


Article

# Software Refactoring Prediction Using SVM and Optimization Algorithms

Mohammed Akour <sup>1,\*</sup>, Mamdouh Alenezi <sup>1</sup>  and Hiba Alsghaier <sup>2</sup><sup>1</sup> Software Engineering Department, Prince Sultan University, Riyadh 11586, Saudi Arabia<sup>2</sup> Computer Science Department, University of Wisconsin-Milwaukee, Milwaukee, WI 53211, USA

\* Correspondence: makour@psu.edu.sa or mohammed.akour@yu.edu.jo

**Abstract:** Test suite code coverage is often used as an indicator for test suite capability in detecting faults. However, earlier studies that have explored the correlation between code coverage and test suite effectiveness have not addressed this correlation evolutionally. Moreover, some of these works have only addressed small sized systems, or systems from the same domain, which makes the result generalization process unclear for other domain systems. Software refactoring promotes a positive consequence in terms of software maintainability and understandability. It aims to enhance the software quality by modifying the internal structure of systems without affecting their external behavior. However, identifying the refactoring needs and which level should be executed is still a big challenge to software developers. In this paper, the authors explore the effectiveness of employing a support vector machine along with two optimization algorithms to predict software refactoring at the class level. In particular, the SVM was trained in genetic and whale algorithms. A well-known dataset belonging to open-source software systems (i.e., ANTLR4, JUnit, MapDB, and McMMO) was used in this study. All experiments achieved a promising accuracy rate range of between 84% for the SVM–JUnit system and 93% for McMMO – GA + Whale + SVM. It was clear that added value was gained from merging the SVM with two optimization algorithms. All experiments achieved a promising F-measure range between the SVM–Antlr4 system’s result of 86% and that of the McMMO – GA + Whale + SVM system at 96%. Moreover, the results of the proposed approach were compared with the results from four well known ML algorithms (NB–Naïve, IBK–Instance, RT–Random Tree, and RF–Random Forest). The results from the proposed approach outperformed the prediction performances of the studied MLs.

**Keywords:** software engineering; optimization algorithms; SVM; software refactoring

**Citation:** Akour, M.; Alenezi, M.; Alsghaier, H. Software Refactoring Prediction Using SVM and Optimization Algorithms. *Processes* **2022**, *10*, 1611. <https://doi.org/10.3390/pr10081611>

Academic Editors: Pablo Chamoso, Guillermo Hernández and Paulo Novais

Received: 18 July 2022

Accepted: 9 August 2022

Published: 15 August 2022

**Publisher’s Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

In any business sector, the quality of a particular product or a service matters, and this quality is often dependent on the process that is followed to build that product or service [1]. Today’s world massively depends on software technology, and high quality in these software systems has been greatly in demand for the past few decades. The main expectancy of high-quality software is in its reliability and ecosystem. This is achieved by reducing the bugs or failures in the software algorithms. These bugs tend to slow down the software’s response and user experience, which can harm its performance. These system errors cause faults, and subsequently, the faults cause system failures [1–5]. Altering a software system in a way that does not affect its external response, but improves its internal structure, is known as refactoring [6]. It also improves the external response by adding qualities such as improved user experience and interfaces.

During a software application’s life cycle, the software is continuously changed to adapt to new features or to modify existing ones to cope with new requirements. In order to continue to satisfy stakeholders’ needs, their requirements oblige the developer to reflect their intended needs into the software. It is known that software maintenance is the most

expensive phase in the software development lifecycle [7]. These maintenance activities usually happen incrementally, and can be carried out to add or modify functionality, or to restructure the design for a better user experience. If the system does not go through several design correction activities, its quality will degrade [8].

Once software developers receive new demands or requests, they modify the software to accommodate these requirements (software refactoring) [9]. Software refactoring modifies the internal structure of the software without altering its external functionality [8,9]. Moreover, software refactoring is employed to enhance the understandability, reduce the complexity, and increase the maintainability of the targeted software [10,11].

Software Refactoring might change the software at three levels, from the lower to higher levels of variables, functions, and classes. These changes introduce a big technical challenge to the software developers, especially when they need to identify both the level and all the code pieces that need refactoring. The primary aim of refactoring is to make the code more maintainable without changing its semantics [12]. Software refactoring is a highly challenging task, particularly in identifying which parts of the software have to be refactored and which methods are to be used. These challenges arise due to the significant functionality limitations that software repositories contain, and the type of data used in them [1]. Hence, much research has raised the need to build refactoring prediction/recommendation systems to assist in evolution tasks [6,11,13–16].

Although the refactoring task is generally dependent on the software developers' skills and insights, this process may still be supported by refactoring prediction/recommendation systems. These prediction systems facilitate the process of detecting the classes or methods that need refactoring.

Several techniques to predict refactoring have been studied, for example, code smells [17], pattern mining [18], invariant mining [19], and search-based techniques [20]. Machine learning algorithms reveal encouraging results when utilized in various fields of software engineering, such as defect prediction [2,3,21–23].

To the best of our knowledge, the work presented in this article introduces a new research contribution. The research work in this paper presents a class-level refactoring prediction from four open-source Java-based systems, i.e., ANTLR4, JUnit, MapDB, and McMMO, using a support vector machine (SVM) and two optimization algorithms: genetic algorithms (GA) and whale algorithms (WA). This paper uses the studied algorithms to predict the refactoring needs at the class level when stand-alone and integrated algorithms are applied. The main problem that software practitioners encounter is recognizing which code segment has to be refactored. Therefore, this paper focuses on the use of SVM and optimization methods in this regard. By repeating the experiments until we reach the best iteration, we can develop an understanding of which technique's response is better, leading to optimized results in terms of software quality. Thus, by conducting these experiments, suggestions and conclusions can be made regarding the aforementioned refactoring methods and algorithms.

## 2. Related Work

In order to make predictions about the defects in particular software, researchers and developers also apply a machine learning approach to a software system in real-time. Some well-known examples of these machine learning approaches involve tele-control/telepresence, robotics, and mission planning systems. Many studies have been conducted in the field of software fault prediction, and the methods used by researchers differ between optimization, machine learning, and classification techniques [3]. There are several procedures employed to examine the defects present in software, but until now there has been no report of a technique that can display highly accurate results.

As discussed, various refactoring implication types exist. The main process of refactoring involves the modification of classes, methods, and variables. Upon doing that, developers must also address an important aspect—identifying all the code elements or code segments in the large complex system of the software that require refactoring.

In this respect, support vector machines (SVMs) have high popularity among software developers and testers. An SVM classifies data into predefined classes by computing a hyper-plane in a high-dimensional space [24–26]. In other words, it is a machine learning technique that can be used for classification. The advantage of using this method for feature selection is that it tends to reduce the computation time, and it also improves the prediction performance. Since it improves prediction accuracy and helps to enable the observation of different values and crucial factors for evaluating performance, many researchers use SVM for feature selection in their work.

Refactoring has been studied extensively within the literature. Fowler initiated the effort by coming up with the first catalog of 72 refactoring types, with an accompanying guide [9]. Simon et al. [27] proposed an approach to generate visualizations that help developers to identify bad smells.

Several different studies that examine the prediction of faults in software using object-oriented metrics have been conducted. The results of these studies show that object-oriented metrics are able to produce significantly enhanced outcomes compared to static code metrics. This is because object-oriented metrics represent different structural characteristics, such as coupling, cohesion, inheritance, encapsulation, complexity, and size [3,11,17,27].

An early survey [8] was conducted to shed light on refactoring which discussed refactoring activities, techniques, and tools. The authors discussed their beliefs about how refactoring can improve software quality in the long run. Most existing research studies are based on rule-based, machine learning, or search-based approaches. A systematic literature review (SLR) in [28] discusses how researchers are increasingly becoming interested in automatic refactoring techniques. Their results suggest that source code approaches are far more studied than model-based ones. The results also show that search-based approaches are more popular, and that recently more machine learning approaches have been explored by researchers to help experts to discover refactoring needs.

Mariani and Vergilio [29] conducted an SLR of search-based refactoring approaches. They observed that evolutionary algorithms, specifically genetic algorithms, were the most used algorithms. Mohan and Greer [30] investigated search-based refactoring in more depth, covering tools, metrics, and evolution, since their focus was software maintenance. They also found that the evolutionary algorithms were the most used.

Moreover, Shepperd and Kadoda [31] used simulation methods to differentiate between software predictions with the help of stepwise regression rule induction (RI), case-based reasoning (CBR), and artificial neural networks (ANN). They compared these prediction models to the results in actual software in terms of accuracy, explanatory value, and configurability, and they found that CBR and RI gave them an advantage over ANN, while CBR was favored by all.

Azeem et al. [32] conducted a systematic literature review to summarize the research on machine learning (ML) algorithms for code smelling predictions. Their review included 15 research studies that involved code smell and prediction models. According to the results of their study, decision trees and SVM are the most widely used ML algorithms for code smell detection, and furthermore, JRip and Random are the most effective algorithms in terms of performance.

In addition to this, Liu et al. [33] describe a tool that uses conceptual relationship, implementation similarity, structural correspondence, and inheritance hierarchies to identify potential refactoring opportunities in the source code of open-source software systems.

Liu et al. [33] also showed that machine learning models that could predict a high level of defect classes could be built using static measures and defect data, which was collected at a high-class level.

Tsantalis and Chatzigeorgiou [34] reported a way to recognize refactoring suggestions with the help of polymorphism. Their main focus was on the detection and elimination of state-checking problems in programs that implement Java and deploy as eclipse add-ons or plug-ins. In 2007, Ng and Levitin proposed correcting faults, in addition to making predictions of faulty parts in software. In order to achieve this, they applied a genetic

algorithm and a number of neural networks iteratively. The genetic algorithm in this project was used to increase the performance of the prediction model.

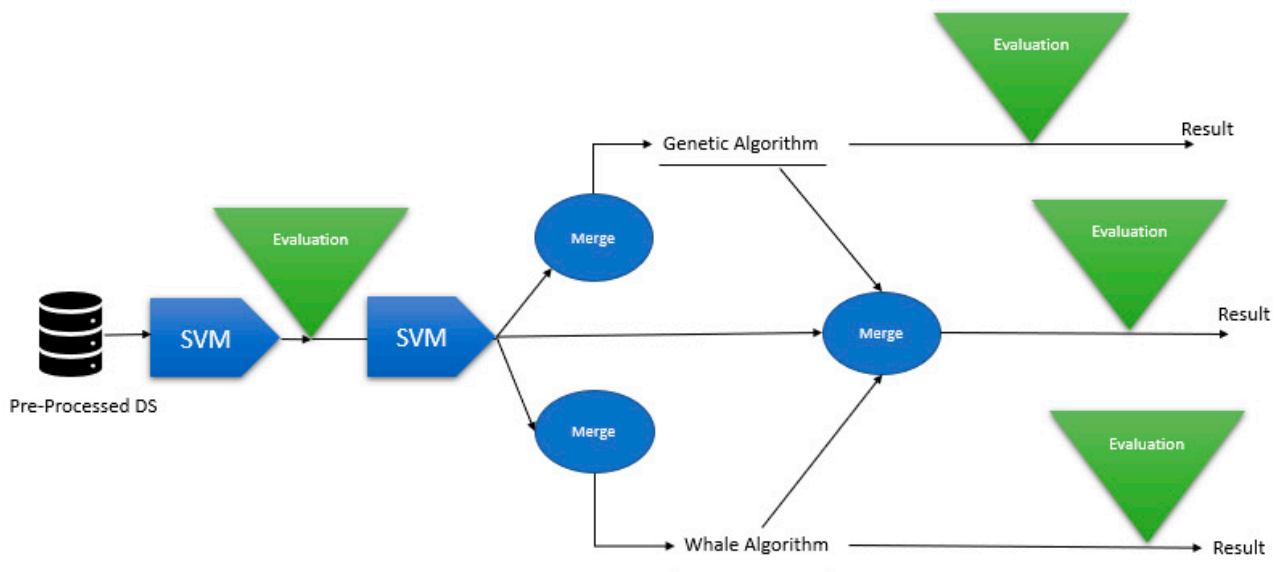
Erturk and Sezer [5] analyzed the evolution of an object-oriented source code at a class level. The refactoring events, which depended on a vector space model, were the main focus here. A list of class refactoring operations was created by the application of this proposed approach to an open-source domain.

Another study by Caldeira et al. [1] investigated the effects of aspects such as dataset size, metrics sets. These aspects had not been researched prior to this study. Random forest algorithms and artificial immune systems were used as machine learning methods, and a dataset was collected from the PROMISE repository. The algorithm selected was determined to be much more important than the metrics selected, as per this study [1].

### 3. Methodology

In this section, the authors present the technique developed for predicting software refactoring using a support vector machine classifier and two optimization algorithms. The developed approach is composed of four main phases. In the first phase, a pre-processing procedure is conducted on collected datasets. In the second phase, the GA, WA, and SVM classifiers are applied to the processed datasets to predict the refactoring opportunities. In the third phase, the results are evaluated by the Wilcoxon signed-rank test [35]. In the last phase, the results are compared, to determine the best overall approach.

Figure 1 depicts the main phases of the proposed technique.



**Figure 1.** Research Methodology.

#### 3.1. Datasets and Pre-Processing

In this work, a well-known dataset was used. The dataset includes empirical refactoring occurrences of four open-source software systems (JUnit, McMMO, MapDB, and ANTLR4) [36]. The dataset is available at the PROMISE Repository, making the experiment easily reproducible. The studied attributes of the datasets are shown in Table 1.

**Table 1.** Datasets Attributes.

Dataset	No. of Attributes	Instances	No. of Refactoring	Percentage (%)
Antlr4	134	436	23	5.2
Junit	134	657	9	1.3
MapDB	134	439	4	0.9
McMMO	134	301	3	0.99

All unnecessary attributes, such as the Long-Name, Parent, path, and Component were deleted during data pre-processing. Moreover, the class labels were replaced with 0 and 1, where false became 0 and true became 1.

### 3.2. Classifiers and Optimization Algorithms

An optimization algorithm is a process that is executed iteratively, by making comparisons of different solutions until an optimum result is found. A classifier is an algorithm that prints the input data to a specific category. The feature of a classifier model is that it can individually measure the properties of the software under inspection [37].

A well-known classifier and two optimization algorithms were used in this study. Moreover, several experiments were performed to discover the best integration of these algorithms in terms of refactoring prediction accuracy.

#### 3.2.1. GA with SVM

Genetic algorithm is either a heuristic optimization algorithm, or it is one of the search-based techniques. GA is commonly used in optimization, classification, and regression problems [38]. It provides a method of solving both constrained and unconstrained optimization problems based on a natural selection process. Genetic algorithms are one of the optimization algorithm types that are widely used in software fault prediction. In this stage, integration of the genetic algorithm with an SVM classifier was applied to the four datasets, and this iteration was repeated 51 times.

#### 3.2.2. Whale Algorithm with SVM

Whale algorithm's structure is based on the way of life of whales. It employs the solution's population to discover the optimal solution. The main idea behind this algorithm is different from the others, as it employs two opposite solutions. These two solutions are the best and the worst solutions, conceived in order to ascertain the optimal situation [39]. The whale algorithm is a new optimization algorithm, that is also used in our work. In this stage, we integrated the whale algorithm with an SVM classifier and applied it to the four datasets. This iteration was also repeated 51 times.

#### 3.2.3. GA and Whale Algorithms with SVM

In this stage, the three algorithms were integrated: we first applied the GA to the dataset with 17 iterations, then we applied the whale algorithm with 17 iterations, and finally the SVM algorithm was applied for another 17 iterations; resulting in a total of 51 iterations.

## 4. Results and Discussion

The proposed approach was empirically assessed using the four studied systems. The experiments were conducted using MATLAB. The genetics algorithm was merged with the support vector machine classifier and applied to the four datasets, where the iteration setting was fixed at 51. The same experiment was repeated for the other five approaches. Tables 2–4 summarize the prediction performance results for the dataset in terms of accuracy, STD, and F-measures, respectively. Comparisons of the four developed

approaches were conducted. In this study, the effectiveness of merging the optimization algorithms and machine learning (SVM) classifiers was evaluated in terms of refactoring prediction performance. Three optimization algorithms and four prediction data sets were studied in this work. To evaluate the developed approach, we used the Wilcoxon signed-rank test to calculate the  $p$ -value and to check for any significant differences.

**Table 2.** Prediction Results—Accuracy.

Dataset	Accuracy			
	SVM	GA + SVM	Whale + SVM	GA + Whale + SVM
Antlr4	0.881	0.904	0.902	0.905
Junit	0.845	0.851	0.845	0.846
MapDB	0.903	0.918	0.923	0.918
McMMO	0.900	0.937	0.934	0.937

**Table 3.** Prediction Results—STD.

Dataset	STD		
	GA + SVM	Whale + SVM	GA + Whale + SVM
Antlr4	5.236	6.039	6.668
Junit	1.785	1.814	1.990
MapDB	3.811	3.725	4.275
McMMO	2.846	1.465	1.275

**Table 4.** Prediction Results—F-measure.

Dataset	F-Measure			
	SVM	GA + SVM	Whale + SVM	GA + Whale + SVM
Antlr4	0.861	0.949	0.948	0.949
Junit	0.916	0.919	0.915	0.916
MapDB	0.940	0.958	0.960	0.957
McMMO	0.964	0.967	0.966	0.967

In this work, the prediction effectiveness is mainly measured through its accuracy. After conducting several experiments, there were differences between the addressed approaches. As shown in Table 2, The experiments achieved a high accuracy rate range of between 0.845 for the SVM-Junit system and 0.937 for McMMO – GA + Whale + SVM. It is clear that added value was gained by merging the SVM with two optimization algorithms.

Table 3 summarizes the comparison of the STD results obtained from all proposed approaches. The lowest ST was achieved by the GA + Whale + SVM – McMMO system, and was 1.2755. A low STD means that the data are close to the expected value, while the highest STD means that the data are the most widely spread from the expected results. The highest STD was achieved by the GA + Whale + SVM – Antlr4 system and was 6.668. Despite this result, the integration of the SVM with the optimization algorithms improved the performance of the refactoring prediction.

Table 4 shows the comparison of the experiments in terms of the F-measure. The experiments achieved a high F-measure range of between 0.861 for the SVM-Antlr4 system and 0.967 for McMMO – GA + Whale + SVM. It is clear that added value was gained from merging the SVM with two optimization algorithms.

Although the empirical refactoring occurrences of the four open-source software system (JUnit, McMMO, MapDB, and ANTLR4) data sets are widely used for prediction purposes, the high rates of accuracy with only slightly tangible differences between the studied algorithms leads the authors to believe that bigger datasets from different domains should be used for generalizing the findings. Such good prediction rates provide promising results, but further investigation using other datasets will be performed in the future.

Four more experimental comparisons were conducted using four well-known and widely used ML algorithms, i.e., NB-Naïve, IBK-Instance, Random Tree, and Random Forest. Tables 5–7 show the prediction performance in terms of the accuracy, F-measure, and STD of the studied classifiers.

**Table 5.** ML Prediction Results—Accuracy.

Dataset	Accuracy			
	NB-Naive	IBK-Instance	RT-Random Tree	RF-Random Forest
<b>Antlr4</b>	0.859	0.825	0.863	0.879
<b>Junit</b>	0.857	0.875	0.876	0.899
<b>MapDB</b>	0.863	0.867	0.875	0.882
<b>McMMO</b>	0.891	0.921	0.916	0.929

**Table 6.** ML Prediction Results—F-measure.

Dataset	F-Measure			
	NB-Naive	IBK-Instance	RT-Random Tree	RF-Random Forest
<b>Antlr4</b>	0.774	0.793	0.794	0.803
<b>Junit</b>	0.875	0.869	0.860	0.859
<b>MapDB</b>	0.859	0.841	0.805	0.857
<b>McMMO</b>	0.853	0.862	0.869	0.876

**Table 7.** ML Prediction Results—STD.

Dataset	STD			
	NB-Naive	IBK-Instance	RT-Random Tree	RF-Random Forest
<b>Antlr4</b>	6.777	3.378	3.191	5.179
<b>Junit</b>	4.189	3.645	3.863	3.499
<b>MapDB</b>	5.283	5.326	3.779	4.096
<b>McMMO</b>	2.875	2.979	4.746	2.089

As shown in Table 5, the lowest prediction accuracy achieved was 0.825 for the IBK-Instance-Junit system, and the highest prediction accuracy was 0.929 for McMMO-RF. It is clear that the lowest accuracy achieved by the proposed approach (0.845) was higher than the lowest accuracy achieved by the studied ML. Moreover, the highest accuracy achieved by the proposed approach (0.937) was higher than the highest accuracy achieved by the studied ML.

Table 6 shows that the random forest method achieved the best results in comparison to the other classifiers for all the datasets in terms of the F-measure. Still, the proposed approach achieved better results in terms of the F-measure. The lowest F-measure was 0.774, while the proposed approach's lowest F-measure was 0.861. On the other hand, the highest achieved F-measure was 0.876, while the proposed approach's highest F-measure was 0.967.

Table 7 shows that the random forest approach achieved the best results in comparison to the other classifiers for all the datasets in terms of STD, while the highest STD was achieved by NB-Naïve and was 6.777.

#### *Threat to Validity*

In this study, two optimization algorithms were utilized by incorporating them into the SVM to improve the proposed fault prediction approach. The summary of the threat to validity is highlighted in regards to the studied software systems and their datasets. In order to reduce the threat in this regard, the authors used a well-known dataset. The dataset includes empirical refactoring occurrences of four open-source software systems (JUnit, McMMO, MapDB, and ANTLR4) [36].

As external threat to validity, the authors have not addressed using the most complex open software systems for evaluating the proposed approach. Moreover, the studied and implemented algorithms are insufficient, but the experiment gives a promising result in terms of refactoring predictions. Therefore, the authors intend to use other optimization algorithms and to incorporate them with the SVM or other classification algorithms in future work.

## 5. Conclusions and Future Work

In this paper, the authors address the refactoring prediction at the class level by employing SVM with two optimization algorithms. Genetic and whale algorithms were used in this work, and the performance was evaluated using four open-source software product datasets. To the best of our knowledge, machine learning algorithms are most effective in predicting software refactoring, and developers can make faster and more educated decisions regarding what needs to be refactored. However, the optimization algorithms employed in this study were used for the first time for refactoring predictions at the class level. Several experiments were conducted, and promising performance results were observed. All experiments achieved a promising accuracy rate range between 84% for the SVM-Junit system and 93% for McMMO – GA + Whale + SVM. Merging the SVM with the two optimization algorithms played an important role in enhancing the accuracy of the F-measure. Moreover, four well-known ML algorithms were also used, and the prediction results were compared; the proposed approach achieved better performance in terms of accuracy and F-measure.

In future work, authors will attempt to predict the refactoring type at the class or method level by using the studied algorithms on another dataset. This will give us further information about the accuracy of these refactoring prediction systems, and may also clarify which among these four systems responds the best.

**Author Contributions:** Conceptualization, M.A. (Mohammed Akour) and M.A. (Mamdouh Alenezi); methodology, M.A. (Mohammed Akour) and H.A.; formal analysis, M.A. (Mohammed Akour) and M.A. (Mamdouh Alenezi); investigation, H.A.; writing—original draft preparation, M.A. (Mohammed Akour); writing—review and editing, M.A. (Mohammed Akour), M.A. (Mamdouh Alenezi) and H.A. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Publicly available datasets were analyzed in this study. This data can be found here: <https://malenezi.github.io/malenezi/data/Internal-Quality-Evolution-Java> (accessed on 1 April 2022).

**Acknowledgments:** The authors would like to acknowledge the support of Prince Sultan University for paying the Article Processing Charges (APC) of this publication.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Caldeira, J.; Brito e Abreu, F.; Cardoso, J.; dos Reis, J. Unveiling process insights from refactoring practices. *Comput. Stand. Interfaces* **2022**, *81*, 103587. [[CrossRef](#)]
2. Al Qasem, O.; Akour, M.; Alenezi, M. The influence of deep learning algorithms factors in software fault prediction. *IEEE Access* **2020**, *8*, 63945–63960. [[CrossRef](#)]
3. Alsgaier, H.; Akour, M. Software fault prediction using particle swarm algorithm with genetic algorithm and support vector machine classifier. *Softw. Pract. Exp.* **2020**, *50*, 407–427. [[CrossRef](#)]
4. Batool, I.; Khan, T.A. Software fault prediction using data mining, machine learning and deep learning techniques: A systematic literature review. *Comput. Electr. Eng.* **2022**, *100*, 107886. [[CrossRef](#)]
5. Erturk, E.; Sezer, E. A comparison of some soft computing methods for software fault prediction. *Expert Syst. Appl.* **2015**, *42*, 1872–1879. [[CrossRef](#)]



6. Aniche, M.; Maziero, E.; Durelli, R.; Durellim, V. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Trans. Softw. Eng. Early Access* **2020**, *48*, 1432–1450. [[CrossRef](#)]
7. Ghannem, A.; Boussaidi, G.E.; Kessentini, M. Model refactoring using interactive genetic algorithm. In *International Symposium on Search Based Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 96–110.
8. Mens, T.; Tourwé, T. A survey of Software Refactoring. *IEEE Trans. Softw. Eng.* **2004**, *30*, 126–139. [[CrossRef](#)]
9. Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. *Refactoring: Improving the Design of Existing Code*, 1st ed.; Addison-Wesley Professional: Berkeley, CA, USA, 1999.
10. Kumar, L.; Satapathy, S.; Krishna, A. Application Of Smote And Lssvm With Various Kernels For Predicting Refactoring At Method Level. In *International Conference on Neural Information Processing*; Springer: Cham, Switzerland, 2018; pp. 150–161.
11. Nyamawe, S.; Liu, H.; Niu, Z.; Wang, W.; Niu, N. Recommending refactoring solutions based on traceability and code metrics. *IEEE Access* **2018**, *6*, 49460–49475. [[CrossRef](#)]
12. Kumar, L.; Sureka, A. Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC)*, Nanjing, China, 4–8 December 2017. [[CrossRef](#)]
13. D’Ambros, M.; Lanza, M.; Robbes, R. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empir. Softw. Eng.* **2012**, *17*, 531–577. [[CrossRef](#)]
14. Singh, P.D.; Chug, A. Software defect prediction analysis using machine learning algorithms. In *Proceedings of the 7th International Conference on Cloud Computing, Data Science Engineering—Confluence*, Noida, India, 12–13 January 2017; pp. 775–781.
15. Silva, D.; Tsantalis, N.; Valente, M.T. Why we refactor? Confessions of GitHub contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, Seattle, WA, USA, 13–18 November 2016; pp. 858–870.
16. Alenezi, M.; Akour, M.; Al Qasem, O. Harnessing deep learning algorithms to predict software refactoring. *Telkomnika* **2020**, *18*, 2977–2982. [[CrossRef](#)]
17. Marinescu, R. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance Proceedings*, Chicago, IL, USA, 11–14 September 2004; pp. 350–359.
18. Bavota, G.; Panichella, S.; Tsantalis, N.; Penta, M.D.; Oliveto, R.; Canfora, G. Recommending refactoring based on team co-maintenance patterns. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, New York, NY, USA, 15–19 September 2014; pp. 337–342.
19. Kataoka, Y.; Imai, T.; Andou, H.; Fukaya, T. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance Proceedings*, Montreal, QC, Canada, 3–6 October 2002; pp. 576–585.
20. O’Keeffe, M.; Cinn’eide, M.O. Search-based refactoring for software maintenance. *J. Syst. Softw.* **2008**, *81*, 502–516. [[CrossRef](#)]
21. Akour, M.; Alsmadi, I.; Alazzam, I. Software fault proneness prediction: A comparative study between bagging, boosting, and stacking ensemble and base learner methods. *Int. J. Data Anal. Tech. Strateg.* **2017**, *9*, 1–16.
22. Al Qasem, O.; Akour, M. Software fault prediction using deep learning algorithms. *Int. J. Open Source Softw. Processes (IJOSSP)* **2019**, *10*, 1–19. [[CrossRef](#)]
23. Akour, M.; Melhem, W. Software defect prediction using genetic programming and neural networks. *Int. J. Open Source Softw. Processes (IJOSSP)* **2017**, *8*, 32–51. [[CrossRef](#)]
24. Adugna, T.; Xu, W.; Fan, J. Comparison of Random Forest and Support Vector Machine Classifiers for Regional Land Cover Mapping Using Coarse Resolution FY-3C Images. *Remote Sens.* **2022**, *14*, 574. [[CrossRef](#)]
25. Cervantes, J.; Garcia-Lamont, F.; Rodríguez-Mazahua, L.; Lopez, A. A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing* **2020**, *408*, 189–215. [[CrossRef](#)]
26. Ahmed, H.; Yu, Y.; Wang, Q.; Darwish, M.; Nandi, A. Intelligent Fault Diagnosis Framework for Modular Multilevel Converters in HVDC Transmission. *Sensors* **2020**, *22*, 362. [[CrossRef](#)] [[PubMed](#)]
27. Simon, F.; Steinbruckner, F.; Lewerentz, C. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal, 14–16 March 2001; pp. 30–38.
28. Baqais, A.; Alshayeb, M. Automatic software refactoring: A systematic literature review. *Softw. Qual. J.* **2020**, *28*, 459–502. [[CrossRef](#)]
29. Mariani, T.; Vergilio, S.R. A systematic review on search-based refactoring. *Inf. Softw. Technol.* **2017**, *83*, 14–34. [[CrossRef](#)]
30. Mohan, M.; Greer, D. A survey of search-based refactoring for software maintenance. *J. Softw. Eng. Res. Dev.* **2018**, *6*, 3–55. [[CrossRef](#)]
31. Shepperd, M.; Kadoda, G. Comparing software prediction techniques using simulation. *IEEE Trans. Softw. Eng.* **2022**, *27*, 1014–1022. [[CrossRef](#)]
32. Azeem, M.I.; Palomba, F.; Shi, L.; Wang, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Inf. Softw. Technol.* **2019**, *108*, 115–138. [[CrossRef](#)]
33. Liu, K.; Kim, D.; Bissyand’e, T.F.; Kim, T.; Kim, K.; Koyuncu, A.; Kim, S.; Le Traon, Y. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*, Montreal, QC, Canada, 25–31 May 2019; pp. 1–12.

34. Tsantalis, N.; Chatzigeorgiou, A. Identification of refactoring opportunities introducing polymorphism. *J. Syst. Softw.* **2022**, *83*, 391–404. [[CrossRef](#)]
35. Woolson, R.F. Wilcoxon Signed-Rank Test. Wiley Encyclopedia of Clinical Trials. 2007. Available online: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9780471462422.eoct979> (accessed on 1 April 2022).
36. Hegedűs, P.; Kádár, I.R. Ferenc, Gyimóthy T. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Inf. Softw. Technol.* **2018**, *95*, 313–327. [[CrossRef](#)]
37. Gharehchopogh, F.; Gholizadeh, H. A comprehensive survey: Whale Optimization Algorithm and its applications. *Swarm Evol. Comput.* **2019**, *48*, 1–24. [[CrossRef](#)]
38. Rosli, M.; Teo, N.H.I.; Yusop, N.S.M.; Mohamad, N.S. Fault prediction model for web application using genetic algorithm. *Int. Conf. Comput. Softw. Modeling (IPCSIT)* **2011**, *14*, 71–77.
39. Ebrahimi, A.; Khamsehchi, E. Sperm whale algorithm: An effective metaheuristic algorithm for production optimization problems. *J. Nat. Gas Sci. Eng.* **2016**, *29*, 211–222. [[CrossRef](#)]