International Journal of Computer Engineering and Data Science (ISSN: 2737-8543)
Volume 5–Issue 2, April - June 2026

Software Evolution Prediction Using Machine Learning Algorithms

Rajeeb Sankar Bal[1], Jibendu Kumar Mantri[2]

[1] PhD Scholar, Department of Computer Application, MSCB University, Odisha, India.

[2] Professor, Department of Computer Application, MSCB University, Odisha, India.

E-mail: rajiv.s.bal@gmail.com , jkmantri@gmail.com

Article history

Received Mar 7, 2026
Revised Apr 14, 2026
Accepted Jun 29, 2026
Published Jul 01, 2026

ABSTRACT

Software evolution represents the most time-consuming phase in the software development life cycle (SDLC) after a software release. Further, it covers an increasingly significant role in modern software development practices (e.g., Agile, Development and Operations (DevOps), Continuous Integration/Continuous Delivery or Continuous Deployment (CI/CD)) and web development (e.g., React, Next.js, etc.), where teams contribute more effort in improving and maintaining existing systems than building new ones. These evolutionary actions such as modifying and adding features, incorporating contributions, and fixing issues generate large software project data that demonstrate the dynamics of software development. Software evolution in machine learning (ML) techniques comprises the unending adaptation of models, data, and pipelines to continue performance under changing domains. In this paper, we are predicting the software evolution by analyzing features such as number of days, revision, fix revision, percentage of fix revision and, average line of code (LOC) cover with Simple Linear Regression, Multiple Linear Regression, Polynomial Regression, and Support Vector Regression (SVR), are methods of ML. Thus, it is based on the results obtained a comparative study is conducted to evaluate and analyze model accuracy.

Keywords: ***Software Evolution, Artificial Intelligence (AI), Machine Learning (ML), Regression.***

I. INTRODUCTION

Manny Lehman was introduced the concept of soft-ware evolution and illustrated the growth characteristics of software systems in 1969. Later, the term “evolution” was widely adopted to software development applications. Laszlo Belady and Manny Lehman specified a set of principles, known as Lehman’s Laws, to explain the evolution of software systems. Software evolution is an important phase in the SDLC. During the SDLC, software undergoes major changes; therefore, software evolution is required. Software evolution with-in the SDLC occurs when maintenance activities such as bug fixing, adding features, and refactoring lead to changes in the software. The refactoring improves the internal structure of the software without changing its behavior. These activities influence requirements, de-sign, and development, helping the software remain maintainable and adaptable over time in the Figure 1.

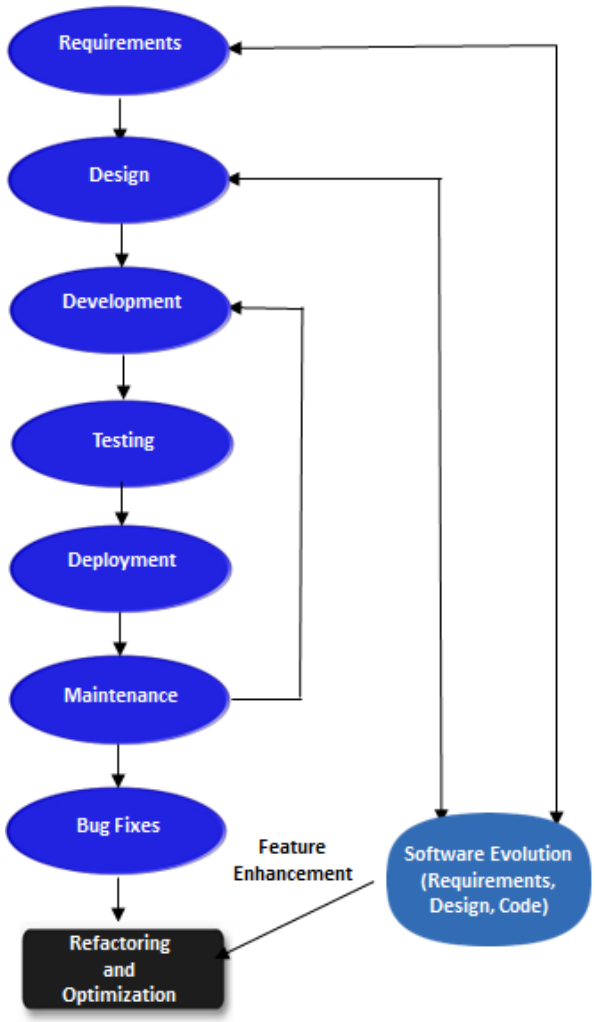

Figure 1. The software evolution in the SDLC showing feedback from maintenance to requirements, design, and development.

The use of AI techniques in software engineering has shown promising results. We have applied AI to support various software engineering activities, such as designing software, testing, maintenance, and decision making. These studies illustrate that AI can help auto-mate repetitive tasks, enhance efficiency, and operate the complex problems in SDLC. In [1], AI has a well-established history in computer science, beginning around 1950 when researchers aimed to execute hu-man-level machine intelligence. From the 1950s to the 1970s, both academia and industry worked extensively on command based and symbolic systems, driven by strong confidence about replicating human intelligence. However, by the 1980s, it became evident that these predictions were overly progressive, as such systems fought with real-world complexity and adaptability. In the 1990s, the realization led to a shift toward ML which systems learned from data instead of relying on explicitly programmed rules. With the increasing complexity of software systems in the 2000s, the challenge for automation and intelligent decision-making in-creased, motivating deeper integration of ML methods and AI techniques. In the last decade, evolutions in computing capability and data availability have further accelerated the application of ML and AI, particularly to present failures and inefficiencies in software systems and throughout the SDLC, covering development, testing, and maintenance. ML plays an important role throughout the SDLC by improving automation, accuracy, and decision-making at each phase. By learning from historical data and past software projects, ML helps enhance efficiency and minimize errors across the development process.

- Software Requirements Analysis Phase: ML techniques help analyze huge volumes of stakeholder inputs, user feedback, and historical requirement documents. ML can recognize patterns, detect ambiguities, prioritize requirements, and predict potential risks, thereby enhancing requirement quality and making down misunderstandings early in the development process.
- Software Architecture Design: During architecture design, ML helps in selecting appropriate architectural patterns by learning from previous system designs and functioning data. It can advance de-sign components, predict system behavior under separate workloads, and help optimize scalability, reliability, and performance.
- Software Implementation: In this phase, ML suggests developers by detecting coding errors, suggesting code improvements, and automating code generation tasks. ML based tools can study code quality and predict defect prone modules, helping developers/ programmers write more reliable software.
- Software Testing: ML significantly enhances software testing by automating test case generation, prioritizing test cases, and predicting areas with higher fault probability. It prepares intelligent testing strategies that reduce testing time while growing defect detection efficiency.
- Software Maintenance: In this phase, ML helps observe software behavior, detect anomalies, and predict future failures. By studying logs and usage data, ML assists proactive maintenance, reduces downtime, and suggests in managing software evolution more effectively.

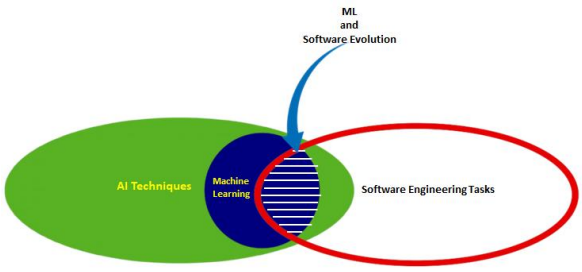

Figure 2. The interchange of AI techniques, ML, and software engineering tasks, highlighting how ML reinforces software evolution by validating

predictive maintenance, automated problem detection, and continuous development throughout the SDLC.

The Figure 2 illustrates how ML fits into the relationship between AI techniques, ML and software engineering tasks. Software evolution is an important part of soft-ware engineering that deals with making continuous enhancements and changes to software systems over time. ML methods can be applied to various software engineering tasks such as predicting defects, automating testing, and recognizing development patterns. As per the intersection, we have applied ML methods to support software engineering tasks, helps in managing and enhancing software evolution by providing tools that analyze data generated during the SDLC. Using ML methods in software evolution makes it easier to find out problems early, plan maintenance activities, and modify the software to changing requirements efficiently. The ML is subset of AI already shown in Figure 2. Now, ML is widely applied in the software industry/IT. It has changed the software process is developed, tested, and maintained by boosting traditional methods and preparing new types of applications and services. As ML methods continue to improve, they are expected to bring major changes to the software industry/IT. These developments will help build new applications and also enhance existing systems, leading to more methodical and intelligent software solutions [2]. Thus, the software engineering community looks for better ways to create high-quality software. However, in real projects, teams often consider more on implementing software quickly than on important activities like studying requirements, planning, and designing. This usually occurs because of limited period and lack of skilled human resources. ML can help in such situations by keeping tasks that normally necessary human judgment. By learning from existing data, ML can assist developers' programmers in making well decisions, enhancing software development processes, and growing the overall quality of software systems [3].

Contribution: This work provides to the understanding of software evolution by expressing how data generated during software development can be applied to study and predict maintenance activities. It shows that common development activities such as commits, pull requests, and contributor participation are strongly connected to issue generation, which reflects maintenance work. By extracting and studying evolution metrics from well-maintained open-source projects, the study highlights which factors most influence software growth and maintenance demands. In addition, it compares separate regression and ML methods and shows that non-linear addresses can better show the complex evolution patterns

II. BACKGROUND

Data Science Process: Data science is a broad field that brings together the processes, theories, concepts, tools and technologies that enable us to review, visualize, analyse and extract valuable knowledge and information from raw data. These have been developed in various disciplines and sub-disciplines in Figure 3. such as statistics, machine learning, data management, human computer interactions (HCI) and business intelligence systems. While databases were traditionally used to store structured data and query efficiently, data science tools aim at deriving insights and predictions even from unstructured, noisy and complex datasets. The applications of data science are wide ranging and expanding to include businesses, governance scientific studies, politics, environment and so on.

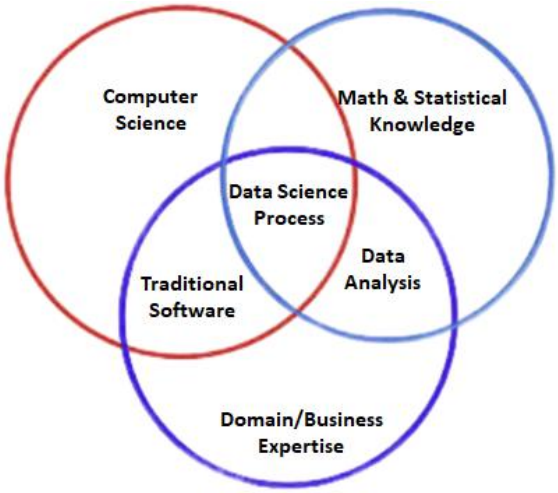

Figure 3. Drew Conway's Venn Diagram of Data Science Process.

Data Analysis study is consistent with the scientific discovery process. It begins with an interesting question that we are seeking to answer and the type of analysis we would like to perform. As we know from the history of science, asking the right question involves a lot of ingenuity. We then gather data in the form of facts, measurements, transaction histories, images, videos, articles, graphs etc. pertinent to the question. We then explore the data by performing descriptive analysis where we compute aggregate measures of data such as its mean, variance etc. We also explore relationships between different variables in our data. At this stage, visual descriptions of data are very useful in the form of distribution plots, scatter plots, heatmaps etc. We then attempt to model the data to perform various kinds of analysis: predictive, causal, inferential etc. Finally, we communicate the findings of the study, report the results, and generate recommendations ac-companied with various visualizations [4].

Machine Learning: It is no doubt that ML is increasingly gaining popularity and has become the hottest trend in the tech industry. ML is incredibly powerful to make predictions or calculate suggestions based on large amounts of data. ML learning is an application of Al that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. learning focuses on the development of computer programs that can access data and use it learn for themselves.

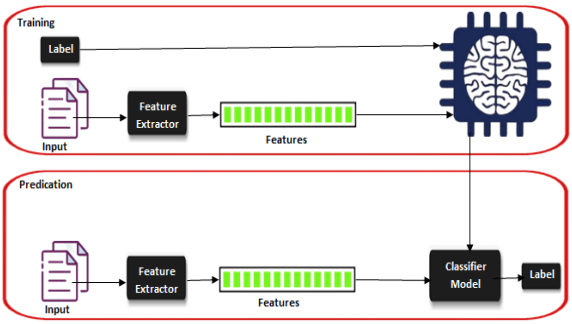

Figure 4. Example of a figure caption. (figure caption)

The process of learning begins with observations or data (Training Data), such as examples, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. The primary aim is to allow the computers learn automatically without human intervention or assistance and adjust actions accordingly shown in Figure 4.ML algorithms are often categorized as supervised or unsupervised. Supervised ML algorithms can apply what has been learned in the past to new data using labeled examples to predict future events. Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values. The system is able to provide targets for any new input after sufficient training. The learning algorithm can also compare its output with the correct, intended output and find errors in order to modify the model accordingly. Unsupervised ML algorithms are used when the information used to train is neither classified nor labeled. Unsupervised learning studies how systems can infer a function to describe a hidden structure from unlabeled data. The system doesn't figure out the right output, but it explores the data and can draw inferences from datasets to describe hidden structures from unlabeled data [5].

III. RELATED WORK

In [6], software evolution is recognized as a critical analysis area due to the substantial attempt required for post release maintenance and continuous system improvement. Predicting change-prone components plays a major role in managing maintenance activities and enhancing overall software quality. Earlier studies commonly relied on software metrics extracted from a single previous release to recognize components likely to change. Further ML methods based on these metrics have achieved promising prediction capability, such methods often overlook the continuous and dynamic nature of software evolution. As software systems evolve, metric values reach across releases, and not all changes carry the same level of importance. Recent analysis emphasizes the significance of analyzing multiple evolutionary snapshots to express long-term trends and recognize meaningful changes. However, many existing proposals do not explicitly distinguish between statistically significant metric forms and minor fluctuations. This restriction highlights the require for evolution-aware prediction models that include meaningful metric changes to improve the accuracy and reliability of change proneness prediction. in [7], the software defect prediction has been widely analysed as a means to enhance the software quality and reduce development costs. Prior analysis has shown that early distinguishing of defects can significantly reduce rework, maintenance efforts, and overall project timelines. Several predictive methods have been used to recognize potential soft-ware flaws, often evaluated applying performance metrics such as accuracy, precision, recall, and F-measure. Comparative analysis of multiple algorithms highlights differences in prediction success, enabling developers to select proposals that maximize defect detection while minimizing false positives. By integrating defect prediction into the development lifecycle, organizations can proactively address issues, enhance both software reliability, and customer satisfaction. In [8], we have studied the importance of analyzing mobile app analysis to support software evolution and maintenance activities. User reviews provide valuable discernments for a change of software development tasks, involving release planning, requirements engineering, and recognizing potential improvements or defects in applications. However, the huge volume of analysis, particularly for widely applied apps, makes manual extraction of relevant information impractical and time-consuming. Also, we address the challenge, several analysis efforts have investigated automated methods, including ML and statistical techniques, for detecting analysis that are pointed to software development activities. Traditional approaches typically rely on multi-class classification, which needs labeling both relevant and irrelevant analysis, thereby growing the effort and resources required for dataset preparation. More recent work has considered on approaches that reduce the dependency on huge labeled datasets, leveraging either smaller annotated sets or improved feature representations of analysis, such as textual content, metadata, or combined multimodal features. These methods aim to enhance the efficiency and accuracy of relevant analysis detection, enabling developers to effectively deploy user feedback for continuous software enhancement, informed decision-making, and well maintenance planning. In [9], we have studied the increasing complexity of software systems has intensified the require for effective software defect prediction (SDP) methods. Traditional manual fault assessment techniques are often resource-intensive and impractical for huge-scale soft-ware projects. we presented these limitations; several ML and heuristic-driven methods have been proposed to enhance defect detection accuracy and efficiency. Techniques such as neural networks, genetic algorithms, and feature engineering planning including resampling and normalization have been deployed to handle challenges like class imbalance, convergence issues, and overfitting. Empirical analyses on benchmark defect datasets have illustrated that methods combining adaptive learning with heuristic optimization can outperform classical methods in metrics such as accuracy, precision, recall, and F-measure. These improvements high-light the potential of integrated ML and optimization techniques to improve SDP, enabling more reliable and secure software systems. In [10], it is presented the software quality prediction, noting that while prior analysis has

extensively explored this topic using ML, most studies consider on evaluating the performance of accurate models or algorithms. For example, various works compare classifiers such as artificial neural net-works, support vector machines, decision trees, k-nearest neighbors, and Naïve Bayes across multiple datasets to recognize high-risk modules and direct targeted testing. These studies illustrate that ML based predictors can correctly detect potential defects early in the SDLC, decreasing costs and enhancing product reliability. In contrast, our analysis shifts the consider from purely model performance to the practical integration of ML components into software engineering processes. By contrast comparing classifiers, we explore the tasks, activities, and process changes that arise when ML based quality prediction is included into real development workflows. By recognizing essential development tasks and proposing structured checklists, our analysis provides actionable guidance for practitioners and demonstrate a foundation for future study on systematically embedding ML into software development and maintenance practices. In [11], it is presented the valuable classifications of software evolution metrics, distinguishing between predictive analyses managed before software evolution to recognize evolution critical, sensitive, or prone components and retrospective analyses conducted after evolution, encompassing both software and process perspectives. These analyses offer a structured understanding of how metrics can support designing, monitoring, and controlling software evolution and suggest directions for future study on software evolution metrics. In [12], we have studied that the open-source software (OSS) has attracted marked attention in software engineering study over the past decade due to its publicly accessible code and collaborative development model. Also, we have extensively ana-lysed well-known OSS projects and examined repositories hosted on environment like GitHub to study project evolution, functionality, and contribution models. However, the evolution processes and elementary rules governing OSS projects in GitHub stay relatively under-explored. We present this gap; computational model-ling methods such as cellular automata and optimization algorithms have been used to simulate OSS project evolution and extract elementary rules. Empirical studies applied historical GitHub data have illustrated that such models can reach high accuracy in increasing project evolution, providing vision into development dynamics and suggesting more effective management of OSS projects.

IV. Proposed Methodology

In [13], we consider that software evolution has become a central view of modern software development, as developers/programmers increasingly work with existing systems rather than creating new software from the ground up. Much of their effort is assigned to ex-tending functionality, enhancing performance, and correcting defects in connected codebases. As a result, understanding and managing software evolution is need for maintaining software quality and ensuring that systems survive to meet changing user essentials. In [10], software evolution is a basic characteristic of software systems, driven by continuing changes in requirements, technology, and operational environments. While modern development tools prepare meaningful support, opportunities remain for more robust mechanisms that express and leverage evolutionary knowledge. Version control systems (VCS) represent one of the most evolve components of modern soft-ware development workflows, enabling structured collaboration across allocated teams. The emergence of ML for software engineering has extended interest in richer evolutionary data. Techniques such as representation learning, code embed-dings, and graph neural networks enable the discovery of complex relationships between code changes, soft-ware defects, and system architecture. ML has shown capacity in:

- Predicting software evolution directions
- Recommending reengineering opportunities
- Detecting high-risk commits
- Automatically connecting software evolution artifacts across systems (e.g., Project, Period, # of revision)

GitHub is an online environment built around the Git VCS. It plays an important role in software evolution because it keeps the total history of a project every change, update, bug fix, and feature addition. This makes it easy for developers to trace how software increases over time. Also, GitHub supports software evolution by producing:

- Predicting Version tracking: Each change is recorded, enabling developers to see how the software has evolved.
- Branching and merging: Teams can build on new features or fixes in parallel and merge them safely.
- Collaboration tools: Pull requests, comments, and code analysers help arrange evolution across distributed teams.
- Issue tracking: Bugs, enhancements, and tasks are connected to the code changes that implement them.
- Visibility and transparency: The complete evolution of a project becomes understandable to anyone.

Finally, GitHub helps manage and document the entire SDLC changes, making it necessity tool for understand-ing and conducting software evolution. As per software evolution applying ML algorithms, we consider metrics in Table 1.

TABLE I. The sketch of each dependency.

Metric	***Role in Software Evolution***	***Dependency***
Project	software project name being analyzed.	Link depends on Name
Period	The data was collected for that project during time span.	It helps track of evolution in the long run.
No. of Days	The total number of days in starting to ending period.	It used to normalize activity.
# of revision	The total number of commits (i.e., code revisions) made in that period.	It indicates development activity.

Metric	Role in Software Evolution	Dependency
# of fix revision	Specifically, the number of revisions related to bug fixes: $\% = \frac{\text{Fix Revisions}}{\text{Total Revisions}} \times 100$	It shows how much attempt goes into maintenance vs new development.
% of fix revision	The Average size of code changes per revision.	It calculates complexity or scale of changes.

The proposed model gives an end-to-end ML workflow for studying and classifying comments released from software evolution datasets. As illustrated in Figure 5, the system is arranged into three important components: data preprocessing, model construction, and model evaluation and deployment.

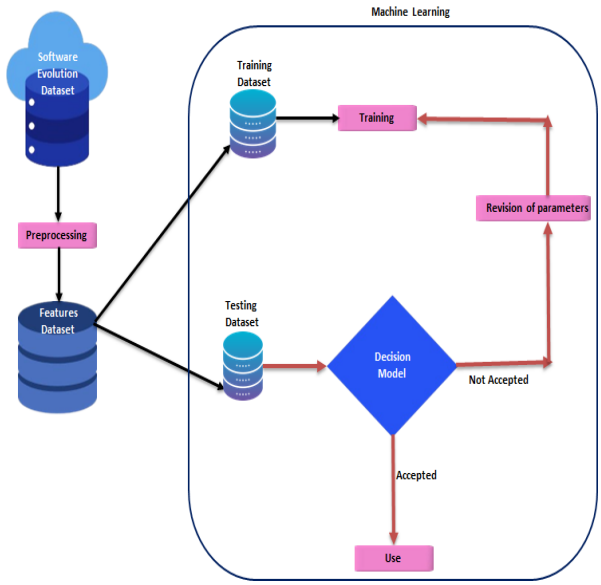

Figure 5. The workflow of ML model, showing the preprocessing of software evolution data, feature generation, model construction, evaluation, and refinement.

A. Data Preprocessing

The process starts with the software evolution dataset, which consists raw textual comments along with metadata such as dates, authors, and sources. Because raw data is frequently noisy and unstructured, a preprocessing module is used to convert it into a clean and consistent representation suitable for ML tasks. The preprocessing phase involves:

- Cleansing: Separating unnecessary characters, duplicated content, stop words, and noise such as URLs or punctuation.
- Normalization: Using tokenization, lowercasing, and optional stemming or lemmatization to regulate the text.
- Filtering: Excluding incomplete, unrelated, or low-quality entries that may affect model presentation.
- Feature generation: Converting the processed text and metadata into numerical feature vectors applying appropriate encoding techniques.
- The output is a structured dataset, in feature which makes the input to the learning component.

B. Model Construction Using ML

The features dataset is divided into two sub process-es: training and testing. In the training phase, a ML model learns patterns, relationships, and discriminative features from the training data. The learning algorithm adjusts its internal parameters to conduct a decision model capable of assigning labels or categories to new, unseen comments. This model express complex patterns in the feature space and represents the predictive rules derived from the training data.

C. Model Evaluation and Parameter Refinement

After the decision model is trained, it is evaluated on the testing subset to compute how well it generalizes beyond the data applied during learning. The execution metrics such as accuracy, precision, recall, and F-measure are computed.

- If the model achieves a performance level that meets or exceeds the required threshold, it is accepted and prepared for real-world use.
- If the performance is unsatisfactory, the system initiates an iterative refinement process. In this stage, the learning algorithm's parameters, pre-processing configurations, or feature-generation settings are adjusted to improve prediction quality.

Hence, the iterative loop in Figure 6 attains satisfactory performance.

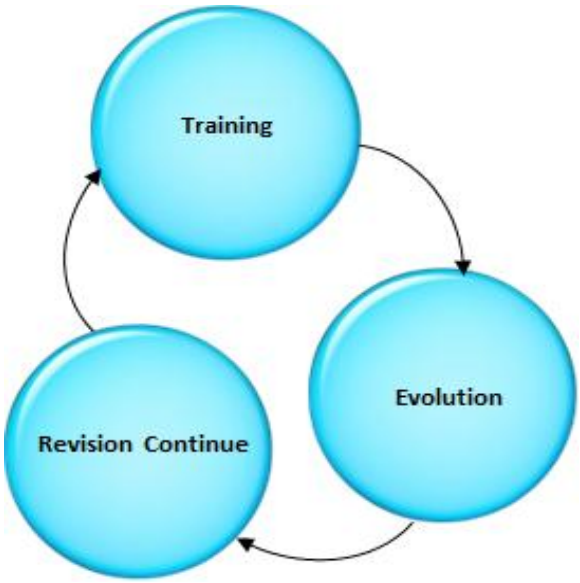

Figure 6. The loop of Training, Evaluation, and Revision.

D. Deployment

After validated, the final decision model is deployed for practical handling. It can automatically study and classify new software evolution comments, supporting maintenance, comprehension, and development-related tasks.

V. Result Analysis

A. Data Cleaning for ML

In [14], ML systems depend densely on data quality. If the data applied for training contains errors, the model may build inaccurate or incorrect results. Similarly, poor-quality data during deployment can conduct to wrong predictions and affect major decisions. Therefore, data cleaning is necessary throughout the entire machine learning lifecycle to ensure reliable and accurate outcomes.

B. Regression Algorithms

Regression models attempt to uncover the relationship between one or more input variables (features) and a target output value. The most basic mathematical representation of a regression model is:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \quad (1)$$

In this expression, $x_1, x_2, \ldots, x_n$ represent the input features, $w_1, w_2, \ldots, w_n$ are the coefficients that determine how strongly each feature influences the prediction, w_0 is the intercept term, $\hat{y}$ is the predicted output generated by the model.

The coefficients and the intercept are learned from training data, and many machine-learning libraries (such as Scikit-Learn) store them using attributes like coef_ (for coefficients) and intercept_ (for the bias).

C. Types of Regression Models

There are several types of regression methods with different kinds of relationships between variables:

a) Simple Linear Regression: It is used when there are a single independent variable and a linear relation-ship with the output.

b) Multiple Linear Regression: It extends linear re-gression to multiple input features, allowing for more complex modeling of relationships.

c) Polynomial Regression: It transforms the original features into polynomial terms (e.g., x^2,x^3), en-abling the model to capture curved or nonlinear patterns.

d) Support Vector Regression (SVR): It applied the rules of support vector machines to predict contin-uous values while maintaining robustness to outliers.

e) Decision Tree Regression: A tree-structured model that splits data into decision nodes, making predic-tions deployed on learned thresholds.

f) Random Forest Regression: A collection method that combines many decision trees to enhance prediction accuracy and reduce overfitting.

g) Non-Linear Regression: General category for models that cannot be demonstrated in a straight-line form, useful when relationships are intrinsically curved.

h) Regularized Regression Methods: These tech-niques introduce penalties to prevent overfitting: Ridge Regression and Lasso Regression.

i) Support Vector Machine Regression: Another term often used interchangeably with SVR; focuses on maximizing margin while tolerating small predic-tion errors.

The Regression is a statistical / ML method used to model relationships between software-evolution metrics shown figure 7.

```
import pandas as pd
import matplotlib.pyplot as plt
```

```
df = pd.read_csv("bugs.csv")
df.head()
```

	Project	Period	No. of Days	# of revision	# of fix revision	% of fix revision	Average LOC
0	tapestry3	2000-07-24 ~ 2024-12-08	8903	1985	351	17.68	54950000
1	apr-iconv	2000-11-20 ~ 2019-01-01	6615	210	43	20.48	2539000
2	tapestry4	2000-07-24 ~ 2024-12-08	8903	3793	929	24.49	76627000
3	sling-old-svn-mirror	2017-10-21 ~ 2018-06-29	251	4	0	0.00	86054000
4	xalan-j	1999-11-09 ~ 2019-05-24	7135	4561	1811	39.71	55092000

```
df.tail()
```

	Project	Period	No. of Days	# of revision	# of fix revision	% of fix revision	Average LOC
14	directory-studio	2006-11-16 ~ 2026-02-25	7040	5829	1846	31.67	639469000
15	felix	2005-07-19 ~ 2020-03-04	5342	15556	2559	16.45	96620000
16	chainsaw	2000-12-14 ~ 2014-11-20	5089	934	201	21.52	2704000
17	maven-wagon	2004-03-30 ~ 2026-03-19	8023	1402	223	15.91	5260000
18	maven-resources	2006-11-19 ~ 2018-05-11	4190	170	19	11.18	188000

Figure 7. The software evolution of bugs dataset is a statistical and ma-chine-learning technique applied.

We apply corr() method to find ne the relationships between revision-related columns or attributes or variables, a Pearson correlation study was conducted using numerical features in the bugs dataset. The result shown in figure 8.

```
data.corr(numeric_only=True)["# of revision"]
```

```
No. of Days          0.200318
# of revision        1.000000
# of fix revision    0.913673
% of fix revision    0.265626
Average LOC          0.470081
Name: # of revision, dtype: float64
```

Figure 8. The correlation results of software evolution metrics based on "# of revision"

From the figure 8, the variable or column number of revisions shown a perfect positive correlation with itself (r = 1.000) which is serving as a baseline for correlation. Also, a very strong positive correlation was found be-tween the number of revisions and the number of fix revisions (r = 0.914) which is indicated that considerable of revisions are correlated with bug fixes or corrective changes. such a high correlation proposes that projects with continuous revisions are associated with by maintenance and activities of error correction. The average lines of code (LOC) shown a moderate correlation with the number of revisions (r = 0.470) which im-plies that

larger codebases incline to undergo more revisions, possibly due to increased complexity, greater probability of defects or functional expansion in the long run. In contrast, the percentage (%) of fix revisions shown a weak positive correlation (r = 0.266) which is slight tendency for projects with more revisions to have a higher proportion of fixes, the relationship is not strong, suggesting that other kinds of revisions (e.g., feature additions or refactoring).

We have presented the visualizations of bug database metrics for comparison which understand the relation-ship between development time, revisions, and code. The comparison between project with two components i.e., number of days and revisions which shown the projects with longer active durations tend to present higher revision counts. Also, the relationship is not line-ar, as some projects shown high revision activity within moderate time spans in figure 9.

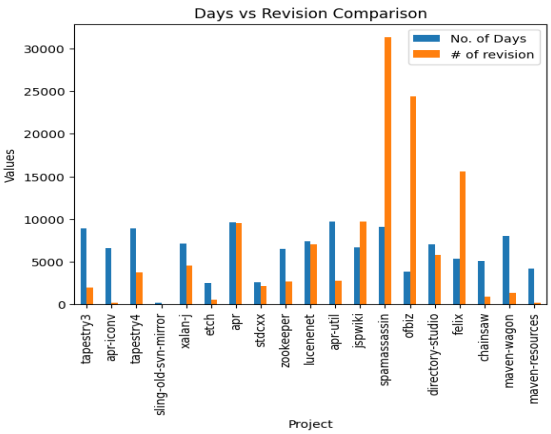

Figure 9. The number of days vs revisions across projects

The comparison between project with three components i.e., number of days, revision, and fix revision which shown fix-related changes closely follow total revisions, indicating that a significant proportion of software development effort is allocated to bug fixing in figure 10.

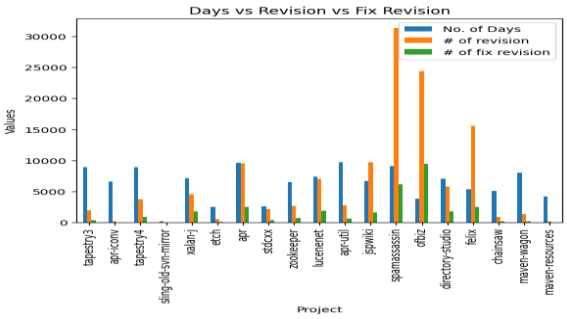

Figure 10. Example The number of days vs revisions vs fix revision across projects

The comparison between project with four components i.e., number of days, revision, fix revision, and percent-age (%) of fix revision which presented total revision counts vary widely across projects, the proportion of fix-related work remains relatively stable which also indicated consistent maintenance behavior across different projects in figure 11.

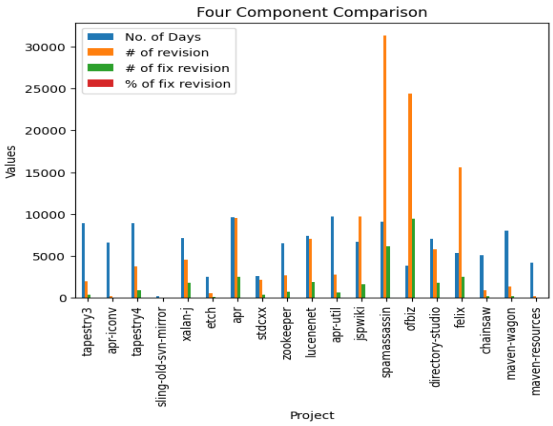

Figure 11. The number of days vs revisions vs fix revision vs percentage (%) of revision across projects

The comparison between project with five components i.e., number of days, revision, fix revision, percentage (%) of fix revision, and average line of codes (LOC) which highlighting that code size contributes to in-creased maintenance demand and a higher likelihood of defects in figure 12.

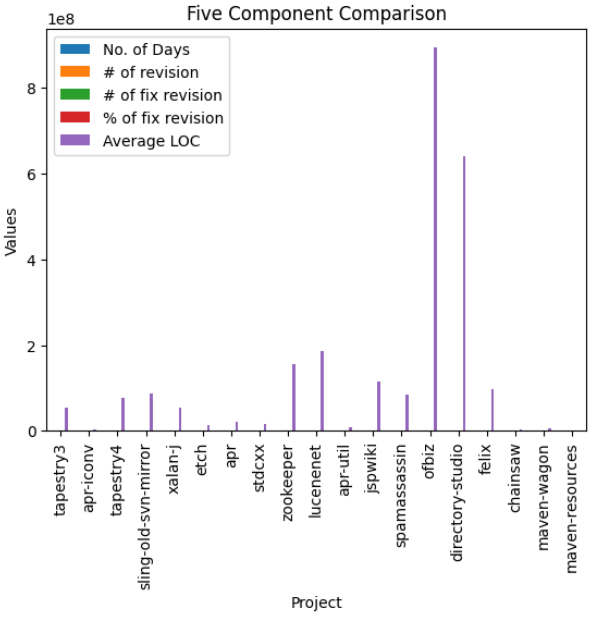

Figure 12. The number of days vs revisions vs fix revision vs percentage (%) of revision vs average LOC across projects

Where, Target is the column for predict, Predictors are the columns used to explain or estimate the target, b_0 is intercept, b_1, b_2, … are coefficients showing how strongly each metric affects the target.

Basically, the Linear regression model finds the best value for the incept and coefficient which results in a line that best fits the data. To see the value of the intercept and slop calculated by linear regression algorithm for our bugs dataset and find the following result.
R² Score: 0.7355421928952399
RMSE: 1996.6181883414072
Intercept: 1717.2870131054178
Coefficients:
No. of Days: 0.6225608650639295
# of fix revision: 4.112294390224693
% of fix revision: -224.4682487134138
Average LOC: -7.0845958347269015e-06

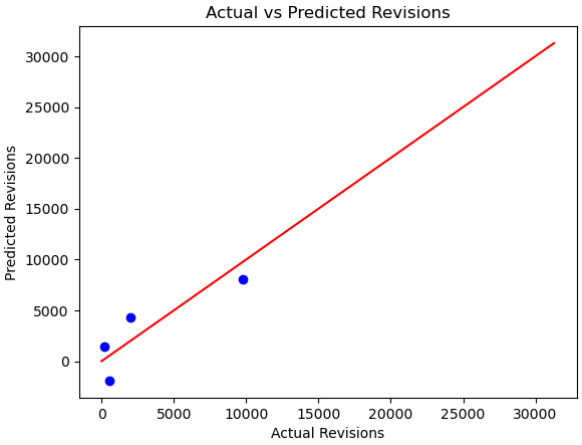

Figure 13. The actual vs. predicted revisions

From the Figure 13, we compare the actual with the predicted values, where points in blue colour which is closer to straight line in red colour indicate more accurate predictions.

Multiple Linear Regression (MLR) is a method used to predict one outcome (called the response or dependent variable) by using several factors (called independent or explanatory variables). Instead of using only one variable to make a prediction, MLR uses two or more variables together to understand how they influence the result. The general formula for MLR is:

$$y_i = B_0 + B_1x_{i1} + B_2x_{i2} + \cdots + B_Dx_{iD} + \varepsilon \quad (3)$$

Where y_i is the value we want to predict, B_0 = intercept (baseline value). $B_1, B_2, \ldots, B_D$= coefficients that show how much each variable affects the result, $x_{i1}, x_{i2}, \ldots, x_{iD}$ are the input variables and ε is the error.
The MLR is applied to the software evolution dataset for defining of independent and dependent columns.
In MLR, a model that predicts a dependent variable or target variable using two or more independent variables or predict variables. The following final result gives each feature contributes with its own coefficient is calculated.
R² Score: 0.7355421928952399
RMSE: 1996.6181883414072
Intercept (b0): 1717.2870131054178
Coefficients (b1, b2, b3, ...):
No. of Days: 0.6225608650639295
# of fix revision: 4.112294390224693
% of fix revision: -224.4682487134138
Average LOC: -7.0845958347269015e-06

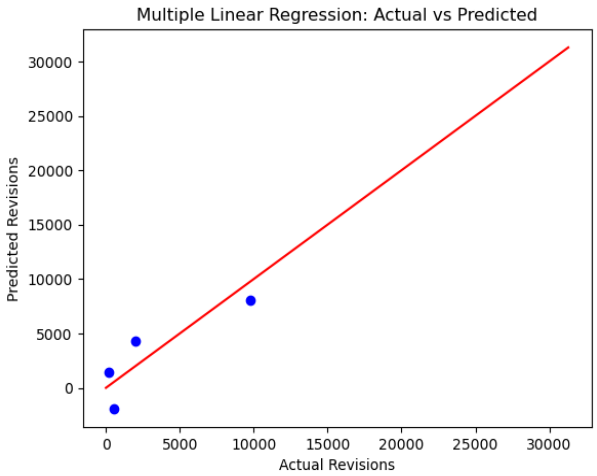

Figure 14. The Actual vs. predicted revisions using MLR, where points closer to the straight line in red colour indicate better model accuracy

From Figure 14, we compare the actual with the predicted revisions using MLR, where points in blue colour which is closer to straight line in red colour indicate more accurate predictions.

Polynomial Regression: It is an extension of linear regression used when the relationship between the independent variable(s) and the dependent variable is curved rather than straight. Unlike simple linear regression, where the predicted value changes at a constant rate with respect to the predictor, in polynomial regression the change can vary non-uniformly it can speed up, slow down, or even reverse direction depending on the value of the predictor.
The general equation of a polynomial regression is:

$$y = \theta_0 + \theta_1x + \theta_2x^2 + \theta_3x^3 + \cdots + \theta_nx^n \quad (4)$$

Where, y is the target variable we want to predict, x is the predictor variable, θ_0 is the intercept (baseline value of $y$when $x = 0$), $\theta_1, \theta_2, \ldots, \theta_n$ are the coefficients (weights) for each term, which determine how strongly each power of $x$influences y, n is the degree of the polynomial, indicating the highest power of x. If the value of degree n increases, the equation (4) includes more higher-order terms, allowing it to fit more complex, wavy patterns in the data. However, higher-degree polynomials also make the model more complexed and can sometimes lead to overfitting, where the model represents noise instead of the true underlying pattern.
The following result of Polynomial Regression is indicating a model which is an excellent fit and very strong predictive performance. Because 99.45% of the variance in the data.
R² Score: 0.9944934338069722
RMSE: 288.1092877985139

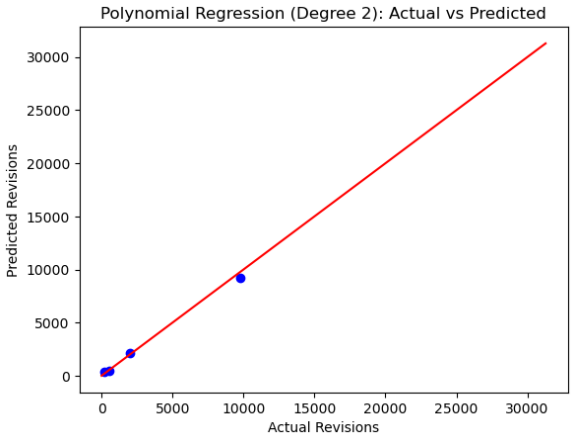

Figure 15. The actual vs. predicted revisions using polynomial regression

From Figure 15, shows the relationship between actual and predicted revisions; most points lie close to diagonal line in red colour which is indicating the polynomial regression model makes highly accurate predictions with minimal error.

Support Vector Regression (SVR) is a goal to find a function $f(x)$ that predicts the target variable y with at most ε deviation for all training points, while keeping the function as flat as possible. The ε-insensitive loss function is defined as:

$$L_\epsilon(y, f(x)) = \begin{cases} 0, & \text{if } |y - f(x)| \leq \epsilon \\ |y - f(x)| - \epsilon, & \text{otherwise} \end{cases} \quad (5)$$

SVR models represent linear and non-linear relationships by mapping data into higher-dimensional spaces, ensuring that most points lie within the ε-tube while penalizing outliers.
The following result of SVR model it has a moderate ability to predict revisions but is not highly accurate compared to other models.
R^2 Score: 0.6269198519955841
RMSE: 2371.47143053105

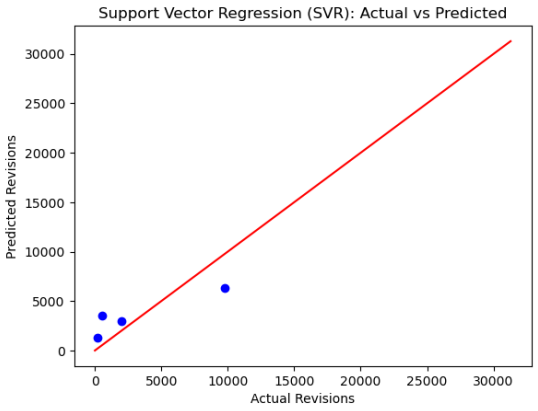

Figure 16. The actual vs. predicted revisions using SVR

From Figure 16, we compare the actual and predicted revisions using SVR, where points are more spread from the straight line in red colour which is indicating moderate prediction accuracy and noticeable deviation between predicted and actual values.

VI. Conclusions

In this paper, we have demonstrated different regression techniques provide varying levels of accuracy in predicting issue occurrence based on software evolution metrics such as number of days, revision, fix revision, percentage of fix revision and, average line of code. Simple Linear Regression offers a basic understanding of the linear relationship between revision and other variables but lacks precision due to its limited scope. Multiple Linear Regression enhances the model by incorporating more predictors. The R^2 score is 0.73 i.e., 73%. Polynomial Regression further enhances the model by capturing non-linear relationships among the predictors. The R^2 score is 0.9945 specifies 99.45% of the variance in the data which is indicating an excellent fit and very strong predictive performance., but the model's predictions differ from actual values by about 288 revisions, showing some prediction error but still relatively low compared to the overall scale. The SVR with an R^2 score is 0.62 and a higher RMSE compare with other regression models. Hence, the findings highlight that non-linear machine learning models, particularly polynomial regression is highly suitable for predicting issue trends in software evolution. These models provide more reliable insights for maintainability planning, resource allocation, and understanding how collaborative development behaviors contribute to software growth and complexity.

Acknowledgment

We would like to acknowledge Professor Jibendu Kumar Mantri for his constant encouragement, expert guidance, and valuable contributions to this work.

References

[1] M. Navaei, N. Tabrizi, Machine Learning in Software Development Life Cycle: A Comprehensive Review, 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022), pages 344-354, Science and Technology Publications, 2022, https://doi.org/10.5220/0011040600003176.

[2] V. K Thatikonda, Machine Learning in Software Engineering: Practical Applications and Impact on The Modern Software Industry, International Journal of Artificial Intelligence & Machine Learning, Volume 2, Issue 1, January- December 2023, 35-43, https://doi.org/10.17605/OSF.IO/6KX3.

[3] S. Shafiq, A. Mashkoor, C. M. Dorn, A. Egyed, A Literature Review of Using Machine Learning in Software Development Life Cycle Stages, IEEE 2021, https://doi.org/10.1109/ACCESS.2021.3119746.

[4] G. R. Gupta, N. k. Nagwani, Data Analytics and Visualization, PHI Learning Pvt. Ltd., 2026.

[5] S. Saurav, Data Science Machine Learning, Eka Publishers, 2020.

[6] R. Shatnawi, predicting software change-proneness from software evolution using machine learning methods. Interdisciplinary Journal of Information, Knowledge, and Management, 2023, 769-790, https://doi.org/10.28945/5193.

[7] M. J. Sadiq, S. R. Sufian, K. S. Reddy, M. Vinisha, Software Defect Estimation Using Machine Learning, International Journal for Research in Applied Science & Engineering Technology (IJRASET), 2020, https://doi.org/10.22214/IJRASET.2022.44551.

[8] M. P. S. Gôlo, A. F. Araújo, R. G. Rossi, R. M. Marcacini, Detecting relevant app reviews for software evolution and maintenance through multimodal one-class learning, Information and Software Technology, 2022, https://doi.org/10.1016/j.infsof.2022.106998.

[9] S. Hassen. A. Yazici, A. Mishra, A New Method for Software Defect Prediction Based on Optimized Machine Learning Techniques, Eurasian Journal of Engineering and Technology (EJET), 2023.

[10] S. M. Reza, M. M. Rahman, H. Parvez, O. Badreddin, S. A. Mamun, Performance Analysis of Machine Learning Approaches in Software Complexity Prediction, Proceedings of International Conference on Trends in Computational and Cognitive Engineering, Advances in Intelligent Systems and Computing 1309, Springer Nature Singapore Pte Ltd. 2021, https://doi.org/10.1007/978-981-33-4673-4_3.

[11] S. Goyal, P. K. Bhatia, Software Quality Prediction Using Machine Learning Techniques, Innovations in Computational Intelligence and Computer Vision, Advances in Intelligent Systems and Computing 1189, Springer Nature Singapore Pte Ltd. 2021, https://doi.org/10.1007/978-981-15-6067-5_62

[12] H. Wang, H. Ji, Evolution Model of Open-Source Software Projects in GitHub, 2022 IEEE 2nd International Conference on Software Engineering and Artificial Intelligence (SEAI), IEEE, 2022, https://doi.org/10.1109/SEAI55746.2022.9832099.

[13] M. Kim, N. Meng, T. Zhang, Handbook of Software Engineering, Springer Nature Switzerland AG, 2019, https://doi.org/10.1007/978-3-030-00262-6_6.

[14] I. F. Ilyas, T. Rekatsinas, Machine Learning and Data Cleaning: Which Serves the Other? Journal of Data and Information Quality, ACM 2022, https://doi.org/10.1145/3506712ACM.