may limit the diversity of the dataset and pose a threat to external validity by affecting the generalizability of the findings. Additionally, the reliance on SourceMonitor restricts the analysis to repositories compatible with the tool, which may further constrain the scope of the dataset. To mitigate these limitations, it should be noted that the dataset collection tool developed in this study has a modular and reusable design, allowing the construction of alternative datasets with different selection criteria. This flexibility enables future studies to reduce potential biases by incorporating more diverse repositories and varying filtering strategies, thereby improving the robustness and generalizability of the proposed approach.

Moreover, it should be noted that a potential bidirectional relationship may exist between static software metrics and repository popularity. Highly popular repositories may attract more contributors, which can lead to improvements in code quality and, consequently, more favorable static metric values. Therefore, static metrics may partially reflect the outcome of popularity rather than its cause. In this study, the focus is on predictive modeling rather than causal inference. Thus, the results should be interpreted as indicating the predictive usefulness of static software metrics, not as evidence of a causal relationship. Investigating causality would require temporal or longitudinal analysis.

6. Conclusions

Public repository analysis has recently gained significant attention as a research area for learning-based approaches, particularly in tasks such as influence analysis and popularity prediction. GitHub serves as a rich and widely utilized data source for such studies, where the star count is commonly accepted as a direct indicator of repository popularity. However, existing approaches in the literature predominantly rely on GitHub-derived metrics, while overlooking the potential contribution of static software metrics that can be extracted from the source code of repositories. In this study, a novel approach is proposed by integrating both GitHub metrics and static software metrics for star-count-based popularity prediction. To support this objective, an automated data acquisition tool has been developed to construct datasets from GitHub repositories across different topics. The resulting datasets include both commonly used GitHub metrics and static code-related metrics obtained via the SourceMonitor CLI tool. Subsequently, the ELM method is employed to evaluate the predictive capability of the proposed feature set. The experimental results demonstrate that ELM achieves strong performance across different datasets, with evaluation metrics indicating promising prediction accuracy. In particular, based on R^2 values, the ELM model achieved the best performance in 4 out of 8 datasets (Database, Learning-based, Service, and Web Apps). The Random Forest model achieved the best results in 3 datasets (Desktop, Mobile, and Total), while Linear Regression performed best in 1 dataset (CICD Apps). These findings indicate that ELM produces strong and competitive results across multiple scenarios; however, its performance varies depending on dataset characteristics. In conclusion, this study provides a new perspective for repository analysis and demonstrates the potential of combining code-level and repository-level features within learning-based models. ELM demonstrates its effectiveness as a strong predictive model by achieving superior performance in several datasets. Nevertheless, the results also show that model performance is influenced by the structural properties of the datasets, and no single method consistently outperforms others across all cases.

For future work, it is planned to expand the dataset by including a larger number of repositories from additional topics, enabling a more comprehensive investigation of

outlier effects on prediction performance. Furthermore, extending the dataset to incorporate repositories developed in a wider range of programming languages is expected to improve the generalizability of the proposed approach. In addition, the application of alternative and more advanced learning-based models will be explored to further enhance prediction accuracy and to provide a comparative analysis of different modeling techniques. Finally, the impact of different data preprocessing techniques on model performance will be systematically investigated. In particular, transformations such as logarithmic scaling and standardization will be considered to better handle skewed feature distributions and reduce the influence of extreme values, thereby providing a more comprehensive evaluation of their effects on the predictive capability of the ELM model.

Author Contributions: Conceptualization, E.B., F.Y. and O.A.; methodology, O.A.; software, Y.Ö.; validation, E.B., O.A. and Y.Ö.; formal analysis, E.B. and F.Y.; investigation, E.B. and F.Y.; resources, E.B. and O.A.; data curation, F.Y. and Y.Ö.; writing—original draft preparation, O.A. and Y.Ö.; writing—review and editing, F.Y. and Y.Ö.; visualization, E.B. and O.A.; supervision, E.B.; project administration, F.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All results reported in this study are generated using MATLAB codes developed by the authors. The dataset and source code are publicly available in a GitHub repository: https://github.com/yusufozcevik/elm-github-popularity-prediction (accessed on 10 May 2026). The repository includes a detailed README file with instructions for data collection, preprocessing, and model training, as well as a LICENSE file specifying the terms of use. To ensure transparency and reproducibility, all necessary scripts and resources required to replicate the experiments are provided in the repository.

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

References

1. Baumgartner, N.; Iyenghar, P.; Schoemaker, T.; Pulvermüller, E. AI-driven Refactoring: A Pipeline for Identifying and Correcting Data Clumps in Git Repositories. *Electronics* **2024**, *13*, 1644. [CrossRef]
2. Pejić, N.; Radivojević, Z.; Cvetavnović, M. Analyzing the Impact of COVID-19 on GitHub Event Trends. *Sustainability* **2023**, *15*, 14622. [CrossRef]
3. Moreno Martínez, C.; Gallego Carracedo, J.; Sánchez Gallego, J. Characterizing Agile Software Development: Insights from a Data-driven Approach using Large-scale Public Repositories. *Software* **2025**, *4*, 13. [CrossRef]
4. Saini, M.; Verma, R.; Singh, A.; Chahal, K.K. Investigating Diversity and Impact of the Popularity Metrics for Ranking Software Packages. *J. Softw. Evol. Process* **2020**, *32*, e2265. [CrossRef]
5. Wang, T.; Wang, S.; Chen, T.H.P. Study the Correlation between the Readme File of GitHub Projects and their Popularity. *J. Syst. Softw.* **2023**, *205*, 111806. [CrossRef]
6. Khezemi, N.; Ejaz, S.; Moha, N.; Guéhéneuc, Y.G. A comparison of code quality metrics and best practices in non-IoT and IoT systems. *Internet Things* **2025**, *34*, 101803. [CrossRef]
7. Coelho, J.; Valente, M.T.; Milen, L.; Silva, L.L. Is this GitHub Project Maintained? Measuring the Level of Maintenance Activity of Open-source Projects. *Inf. Softw. Technol.* **2020**, *122*, 106274. [CrossRef]
8. Xie, Q.; Wang, J.; Kim, G.; Lee, S.; Song, M. A Sensitivity Analysis of Factors Influential to the Popularity of Shared Data in Data Repositories. *J. Informetr.* **2021**, *15*, 101142. [CrossRef]
9. Moid, M.A.; Siraj, A.; Ali, M.F.; Amoodi, A.O. Predicting Stars on Open-Source GitHub Projects. In Proceedings of the 2021 Smart Technologies, Communication and Robotics (STCR), Sathyamangalam, India, 9–10 October 2021; pp. 1–9. [CrossRef]
10. AlMarzouq, M.; AlZaidan, A.; AlDallal, J. Mining GitHub for Research and Education: Challenges and Opportunities. *Int. J. Web Inf. Syst.* **2020**, *16*, 451–473. [CrossRef]
11. Abedini, Y.; Heydarnoori, A. Can GitHub Issues Help in App Review Classifications? *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 209. [CrossRef]
12. Ghodke, G.M.; Chavan, T. An Overview of Git. *Int. J. Sci. Res. Mod. Sci. Technol.* **2024**, *3*, 17–23. [CrossRef]

13. Kalliamvakou, E.; Gousios, G.; Blincoe, K.; Singer, L.; German, D.M.; Damian, D. An in-depth Study of the Promises and Perils of Mining GitHub. *Empir. Softw. Eng.* **2016**, *21*, 2035–2071. [CrossRef]
14. Dabic, O.; Aghajani, E.; Bavota, G. Sampling Projects in GitHub for MSR Studies. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 560–564. [CrossRef]
15. Wang, J.; Zhang, X.; Chen, L.; Xie, X. Personalizing Label Prediction for GitHub Issues. *Inf. Softw. Technol.* **2022**, *145*, 106845. [CrossRef]
16. Moriconi, F.; Durieux, T.; Falleri, J.R.; Troncy, R.; Francillon, A. GHALogs: Large-scale Dataset of GitHub Actions Runs. In Proceedings of the 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), Ottawa, ON, Canada, 28–29 April 2025; pp. 669–673. [CrossRef]
17. Alqaradaghi, M.; Kozsik, T. Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code. *IEEE Access* **2024**, *12*, 55824–55842. [CrossRef]
18. Ehrlinger, L.; Wöß, W. A Survey of Data Quality Measurement and Monitoring Tools. *Front. Big Data* **2022**, *5*, 850611. [CrossRef]
19. Schnoor, H.; Hasselbring, W. Comparing Static and Dynamic Weighted Software Coupling Metrics. *Computers* **2020**, *9*, 24. [CrossRef]
20. Pachouly, J.; Ahirrao, S.; Kotecha, K.; Selvachandran, G.; Abraham, A. A Systematic Literature Review on Software Defect Prediction using Artificial Intelligence: Datasets, Data Validation Methods, Approaches, and Tools. *Eng. Appl. Artif. Intell.* **2022**, *111*, 104773. [CrossRef]
21. Goyal, S. Static Code Metrics-based Deep Learning Architecture for Software Fault Prediction. *Soft Comput.* **2022**, *26*, 13765–13797. [CrossRef]
22. Borandag, E. Software Fault Prediction using an RNN-based Deep Learning Approach and Ensemble Machine Learning Techniques. *Appl. Sci.* **2023**, *13*, 1639. [CrossRef]
23. Nevendra, M.; Singh, P. A Survey of Software Defect Prediction based on Deep Learning. *Arch. Comput. Methods Eng.* **2022**, *29*, 5723–5748. [CrossRef]
24. Jorayeva, M.; Akbulut, A.; Catal, C.; Mishra, A. Machine Learning-based Software Defect Prediction for Mobile Applications: A Systematic Literature Review. *Sensors* **2022**, *22*, 2551. [CrossRef]
25. Khleel, N.A.A.; Nehéz, K. Software Defect Prediction using a Bidirectional LSTM Network Combined with Oversampling Techniques. *Clust. Comput.* **2024**, *27*, 3615–3638. [CrossRef]
26. Jo, S.; Kwon, R.; Kwon, G. Probabilistic Model Checking GitHub Repositories for Software Project Analysis. *Appl. Sci.* **2024**, *14*, 1260. [CrossRef]
27. Chowdhury, S.; Uddin, G.; Hemmati, H.; Holmes, R. Method-level Bug Prediction: Problems and Promises. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 98. [CrossRef]
28. Jász, J. The Effectiveness of Hidden Dependence Metrics in Bug Prediction. *IEEE Access* **2024**, *12*, 77214–77225. [CrossRef]
29. Wu, X.; Wang, L.; Zheng, Z.; Sang, B.; Zhang, J.; Tao, X. An Entropy-based Measure of Fork Diversity and its Correlations with Open Source Software Projects' Received Contributions. *Empir. Softw. Eng.* **2025**, *30*, 111. [CrossRef]
30. Battulga, B.; Tsoodol, L.; Dovdon, E.; Bold, N.; Namsrai, O.E. Metric-based Defect Prediction from Class Diagram. *Array* **2025**, *27*, 100438. [CrossRef]
31. Kalyani, P.; Rao, C.P.; Goparaju, B.; Babu, K.K.; Kandimalla, P.C.R. BugPrioritizeAI for Multimodal Test Case Prioritisation using Bug Reports, Code Changes, and Test Metadata. *Sci. Rep.* **2026**, *16*, 1539. [CrossRef]
32. Tang, Y.; Du, Y.; Gao, J.B.; Li, A.; Yang, M.S. ISRLNN: A Software Defect Prediction Method Based on Instance Similarity Reverse Loss. *J. Syst. Softw.* **2026**, *235*, 112766. [CrossRef]
33. Guo, Y.; Bettaieb, S.; Casino, F. A Comprehensive Analysis on Software Vulnerability Detection Datasets: Trends, Challenges, and Road Ahead. *Int. J. Inf. Secur.* **2024**, *23*, 3311–3327. [CrossRef]
34. Tang, B.; Maruyama, K. PRCollector: Facilitating on-demand Collection of Pull Request Data from GitHub. *IEEE Access* **2025**, *13*, 150608–150622. [CrossRef]
35. Özçevik, Y.; Altay, O. MetricHunter: A Software Metric Dataset Generator Utilizing SourceMonitor upon Public GitHub Repositories. *SoftwareX* **2023**, *23*, 101499. [CrossRef]
36. Li, J.; Wen, Y.; Liu, J.; Zeng, B.; Mirjalili, S. GFTrans: An on-the-fly Static Analysis Framework for Code Performance Profiling. *Front. Big Data* **2026**, *9*, 1779935. [CrossRef]
37. Wang, J.; Lu, S.; Wang, S.H.; Zhang, Y.D. A Review on Extreme Learning Machine. *Multimed. Tools Appl.* **2022**, *81*, 41611–41660. [CrossRef]
38. Kaur, R.; Roul, R.K.; Batra, S. Multilayer Extreme Learning Machine: A Systematic Review. *Multimed. Tools Appl.* **2023**, *82*, 40269–40307. [CrossRef]
39. Altay, O.; Ulas, M.; Alyamac, K.E. DCS-ELM: A Novel Method for Extreme Learning Machine for Regression Problems and a New Approach for the SFRSCC. *PeerJ Comput. Sci.* **2021**, *7*, e411. [CrossRef] [PubMed]

40. Singh, D.; Singh, B. Investigating the Impact of Data Normalization on Classification Performance. *Appl. Soft Comput.* **2020**, *97*, 105524. [CrossRef]
41. Ulas, M.; Altay, O.; Gurgenc, T.; Özel, C. A New Approach for Prediction of the Wear Loss of PTA Surface Coatings using Artificial Neural Network and Basic, Kernel-Based, and Weighted Extreme Learning Machine. *Friction* **2020**, *8*, 1102–1116. [CrossRef]