Interdisciplinary Journal of Information, Knowledge, and Management

An Official Publication of the Informing Science Institute InformingScience.org

IJIKM.org

Volume 18, 2023

PREDICTING SOFTWARE CHANGE-PRONENESS FROM SOFTWARE EVOLUTION USING MACHINE LEARNING METHODS

Raed Shatnawi | Jordan University of Science and Technology, Irbid, Jordan | raedamin@just.edu.jo

ABSTRACT

Aim/Purpose	To predict the change-proneness of software from the continuous evolution using machine learning methods. To identify when software changes become statistically significant and how metrics change.
Background	Software evolution is the most time-consuming activity after a software release. Understanding evolution patterns aids in understanding post-release software activities. Many methodologies have been proposed to comprehend software evolution and growth. As a result, change prediction is critical for future software maintenance.
Methodology	I propose using machine learning methods to predict change-prone classes. Classes that are expected to change in future releases were defined as change-prone. The previous release was only considered by the researchers to define change-proneness. In this study, I use the evolution of software to redefine change-proneness. Many snapshots of software were studied to determine when changes became statistically significant, and snapshots were taken bi-weekly. The research was validated by looking at the evolution of five large open-source systems.
Contribution	In this study, I use the evolution of software to redefine change-proneness. The research was validated by looking at the evolution of five large open-source systems.
Findings	Software metrics can measure the significance of evolution in software. In addition, metric values change within different periods and the significance of change should be considered for each metric separately. For five classifiers, change-proneness prediction models were trained on one snapshot and tested on the next. In most snapshots, the prediction performance was excellent. For example, for Eclipse, the F-measure values were between 80 and 94.

Accepting Editor Maureen Tanner │ Received: June 26, 2023 │ Revised: August 19, September 16, September 21, 2023 │ Accepted: September 22, 2023.

Cite as: Shatnawi, R. (2023). Predicting software change-proneness from software evolution using machine learning methods. *Interdisciplinary Journal of Information, Knowledge, and Management*, *18,* 769-790. https://doi.org/10.28945/5193

	For other systems, the F-measure values were higher than 75 for most snapshots.
Recommendations for Practitioners	Software change happens frequently in the evolution of software; however, the significance of change happens over a considerable length of time and this time should be considered when evaluating the quality of software.
Recommendation for Researchers	Researchers should consider the significance of change when studying software evolution. Software changes should be taken from different perspectives besides the size or length of the code.
Impact on Society	Software quality management is affected by the continuous evolution of projects. Knowing the appropriate time for software maintenance reduces the costs and impacts of software changes.
Future Research	Studying the significance of software evolution for software refactoring helps improve the internal quality of software code.

Keywords software evolution, software metrics, change-proneness, machine learning

INTRODUCTION

Software evolution is inevitable for all software applications and to improve software quality. According to the ISO9000 definition, quality is defined as a degree "to which a set of inherent characteristics of an object fulfills requirements" (Témolé & Atanasova, 2023). Software quality describes the desirable attributes of software products and is paramount to the success of software projects, including software in maintenance. Software maintenance is a paramount activity in the software process and costs more than 50% of the total project costs (Malhotra & Khanna, 2019), accounting for 50–80% of software production expenses in the United States (Carr & Wagner, 2002). Software maintenance is driven by several factors including user requests, emerging technologies, and new platforms (Rajlich, 2014; Xie et al., 2017). Software maintenance activities have different types including adding new features, quality improvements, and system adaptation to new environments. These types of changes cause the software to grow, and changes make classes more prone to change. Understanding evolution patterns in software helps engineers understand how software grows. Software growth can be used in maintenance activities such as corrective maintenance for the next version or release (Kemerer & Slaughter, 1999). In previous research, software evolution trends were modeled using mathematical models, such as linear and sublinear models. These models provide insights into the growth of software. Evolution trends can be used to understand software quality and help in controlling the quality of software (Chatzigeorgiou & Melas, 2012; Israeli & Feitelson, 2010; Mens & Demeyer, 2008). For example, evolution metrics were used to build fault-prediction models using machine learning and regression classifiers (Illes-Seifert & Paech, 2010; Kaur et al., 2016). In addition, understanding software evolution helps to improve the maintainability of future releases. Therefore, more effective and efficient evolution analysis tools are required (Yu & Mishra, 2013). Agrawal and Singh (2020) studied the history of change and how ripple change happens. This study helps researchers in studying the future changeability pattern.

Keeping software in high quality is vital for large software systems that evolve for a long time. Researchers have studied and investigated how to improve software quality using important quality attributes such as fault-proneness (Illes-Seifert & Paech, 2010), maintainability (Giray et al., 2023), and reusability (P. Kumar et al., 2022). Software systems continue to grow while the quality of these systems continues to either improve or degrade (Agrawal & Singh, 2020). However, when software changes become significant it is not studied in previous literature. The research aims to study the relationship between software quality as measured using internal software metrics and the evolution of software using machine learning models to know when internal properties of software, such as coupling, and cohesion, become significantly different. In addition, using machine learning helps automate the change prediction and reduces the human-centric efforts of prioritizing the tasks of software development and maintenance. Hence, software activities affected by change, such as regression testing, refactoring, and maintenance, are directed to most change-prone classes.

Change-proneness was measured in the literature (e.g., Abbas et al., 2020; Bansal et al., 2022; Catolino & Ferrucci, 2019; Zhu et al., 2022) by taking differences between consecutive releases without considering the significance of the change. Change-proneness has been studied previously without considering the significance of software evolution or change at specific intervals or time windows. For a better definition of change-proneness, this research proposes to measure evolution using two methods: the number of files affected by evolution and the time when static metrics change significantly. These two methods are explored using both descriptive and visual analytics to identify the periods of significant changes in six object-oriented metrics. These metrics measure coupling, cohesion, inheritance depth and breadth, complexity, and response set for a class. The study was applied to the evolution of five large open-source systems that have evolved for more than three years (more than 90 biweekly snapshots). The results were used to build change-proneness prediction for ten snapshots that were identified as significantly changed from previous snapshots. Five well-known machine learning classifiers – Logistic Regression (LR), Naïve Bayes (NB), Nearest Neighbors (NN), Support Vector Machines (SVM), and Decision Trees (CART) – are used to provide evidence of change-proneness prediction. The training of each model was considered on one snapshot and tested on the next snapshot of software for a better validation of the model's performance. The performance of these classifiers was measured using precision, recall, and F-measure. The results were high and can be considered helpful in predicting future change-prone classes in software. The collected data were provided publicly for further investigation by D'Ambros et al. (2010).

Two research questions are proposed to understand software evolution using metrics.

RQ1: Which metrics are most affected by software evolution?

The research aims to know how software evolution affects software quality. Static internal metrics, such as object-oriented metrics, are used to assess software quality. Therefore, this research proposes measuring the effect of metrics on software quality by determining which classes have changed between two consecutive snapshots of software. Each metric's change effect is calculated separately. The number of classes that were different in two consecutive snapshots are counted, i.e., every two weeks, for each metric. The effect of evolution on each metric is graphically depicted with line charts and summarized with boxplots. Charts compare the six metrics and determine which are most affected by the change.

RQ2: When does the software quality change significantly?

For example, it is vital to know when couplings become significantly different, and whether couplings increase or decrease over time.

The most expensive activity in software production is software maintenance and evolution (Erlikh, 2000). Understanding how software evolves and when software differs significantly in size and quality is critical for understanding the impact of change on software. Understanding the impact of software evolution on software quality also requires identifying differences in other quality attributes. This research aims to answer the following sub-questions:

RQ2.1. When does coupling in software become statistically different from a measured version?

RQ2.2. When does software cohesion become statistically different from a measured version?

RQ2.3. When does the depth of inheritance in software become statistically different from a measured version?

RQ2.4. When does the breadth of inheritance in software become statistically different from a measured version?

RQ2.5. When does a set of responsibilities in software become statistically different from a measured version?

RQ2.6. When does the complexity of software become statistically different from a measured version?

The rest of the paper is structured as follows. In the next section, the related work is discussed and compared to previous work on software evolution and growth. Then the research methodology is presented including data collection and processing. The results of the research questions are then discussed and analyzed, and the change-proneness classifiers are built and evaluated. Finally, the work is concluded.

RELATED WORK

The change history of software systems has been studied by many previous researchers to understand software maintainability. Authors have studied change at different levels: implementation and design. Implementation changes were studied for the line of code changed, added, or deleted in software (Herraiz et al., 2013), while design and architecture changes were studied at the function and class levels such as method body changes, method additions, method deletions, and signature changes (Wermelinger et al., 2008). The study of software evolution helps in understanding software design and evolution patterns (Xing & Stroulia, 2004). Many previous works have shown a growth in the size or complexity of software systems stability (Chatzigeorgiou & Melas, 2012; Israeli & Feitelson, 2010). Researchers have studied the evolution of many properties of software. They found that minor releases are introduced usually to restore software familiarity and stability (Israeli & Feitelson, 2010). Chatzigeorgiou and Melas (2012) studied the growth in coupling measured from the network properties of systems. The authors found an exponential relationship between coupling and release time. However, the exponent was close to 1 and therefore the relationship is close to linear. The evolution of software was not utilized directly to predict software quality attributes such as change proneness.

Change-proneness prediction is studied by many researchers using regression and machine learning methods to classify classes into either change-prone or not change-prone as shown studies summarized in Table 1. Lindvall (1998) has studied the correlation between metrics and maintenance efforts. Lindvall also studied the correlation between software size and change-proneness. Arisholm et al. (2004) studied the dynamic coupling metrics' correlation with change-proneness and found a significant correlation. Koru and Tian (2005) investigated the correlation of the highest values of metrics with highly changed classes. The authors collected the change count from CVS as well as 51 metrics for Mozilla and 46 metrics for OpenOffice. They found a correlation between highly changed classes and properties such as large size, high coupling, low cohesion, or deep inheritance. Giger et al. (2012) have proposed to predict change-prone classes using a combination of OO metrics and social network analysis. The study used neural network models to predict change-proneness on either code metrics or social analysis metrics and found that a combination of both outperforms using either one. Lu et al. (2012) have studied the prediction of change-proneness on 102 Java systems using 62 OO metrics. The authors used statistical meta-analysis methods to predict change-proneness. In this study, changes between two software releases were measured from two consecutive releases. Change-proneness was measured by considering classes as change-prone if one change was detected from the previous version of the software, otherwise not change-prone. Size metrics, coupling, and cohesion exhibited discrimination between change-prone and not change-prone, while inheritance metrics were poor in discriminating between change-prone and not change-prone. Elish et al. (2015) conducted maintenance efforts and change-proneness prediction using advanced techniques such as ensemble methods. The authors performed classification on maintenance data. This empirical study compared individual prediction models with ensemble methods. The authors found that ensemble methods have better accuracy than individual models across datasets. Yan et al. (2017) proposed a new self-learning method for change-proneness prediction. The authors applied unsupervised learning methods, including clustering, labeling, metrics selection, and instance selection, to predict change-prone classes. The study was conducted on the CK metrics in addition to dynamic coupling metrics. Malhotra and Jangra (2017) have studied the prediction of change-proneness of classes using object-oriented metrics. The authors conducted 10 machine learning techniques and compared the performance of the resulting models with a statistical model. The results were similar in the two types of classifications. The study was conducted on two open-source systems built in Java. L. Kumar et al. (2017) studied a large set of software metrics as predictors of change-proneness. The models were built for eight machine learning techniques and the features were selected by five feature selection techniques. The results show that coupling metrics are better than inheritance, cohesion, and size metrics in predicting change-prone modules.

Table 1. Related works to change-proneness prediction

Study	Models	Results
Lindvall (1998)	Statistical analysis	Large classes are more change-prone
Arisholm et al. (2004)	Multiple linear regression	Positive correlation
Koru and Tian (2005)	Ranking and a clustering technique	Correlation between highly changed classes and properties such as large size, high coupling, low cohesion, or deep inheritance
Giger et al. (2012)	Neural network	A combination of both outperforms using either one
Lu et al. (2012)	Statistical meta-analysis	Size metrics, coupling, and cohesion
Elish et al. (2015)	Ensemble methods	Ensemble methods have better accuracy than individual models
Yan et al. (2017)	Clustering	The proposed CLAMI+ slightly improves the CLAMI and unsupervised methods
Malhotra and Jangra (2017)	10 ML and statistical models	ML models are better than statistical
L. Kumar et al. (2017)	8 ML techniques	Coupling metrics are better than inheritance, cohesion, and size metrics
Zhu et al. (2018)	Bagging and resampling	Bagging with resampling improves the prediction performance
Liu et al. (2018)	Unsupervised ML	Cross-project prediction
Catolino and Ferrucci (2019)	Ensemble methods and ML methods	Ensemble methods outperform
Abbas et al. (2020)	10 single ML models And their combination	Ensemble classifiers outperformed
Catolino et al. (2020)	Bad-smells measures	The performance of baseline change prediction models increased by an average of 10% in terms of f-measure.
Malhotra et al. (2021)	11 feature selection techniques and three ML models.	The feature selection techniques were effective in improving models and some were the best techniques.
Zhu et al. (2022)	CNN models	CNN models outperform baseline models
Bansal et al. (2022)	Proposed an algorithm	Better performance
Alsolai and Roper (2022)	4 ml	Ensemble feature selection and sampling techniques improve results
de Carvalho Silva et al. (2022)	Change history and Four ML algorithms	Random Forest showed the best, and smell-related information does not improve the models.

Zhu et al. (2018) have proposed to predict change-proneness using a combination of bagging and resampling methods. The change-proneness was defined using partitioning methods based on box plots. Liu et al. (2018) proposed and evaluated a selective cross-project prediction of fault-proneness on 14 open-source projects. The results were compared with two related change-proneness models. The model works in three phases: (1) estimates the unknown labels of classes using an unsupervised model; (2) searches for the best match distribution in the source project to train a classifier; and (3) labels are predicted by a classifier and are evaluated and measured. The results show the proposed model improves on the previous change-proneness models. Catolino and Ferrucci (2019) have studied change-proneness prediction using ensemble methods and traditional ML methods. They found ensemble methods outperform the traditional ML methods. Abbas et al. (2020) proposed predicting change-proneness using object-oriented metrics. They studied the change-proneness using machine learning on a large dataset of many commercial software systems. They also aim to identify which of the OO metrics are more necessary to predict change-prone classes. The authors proposed using various models (10 single) and their combination such as ensemble classifiers with voting, select-Best, and staking scheme. Ensemble classifiers outperformed the single models in predicting change-prone classes in software.

Zhu et al. (2022) have conducted a study on using deep learning to predict change-proneness using convolutional neural networks. The results show that the CNN models in combination with the resampling methods perform better than the baseline methods. Bansal et al. (2022) proposed a cross-projects change-proneness model. The focus was on identifying the most suitable projects using a proposed algorithm that provides the best prediction accuracy. Alsolai and Roper (2022) proposed change proneness models using many machine learning models (naive Bayes, support vector machines, k-nearest neighbors, and random forests) for seven datasets. They used different combinations of feature selection, sampling, and ensemble sampling techniques. The results found that the ensemble feature selection and sampling techniques have the best accuracy in predicting the fault-prone classes. Singh and Agrawal (2023) collected changelogs and change requests from three open-source software projects with the aim of analyzing the change-prone of classes. The research aims to identify dependencies of change-prone classes that may help to manage the consequences of changes. This type of research is on the applications of change-proneness models and can utilize change-proneness prediction models to further analyze and identify dependencies in change.

The necessity for new machine learning models in this kind of research is evident from earlier publications. The use of ensemble learning, and cross-project predictions, was a trend in most recent works on change-proneness. The use of modern trends in prediction models, such as deep learning and various forms of ensemble learning, is necessary. To include the most crucial features in models, it is also necessary to use feature selection and sampling of unbalanced data. From a different angle, none of these articles has taken software evolution into account; they have only looked at changes from the prior release. The question of whether changes differ significantly between successive software releases has not been examined by the authors. In addition, even if the product has not been released officially, the developers want to know when changes turn into major ones. This might be a research trend to figure out how to quantify change more accurately to enhance the prediction models for change and change-proneness. For this kind of job, it is also crucial to understand how to define the significance of change.

This research focuses on the trends in software evolution and growth in the long-term evolution of software. The study aims to understand how software evolution affects change-proneness and redefine change-proneness from the evolution of software. Knowing the number of classes affected by software metrics is important in knowing how the software structure changes. Six well-known metrics are studied to measure coupling, cohesion, inheritance, response set of a class, and complexity. This research investigates when these properties become significantly different. Activities that are required as results of change, such as regression testing and change impact analysis, are necessary after every change. However, in very large systems these activities are time-consuming and therefore, I need to know when it becomes appropriate to start such activities. Therefore, this research aims to study the evolution of software from this perspective and view.

RESEARCH METHODOLOGY

Many software metrics, such as change-proneness or maintenance effort, have been proposed as surrogates for software quality. Many studies have been conducted to investigate the relationship between metrics and change-proneness. On the other hand, more research into the relationship between metrics and evolution is needed to understand how software systems evolve and metrics grow significantly. The purpose of this study is to investigate the significance of software evolution by investigating the effect of software changes on the properties of software classes.

To answer research questions, the evolution of five large systems is being studied, and the differences between consecutive releases are being measured on a regular basis, i.e., every two weeks. The Wilcoxon signed-rank test is used to determine the importance of differences. The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test that is used to compare the significance of the difference between two populations using two matched samples (Conover, 1999). The Wilcoxon test is used when the differences are non-normally distributed. The metric data for each version are compared to the succeeding versions until a significant difference is discovered. The difference between the two versions is reported as a significant change. The maximum difference for all metrics is then used to select the snapshots at which change-proneness prediction models are built using five well-known classifiers.

DATA COLLECTION

Many previous works have reported on software evolution research (Chatzigeorgiou & Melas, 2012; Israeli & Feitelson, 2010). Few, however, have reported the evolution regularly as D'Ambros et al. (2010). The authors gathered biweekly data from five large open-source systems. The systems have been measured over 90 times, for a total of 180 weeks. The authors compiled a number of metrics for coupling, cohesion, inheritance, and complexity. The source code was obtained from the system repositories. Metrics were calculated using FAMIX models generated by the Moose tool.

Collecting bi-weekly snapshots of the systems for more than 90 snapshots is a time-consuming task, so the work is restricted to only five systems. These are large systems that are representative of other large systems.

- *Eclipse JDT Core:* Metrics were collected for 91 bi-weekly versions of the system. JDT Core is the Java infrastructure of the Java IDE. More information on JDT core is provided on the official website https://www.eclipse.org/jdt/core/
- *Eclipse PDE UI:* Metrics were collected for 97 bi-weekly versions of the system. The PDE UI provides tool sets to help in all development activities of Eclipse components. More information on PDE is provided on the official website https://www.eclipse.org/pde/pde-ui/
- *Equinox Framework:* Metrics were collected for 91 bi-weekly versions of the system. Equinox is mainly used for developing and delivering the OSGi framework implementation for all Eclipse products. More information on Equinox is provided on the official website http://www.eclipse.org/equinox/framework/
- *Lucene:* Metrics were collected for bi-weekly versions of the system. Lucene is a free and open-source information retrieval software library, originally written completely in Java. More information on Lucene is provided on the official website https://lucene.apache.org/

- *Mylyn:* metrics were collected for 98 bi-weekly versions of the system. Mylyn is an application lifecycle management framework for Eclipse. More information on Mylyn is provided on the official website http://www.eclipse.org/mylyn/

Software Metrics

In this section, the change-proneness redefinition is discussed, and the OO metrics are described. The change-proneness is defined in previous works as the median of differences between any consecutive releases of software. If the differences are larger than the median, then the instance is labeled as 1, otherwise as 0. This process is repeated for all instances in the second release. However, in this work, the statistical significance of the difference between consecutive releases or snapshots are added to the definition of change-proneness. This methodology works for either releases or snapshots of software. Then the median of the differences is used to mark an instance as change-prone if the changes are larger than the median of change.

The Chidamber and Kemerer (CK) suite assesses the internal quality of a software product (Chidamber & Kemerer, 1994). These metrics assess object-oriented software written in languages such as Java or C++. The six quality properties measured by the CK suite are coupling, cohesion, inheritance depth and breadth, class responsibility, and complexity. The CK metrics are defined as follows:

- *Coupling Between Objects (CBO):* The number of couplings between classes is counted by the CBO. CBO for each class is calculated by counting the other classes that are coupled to it.
- *Response for Class (RFC):* The RFC metric counts a class's responsibility set, which is represented by the number of local methods and called methods.
- *Weighted Methods per Class (WMC):* The WMC measures class complexity. The complexity of a class is determined by adding the complexity of its methods.
- *Depth of Inheritance Hierarchy (DIT):* The number of classes descended from the inheritance's root.
- *Number of Children (NOC):* The NOC metric counts the classes that directly inherit a class. The number of children in a class indicates the number of specializations and uses. As a result, understanding all specializations is critical for maintaining and testing the parent.
- *Lack of Method Cohesion (LCOM):* The LCOM metric assesses interconnection within a class. The interconnections track how data attributes are used in methods. LCOM is the difference between pairs of methods that share data attributes (Q) and those that do not (P). The LCOM is calculated as follows: (P > Q) LCOM? (P - Q): 0. The LCOM metric assesses class structure cohesion. Low cohesive classes have a wide range of functionalities, making them difficult to reuse and maintain.

Descriptive Statistics

Table 2 displays the descriptive statistics for the number of changed classes in the five systems. Column 2 displays the number of classes at the end of the measured evolution.

Table 2. The change-proneness distribution for all systems

System	#Classes	SLOC	Domain	Percentage of changed classes	Period	Versions
Eclipse JDT	997	>224k	Development	64%	1.1.2005 - 6.17.2008	91
Equinox Framework	324	>39k	Library	48%	1.1.2005 - 6.25.2008	91
Apache Lucene	691	>64k	Library	46%	1.1.2005 - 10.8.2008	99
Mylyn	1862	>156k	Development	47%	1.17.2005 - 3.17.2009	98
Eclipse PDE	1497	>146k	Development	73%	1.1.2005 - 9.11.2008	97

In Eclipse JDT, there were 64% changes among these classes. These statistics demonstrate that not all classes have evolved. Knowing when classes change significantly over time, on the other hand, is critical for regression testing and change impact analysis. The systems have various sizes as measured by the number of classes and SLOC. Three systems are software development applications of medium to large, JDT, Mylyn, and PDE, whereas two can be considered small, Equinox and Lucene, which are software libraries for development. All systems under study are from one domain.

MACHINE LEARNING MODELS AND PERFORMANCE EVALUATION

In this study, I conduct our experiment on five well-known classification techniques. In the following, a brief description of each technique is provided:

- *Logistic Regression:* The regression function LR was widely used to predict binary variables. The LR model is a regression model that works well with binary predictors (Hosmer & Lemeshow, 2000). The LR model is constructed from a logistic curve as a combination of all metrics to predict the change-prone class.
- *Naïve Bayes (NB):* NB is a simple classifier that was commonly used in software quality prediction and has been used as a classifier for defect prediction in many studies (Lessmann et al., 2008; Menzies et al., 2007). NB is intuitive and simple to build. A naive Bayes classifier is a supervised learning algorithm based on applying Bayes' theorem. Naïve Bayes considers the variables as conditionally independent given the predicted values.
- *Nearest neighbor (kNN):* Nearest neighbor classification is a type of *instance-based learning,* and it simply stores instances of the training data. kNN assigns the dominant label of the closest group of k objects in the training set. kNN uses the distance (similarity) metric to find the nearest neighbors and assigns the label that has the majority class (Aha et al., 1991). The 5NN was selected as a classifier that finds the distance with the nearest 5 instances and selects the class with the majority.
- *Support Vector Machine (SVM):* SVM uses hyperplanes (works even when the data are not linearly separable) to find the best function that discriminates between two classes (change-prone and not change-prone) by maximizing the margin between the two classes. SVM finds the maximum margin hyperplane that ensures generalizability (Burges, 1998). SVM models are effective in high-dimensional spaces.
- *Decision trees (CART):* CART is another classifier that builds decision trees using the Gini diversity index. CART predicts classes by learning decision rules inferred from the data set. The CART grows recursively by partitioning the training data set into subsets with similar values for the class. The CART algorithm grows the tree by conducting an exhaustive search of all attributes (i.e., metrics) and all possible splitting values, selecting the split that reduces impurity in each node (Ebert, 1996).

Measuring the performance of the prediction models is vital to compare the resulting models. There are many performance measures including accuracy, precision, recall, and F-measure. In comparison, the results of the F-measure are presented and discussed, which provides a score that incorporates both precision and recall measures into the individual score. The following is how precision and recall are calculated:

$$\text{Precision} = \text{True Positives} / (\text{True Positives} + \text{False Positives}) \qquad (1)$$

Precision is the ratio between the True Positives (instances correctly predicted as change-prone) and all instances that were predicted as change-prone. The precision measures how many of the found results are change-prone.

$$\text{Recall} = \text{True Positives} / (\text{True Positives} + \text{False Negatives}) \qquad (2)$$

The recall measures how a model correctly identifies existing change-prone instances. Thus, for all the instances that were changed, recall tells us how many I correctly identified as having changed.

The F-measure is the harmonic mean of both the precision and recall and summarizes both values.

$$\text{F-measure} = 2 \cdot \text{Precision} \cdot \text{Recall}/(\text{Precision} + \text{Recall}) \qquad (3)$$

RESULTS ANALYSIS

In the following two sections, first, how to find the significant difference between several software snapshots is provided. The goal is to determine when differences become statistically significant. In the second section, change-proneness prediction using a variety of machine-learning techniques are presented.

FINDING CHANGE SIGNIFICANCE

The research questions are answered in the following using both descriptive and statistical methods.

RQ1*: Which metrics are more affected by software evolution?*

For each release, I count the number of changed files in each metric. As a result, more than 89 values for each system are studied. Line charts and box plots are the best statistics for analyzing the results. Figure 1 depicts the evolution of the five systems as line charts. It can discover when significant changes occur in the system's evolution using line charts. For example, in Eclipse evolution at period 20, the DIT metric has increased significantly, indicating that 243 classes have changed their inheritance depth. The graph shows that metrics change all the time, but the number of affected classes does not always remain constant. This pattern demonstrates that systems evolve significantly at some points during the project's lifespan.

The line chart depicts several Equinox peaks where changes in many metrics are significant. This graph illustrates how metrics can change simultaneously, i.e., cyclic patterns. The Lucene line chart shows some peaks where all metrics change, indicating significant changes. The final line chart is for PDE and shows many fluctuations in the metrics' evolution. Changes in the evolution of software have a greater impact on the metrics at certain points in time (peaks in Figure 1) than at others.

Boxplots in Figure 2 depict the range of evolution's effect on the six metrics. DIT and NOC appear to be the least affected by evolution. The use of line and box plots together is essential for understanding the local points where evolution is important and determining which metrics are more affected by evolution. Among metrics, RFC and WMC are the most affected. Evolution has had the greatest impact on large classes (i.e., God classes).

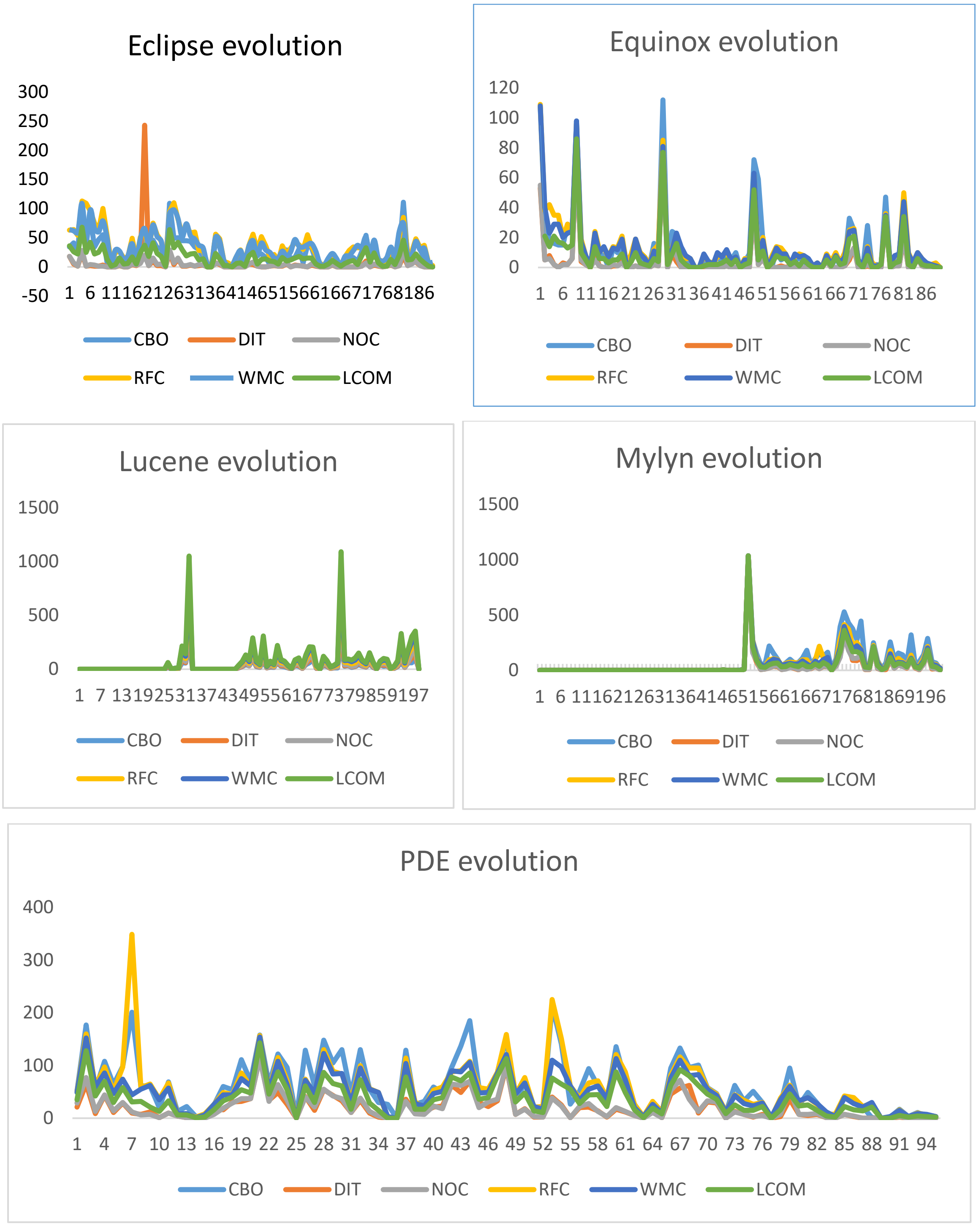

Figure 1. The line charts for the five systems

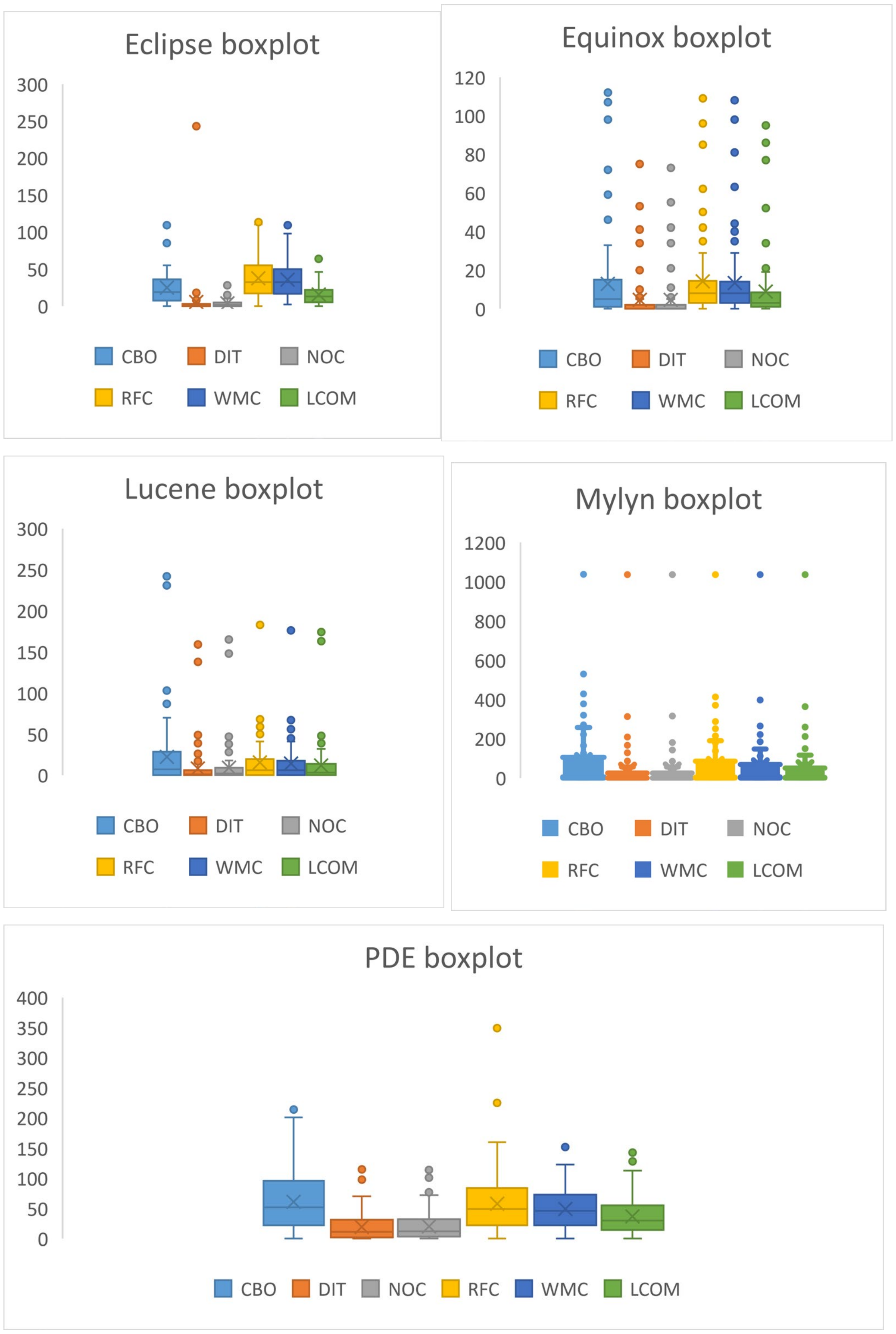

Figure 2. Boxplots for the five systems

***RQ2**: when does the software quality change significantly?*

I used the Wilcoxon signed rank test to determine the difference between the two snapshots in order to answer this question. The comparison with subsequent snapshots is repeated until a significant difference is found. The time it takes for differences to become statistically significant is calculated and reported. Table 3 displays the average number of weeks with significant differences. Based on the results shown in Table 3, the research questions RQ2.1 to RQ2.6 in numbers are answered. For RQ2.1, the CBO metric requires 3 to 4 weeks to achieve significant differences, which is considered fast to change. For RQ2.2, the LCOM metric needs more time to change than CBO but with a slight increase. Therefore, lack of cohesion is fast to change as well. For RQ2.3 and 2.3 for the inheritance metrics, the change is very slow, which means developers do not change the inheritance depth and breadth frequently. For example, it takes NOC 18 weeks to achieve significant differences in Equinox. RQ3.5 asked about how the responsibilities of a class have changed. The responsibility set changes frequently. RQ2.6 asks about how significant the changes in the complexity of classes. The complexity shows the most frequency of change as the responsibility of a class.

Table 3. The average number of bi-weeks to reach significant differences

	CBO	LCOM	DIT	NOC	RFC	WMC
Eclipse	2.02	2.23	7.83	6.77	1.39	1.34
Equinox	3.35	3.88	8.95	9.00	3.21	2.95
Lucene	2.56	3.11	3.34	3.03	2.86	2.60
Mylyn	1.26	1.30	1.98	1.80	1.30	1.17
PDE	1.46	1.71	2.27	2.36	1.41	1.43

Box plots, in Figure 3, depict which metrics achieve the fastest significant differences. Knowing which metrics are affected first provides software engineers with insights into how software evolves and how object-oriented design is maintained. It can be seen that inheritance hierarchies are slower to change and thus have the least impact among object-oriented metrics. Many previous works have shown that the inheritance metrics have lower prediction performance for fault prediction (e.g., Shatnawi & Mishra, 2021). DIT and NOC require the most weeks to achieve significant differences. These metrics are not as frequently updated as other metrics. The complexity and responsibilities of a class are the most frequently changed. Therefore, they are supposed to have more effect on software quality. These metrics measure the internal properties of a class. The coupling and cohesion measures change significantly every four weeks on average and these metrics also have fast change but less than the responsibilities and complexity of a class.

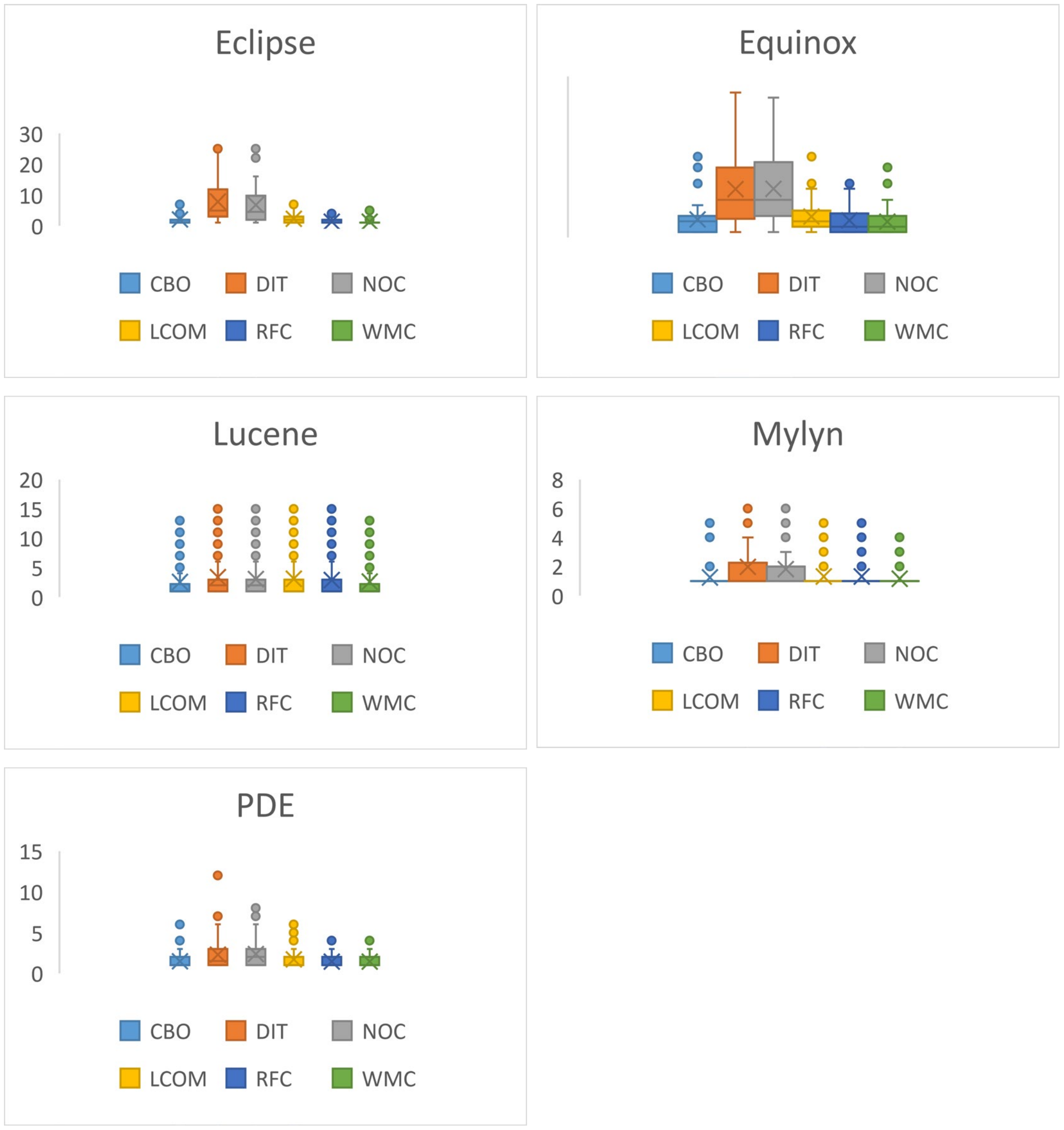

Figure 3. Boxplots for the significance of change measured biweekly

Change-Proneness Prediction

Change-proneness prediction is critical for engineers to predict which classes are more prone to changes, allowing for more effective and efficient resource allocation for future maintenance. The research questions specified the number of weeks required to achieve significant changes. I use 18 weeks to divide the data sets into training and testing datasets. As a result, I have ten snapshots to consider as significant software evolution. Changes from the previous snapshot are calculated for each snapshot, and a class is labeled as change-prone if it was changed more than the median of changes for all classes in a system. Table 4 displays the ten snapshots, as well as the percentage of change-prone classes identified in each. For example, at snapshot 9 (after 18 weeks), I discovered that a large proportion of classes in Eclipse (36%), PDE (50%), and Equinox (62%) were change-prone. Lucene and Mylyn were discovered unaltered until after snapshot 36. As a result, several prediction models were created at regular intervals for each system (18 weeks or 9 biweekly snapshots).

For each system, I have ten snapshots. To develop and test prediction models. Models were trained on each snapshot and tested for their ability to predict the change-prone classes in the following snapshot. As a result, I have nine models for each classifier and system. Tables 5, 6, 7, 8, and 9 show the results of change-propones predictions for the five systems, respectively. The results of the five classifiers are presented in each table. For all five classifier snapshots, the F-measure values for Eclipse, PDE, and Equinox are large as shown in Tables 5, 6, and 7, respectively. I notice that prediction performance is lower for early snapshots and improves for later snapshots. This is more noticeable in the Mylyn system as shown in Table 9. As a result, I can conclude that knowing when to make significant changes can lead to improved performance in change-proneness models. With high precision or recall, these models predict the change in the next snapshot.

Table 4. Statistics of change-prone classes among selected snapshots (biweekly)

System		R9	R18	R27	R36	R45	R54	R63	R72	R81	R90
Eclipse	#classes	874	882	921	935	944	950	952	954	963	997
	%change-prone	36%	22%	31%	34%	16%	19%	23%	12%	20%	21%
PDE	#classes	800	813	944	1066	1181	1292	1340	1420	1474	1494
	%change-prone	50%	26%	40%	39%	27%	30%	35%	25%	56%	13%
Equinox	#classes	186	194	194	225	224	271	271	290	302	324
	%change-prone	62%	54%	28%	47%	17%	36%	17%	23%	18%	24%
Lucene	#classes	344	344	344	511	511	572	641	694	563	615
	%change-prone	0%	0%	0%	47%	0%	18%	19%	20%	19%	19%
Mylyn	#classes	0	0	0	0	1077	1211	1309	1437	1702	1863
	%change-prone	0	0	0	0	100%	37%	49%	52%	36%	27%

Table 5. F-measure values for the five classifiers for Eclipse snapshots

Training	Testing	LR	NB	KNN	SVM	DT
9	18	89%	89%	80%	89%	80%
18	27	85%	84%	82%	83%	82%
27	36	83%	83%	81%	81%	81%
36	45	91%	91%	82%	92%	81%
45	54	91%	90%	87%	90%	88%
54	63	89%	88%	86%	88%	87%
63	72	94%	94%	88%	94%	88%
72	81	90%	90%	89%	89%	89%
81	90	90%	89%	86%	89%	86%

Table 6. F-measure values for the five classifiers for PDE snapshots

Training	Testing	LR	NB	KNN	SVM	DT
9	18	77%	84%	71%	77.0%	73.9%
18	27	76%	75%	71%	72.8%	71.8%
27	36	77%	76%	67%	71.4%	69.4%
36	45	79%	82%	69%	74.5%	71.4%
45	54	83%	82%	77%	79.1%	77.8%
54	63	81%	81%	71%	75.8%	73.5%
63	72	82%	84%	75%	79.0%	76.8%
72	81	62%	64%	60%	62.0%	61.0%
81	90	69%	86%	63%	73.0%	67.8%

Table 7. F-measure values for the five classifiers for Equinox snapshots

Training	Testing	LR	NB	KNN	SVM	Decision Trees
9	18	57%	69%	57%	22.0%	58%
18	27	78%	83%	65%	79.8%	65%
27	36	72%	73%	70%	72.2%	66%
36	45	82%	89%	71%	85.6%	71%
45	54	80%	80%	76%	79.4%	77%
54	63	91%	90%	76%	88.9%	77%
63	72	89%	88%	85%	87.9%	85%
72	81	91%	89%	85%	91.2%	86%
81	90	88%	88%	82%	87.0%	83%

Table 8. F-measure values for the five classifiers for Lucene snapshots

Training	Testing	LR	NB	KNN	SVM	DT
54	63	89.5%	87%	83%	90%	84%
63	72	89.3%	88%	83%	89%	83%
72	81	89.6%	88%	83%	90%	84%
81	90	89.3%	88%	84%	89%	85%

Table 9. F-measure values for the five classifiers for Mylyn snapshots

Training	Testing	L.R	NB	KNN	SVM	DT
54	63	71%	70%	66.1%	69%	66%
63	72	67%	66%	63.2%	63%	63%
72	81	67%	76%	60.6%	73%	62%
81	90	82%	83%	74.6%	85%	75%

Discussion of Results

The proposed work is different from previous literature in many folds:

i. The metrics are measured bi-weekly for the systems under investigation. In addition, a snapshot can be considered for different types of versioning including a regular candidate, or milestone release. Therefore, the dependent variable (change-proneness) is defined with the help of a statistical method to determine the significance of differences between biweekly snapshots. The change-proneness is not measured until there is a significance of the change.

ii. Metrics are validated individually using evolution and statistical tests to understand how strongly they are affected by evolution and therefore they have more correlation with change.

iii. The validation of the results is consistent with how the change-proneness is defined. The models are trained on a snapshot and tested on the next snapshot for better generalizability of the models.

In comparison with the previous works on software evolution, the focus was on the magnitude and direction of the growth of large systems only such as Linux systems (Chatzigeorgiou & Melas, 2012; Israeli & Feitelson, 2010; Mens & Demeyer, 2008). In this research, the evolution of software is measured, so the work aims to understand the evolution from a different perspective. The work investigates and finds when differences become statistically significant, and studies different OO properties such as coupling, cohesion, complexity, and inheritance instead of merely studying the size of software using LOC. The findings are also consistent with previous works. It advises using coupling, cohesion, and complexity more than inheritance metrics as they do not change frequently unless the software is measured for longer periods. For the aim of building a change prediction model, a split at 18 weeks is selected to split data into training and testing datasets. The results of the models were excellent when measured using F-measure scores. Therefore, the proposed research questions are answered.

In comparison with previous literature (Malhotra & Khanna, 2019), the results in this work use F-measure while most previous works report accuracy and AUC scores. The models have a very good score for most models. It is also observed that the models' performance improves for later snapshots as shown in the results.

THREATS TO VALIDITY

Construct validity is concerned with the validity of the data sets. The data sets are open-source and can be validated by other researchers. An open-source tool was used to measure the software metrics, which can be used for other purposes. The metrics are calculated every two weeks, which is sufficient for answering the research questions. However, how a metric tool defines and calculates metrics could be a concern as there might be slight to large differences among different tools. In addition, the definition of change-proneness is more generic and can be used for both a snapshot or a release (or candidate release or milestone).

Internal validity is the extent to which research questions are answered by data sets under investigation. Although investigating five systems, they are large and represent both open-source and commercial systems. These systems are widely used in both open-source and industrial settings.

External validity is concerned with the extent to which the findings and conclusions can be generalized to other types of systems. The systems are written in Java, and the results can be applied to other object-oriented applications. However, because these systems are built using the object-oriented methodology, the findings of this study may not be applicable to other methodologies, such as procedural and aspect-oriented. In addition, all systems are from one domain, i.e., development systems, which makes the results generalizable to these types of systems. However, these are complex systems, and usually, this applies to similar domains.

Conclusion

Software evolution is important because it persists after production. This evolution affects the change-proneness of classes and predicting which classes are going to change in the future is important in software project management. In addition, the evolution patterns in software reveal a lot about the system's quality. Understanding the evolution of software is critical to understanding the future of software production activities such as regression testing. Therefore, the evolution of five large open-source systems is studied in order to understand software evolution and how it affects changes in software. Therefore, change-proneness is redefined from studying software evolution. A change-proneness variable is proposed that is derived from changes on many snapshots for a long time. The change-proneness is considered only for the snapshots that become statistically different as tested using the Wilcoxon test. Then ML models are built for change-proneness prediction using object-oriented metrics and tested on the next snapshot. CK metrics were chosen as a well-known suit and a representative of software metrics that measure different aspects of software structure and thus represent different software qualities. To achieve these objectives, more than 89 measurements were taken at biweekly intervals on the systems under investigation. Therefore, the experiment was repeated many times in this interval. Each time differences were found, the change-proneness was trained and tested on the next different snapshot.

First, to understand the impact of evolution on software metrics, line charts, and boxplots were used. The findings of this study show that object-oriented metrics can be used to study software evolution. Peaks in line charts show metrics changing significantly over short time periods but less so over longer time periods. The metrics do not demonstrate the same degree of change. Other metrics evolve faster than inheritance metrics. The metrics do not have to change at the same time, and peaks for the same metrics may coincide, indicating large updates.

The systems evolved on a regular basis, but the impact of changes requires at least three weeks for most metrics and more for inheritance metrics. The findings of this study shed more light on the evolution of software structure. The results show that studying software growth solely through lines of code is insufficient and that other metrics must be considered when studying software evolution. Knowing when software properties like coupling, cohesion, size, and complexity change will drive effort estimation for software engineering activities and aid in the planning of future software releases, as well as how testing like regression testing and maintenance activities like refactoring can be directed. Therefore, this analysis answers RQ1. Using this methodology, it is known when changes become significant, which metrics become significantly changed faster, and the appropriate time for each metric change is known. Based on these results, change-prediction models were trained and tested appropriately at the suggested intervals and the results of the ML models were satisfying and good enough for most systems

For future work, I aim to study ensemble learning in change-proneness and compare different models of ensembles. As found by a systematic review study of ensemble techniques for software defect and change prediction (Khanna, 2022), there is a need for more studies on ensemble learning for both fault-proneness and change-proneness prediction. In a systematic mapping study on change impact analysis, Kretsou et al. (2021) found that change-proneness prediction and its effect on change impact analysis is not fully explored.

Availability of Data

The authors confirm that the data supporting the findings of this study are available within the work of D'Ambros et al. (2010) and its supplementary materials. The original datasets are publicly available at https://bug.inf.usi.ch/index.php

References

Abbas, R., Albalooshi, F. A., & Hammad, M. (2020, December). Software change proneness prediction using machine learning. *Proceedings of the* International Conference on Innovation and Intelligence for Informatics, Computing and Technologies*, Sakheer, Bahrain*. https://doi.org/10.1109/3ICT51146.2020.9311978

Agrawal, A., & Singh, R. K. (2020). Predicting co-change probability in software applications using historical metadata. *IET Software*, *14*(7), 739–747. https://doi.org/10.1049/iet-sen.2019.0368

Aha, D. W., Kibler, D., & Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning*, *6*, 37–66. https://doi.org/10.1007/BF00153759

Alsolai, H., & Roper, M. (2022). The impact of ensemble techniques on software maintenance change prediction: An empirical study. *Applied Sciences*, *12*(10), 5234. https://doi.org/10.3390/app12105234

Arisholm, E., Briand, L., & Føyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, *30*(8), 491–506. https://doi.org/10.1109/TSE.2004.41

Bansal, A., Madaan, V., Gaur, R., & Shakya, R. (2022, February). Cross-project change-proneness prediction with selected source project. *Proceedings of the International Conference on Innovative Trends in Information Technology, Kottayam, India*. https://doi.org/10.1109/ICITIIT54346.2022.9744186

Burges, C. J. C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery, 2*, 121–167. https://doi.org/10.1023/A:1009715923555

Carr, M., & Wagner, C. (2002). A study of reasoning processes in software maintenance management. *Information Technology and Management*, *3*, 181–203. https://doi.org/10.1023/A:1013125112217

Catolino, G., & Ferrucci, F. (2019). An extensive evaluation of ensemble techniques for software change prediction. *Journal of Software: Evolution and Process*, *31*(2), e2156. https://doi.org/10.1002/smr.2156

Catolino, G., Palomba, F., Fontana, F. A., De Lucia, A., Zaidman, A., & Ferrucci, F. (2020). Improving change prediction models with code smell-related information. *Empirical Software Engineering*, *25*, 49–95. https://doi.org/10.1007/s10664-019-09739-0

Chatzigeorgiou, A., & Melas, G. (2012, June). Trends in object-oriented software evolution: Investigating network properties. *Proceedings of the 34th International Conference on Software Engineering, Zurich, Switzerland*, 1309–1312. https://doi.org/10.1109/ICSE.2012.6227092

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895

Conover, W. J. (1999). *Practical nonparametric statistics* (3rd ed.). Wiley.

D'Ambros, M., Lanza, M., & Robbes, R. (2010, May). An extensive comparison of bug prediction approaches. *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, Cape Town, South Africa*, 31–41. https://doi.org/10.1109/MSR.2010.5463279

de Carvalho Silva, R., Farah, P., & Vergilio, S. R. (2022). Machine learning for change-prone class prediction: A history-based approach. *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, 289–298. https://doi.org/10.1145/3555228.3555249

Ebert, C. (1996). Classification techniques for metric-based software development. *Software Quality Journal*, *5*, 255–272. https://doi.org/10.1007/BF00209184

Elish, M., Aljamaan, H., & Ahmad, I. (2015). Three empirical studies on predicting software maintainability using ensemble methods. *Soft Computing*, *19*, 2511–2524. https://doi.org/10.1007/s00500-014-1576-2

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, *2*(3), 17–23. https://doi.org/10.1109/6294.846201

Giger, E., Pinzger, M., & Gall, H. C. (2012, June). Can we predict types of code changes? An empirical analysis. *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, Zurich, Switzerland,* 217–226. https://doi.org/10.1109/MSR.2012.6224284

Giray, G., Ebo Bennin, K., Köksal, Ö., Babur, Ö., & Tekinerdogan, B. (2023). On the use of deep learning in software defect prediction. *Journal of Systems and Software*, *195*, 111537. https://doi.org/10.1016/j.jss.2022.111537

Herraiz, I., Rodriguez, D., Robles, G., & Gonzalez-Barahona, J. M. (2013). The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Survey*, *46*(2), Article 28. https://doi.org/10.1145/2543581.2543595

Hosmer, D. W., & Lemeshow, S. (2000). *Applied logistic regression* (2nd ed.). Wiley. https://doi.org/10.1002/0471722146

Illes-Seifert, T., & Paech, B. (2010). Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open-source programs. *Information & Software Technology*, *52*(5), 539–558. https://doi.org/10.1016/j.infsof.2009.11.010

Israeli, A., & Feitelson, D. G. (2010). The Linux kernel as a case study in software evolution. *Journal of Systems and Software*, *83*(3), 485–501. https://doi.org/10.1016/j.jss.2009.09.042

Kaur, A., Kaur, K., & Kaur, H. (2016). Application of machine learning on process metrics for defect prediction in mobile application. In S. Satapathy, J. Mandal, S. Udgata, & V. Bhateja (Eds.), *Information systems design and intelligent applications* (pp. 81–98). Springer. https://doi.org/10.1007/978-81-322-2755-7_10

Kemerer, C. F., & Slaughter, S. (1999). An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, *25*(4), 493–509. https://doi.org/10.1109/32.799945

Khanna, M. (2022). A systematic review of ensemble techniques for software defect and change prediction. *e-Informatica Software Engineering Journal*, *16*(1), 220105. https://doi.org/10.37190/e-Inf220105

Koru, A. G., & Tian, J. (2005). Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Transactions on Software Engineering*, *31*(8), 625–642. https://doi.org/10.1109/TSE.2005.89

Kretsou, M., Arvanitou, E.-M., Ampatzoglou, A., Deligiannis, I., & Gerogiannis, V. C. (2021). Change impact analysis: A systematic mapping study. *Journal of Systems and Software*, *174*, 110892. https://doi.org/10.1016/j.jss.2020.110892

Kumar, L., Rath, S. K., & Sureka, A. (2017). Using source code metrics to predict change-prone web services: A case-study on eBay services. *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, Klagenfurt, Austria, 1-7. https://doi.org/10.1109/MALTESQUE.2017.7882009

Kumar, P., Singh, S. N., & Dawra, S. (2022). Software component reusability prediction using extra tree classifier and enhanced Harris hawks optimization algorithm. *International Journal of System Assurance Engineering and Management, 13*, 892–903. https://doi.org/10.1007/s13198-021-01359-6

Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, *34*(4), 485–496. https://doi.org/10.1109/TSE.2008.35

Lindvall, M. (1998). Are large C++ classes change-prone? An empirical investigation. *Software Practice and Experience*, *28*(15), 1551–1558. https://doi.org/10.1002/(SICI)1097-024X(19981225)28:15<1551::AID-SPE212>3.0.CO;2-0

Liu, C., Yang, D., Xia, X., Yan, M., & Zhang, X. (2018). Cross-project change-proneness prediction. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC*), Tokyo, Japan, 2018, pp. 64-73, https://doi.org/10.1109/COMPSAC.2018.00017

Lu, H., Zhou, Y., Xu, B., Leung, H., & Chen, L. (2012). The ability of object-oriented metrics to predict change-proneness: A meta-analysis. *Empirical Software Engineering*, *17*, 200–242. https://doi.org/10.1007/s10664-011-9170-z

Malhotra, R., & Jangra, R. (2017). Prediction and assessment of change prone classes using statistical and machine learning techniques. *Journal of Information Processing Systems*, *13*(4), 778–804. https://doi.org/10.3745/JIPS.04.0013

Malhotra, R., Kapoor, R., Aggarwal, D., & Garg, P. (2021, May). Comparative study of feature reduction techniques in software change prediction. *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories, Madrid, Spain,* 18–28. https://doi.org/10.1109/MSR52588.2021.00015

Malhotra, R., & Khanna, M. (2019). Software change prediction: A systematic review and future guidelines. *e-Informatica Software Engineering Journal*, *13*(1), 227–259. https://doi.org/10.5277/e-Inf190107

Mens, T., & Demeyer, S. (Eds.). (2008). *Software evolution.* Springer. https://doi.org/10.1007/978-3-540-76440-3

Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, *33*(1), 2–13. https://doi.org/10.1109/TSE.2007.256941

Rajlich, V. (2014). Software evolution and maintenance. *Proceedings of the Future of Software Engineering* (pp. 133–144). ACM. https://doi.org/10.1145/2593882.2593893

Shatnawi, R., & Mishra, A. (2021). An empirical study on software fault prediction using product and process metrics. *International Journal of Information Technologies and Systems Approach*, *14*(1), 62–78. https://doi.org/10.4018/IJITSA.2021010104

Singh, R. K., & Agrawal, A. (2023). Identification and analysis of change ripples in object-oriented software applications. *Sādhanā*, *48*, Article 95. https://doi.org/10.1007/s12046-023-02137-9

Témolé, F., & Atanasova, D. (2023, May). Role, importance and significance of software quality. *Proceedings of the 46th MIPRO ICT and Electronics Convention*, *Opatija, Croatia*, 1658–1663, https://doi.org/10.23919/MIPRO57284.2023.10159733

Wermelinger, M., Yu, Y., & Lozano, A. (2008, September). Design principles in architectural evolution: A case study. *Proceedings of the 24th IEEE International Conference on Software Maintenance*, *Beijing, China,* 396–405. https://doi.org/10.1109/ICSM.2008.4658088

Xie, H., Yang, J., Chang, C. K., & Liu, L. (2017). A statistical analysis approach to predict user's changing requirements for software service evolution. *Journal of Systems and Software*, *132*, 147–164. https://doi.org/10.1016/j.jss.2017.06.071

Xing, Z., & Stroulia, E. (2004, June). Understanding class evolution in object-oriented software. *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, *Bari, Italy,* 34–43. https://doi.org/10.1109/WPC.2004.1311045

Yan, M., Zhang, X., Liu, C., Xu, L., Yang, M., & Yang, D. (2017). Automated change-prone class prediction on unlabeled dataset using unsupervised method. *Information and Software Technology*, *92*, 1–16. https://doi.org/10.1016/j.infsof.2017.07.003

Yu, L., & Mishra, A. (2013). An empirical study of Lehman's law on software quality evolution. *International Journal of Software and Informatics*, *7*(3), 469–481.

Zhu, X., He, Y., Cheng, L., Jia, X., & Zhu, L. (2018). Software change-proneness prediction through combination of bagging and resampling methods. *Journal of Software: Evolution and Process*, *30*(12), e2111. https://doi.org/10.1002/smr.2111

Zhu, X., Li, N., & Wang, Y. (2022). Software change-proneness prediction based on deep learning. *Journal of Software: Evolution and Process*, *34*(4), e2434. https://doi.org/10.1002/smr.2434

AUTHOR

Raed Shatnawi received an MSc degree in software engineering and a Ph.D. degree in computer science from the University of Alabama in Huntsville. I am currently a full professor in the Department of Software Engineering at Jordan University of Science and Technology. I have published many papers in highly-ranked journals and conferences. I have reviewed papers for many reputed journals, and international conferences. My main interests are in software metrics, software refactoring, software maintenance, software security analysis, and open-source systems development. I was listed among the top 2% of researchers based on a study by Stanford University research.