e-Informatica Software Engineering Journal, Volume 18, Issue 1, 2024, pages: 240107, DOI: 10.37190/e-Inf240107

Boosting and Comparing Performance of Machine Learning Classifiers with Meta-heuristic Techniques to Detect Code Smell

Shivani Jain*, Anju Saha*
**Information Technology, GGS Indraprastha University*
shivani.1091@gmail.com, anju_kochhar@yahoo.com

Abstract

Background: Continuous modifications, suboptimal software design practices, and stringent project deadlines contribute to the proliferation of code smells. Detecting and refactoring these code smells are pivotal to maintaining complex and essential software systems. Neglecting them may lead to future software defects, rendering systems challenging to maintain, and eventually obsolete. Supervised machine learning techniques have emerged as valuable tools for classifying code smells without needing expert knowledge or fixed threshold values. Further enhancement of classifier performance can be achieved through effective feature selection techniques and the optimization of hyperparameter values.
Aim: Performance measures of multiple machine learning classifiers are improved by fine tuning its hyperparameters using various type of meta-heuristic algorithms including swarm intelligent, physics, math, and bio-based, etc. Their performance measures are compared to find the best meta-heuristic algorithm in the context of code smell detection and its impact is evaluated based on statistical tests.
Method: This study employs sixteen contemporary and robust meta-heuristic algorithms to optimize the hyperparameters of two machine learning algorithms: Support Vector Machine (SVM) and k-Nearest Neighbors (k-NN). The No Free Lunch theorem underscores that the success of an optimization algorithm in one application may not necessarily extend to others. Consequently, a rigorous comparative analysis of these algorithms is undertaken to identify the best-fit solutions for code smell detection. A diverse range of optimization algorithms, encompassing Arithmetic, Jellyfish Search, Flow Direction, Student Psychology Based, Pathfinder, Sine Cosine, Jaya, Crow Search, Dragonfly, Krill Herd, Multi-Verse, Symbiotic Organisms Search, Flower Pollination, Teaching Learning Based, Gravitational Search, and Biogeography-Based Optimization, have been implemented.
Results: In the case of optimized SVM, the highest attained accuracy, AUC, and F-measure values are 98.75%, 100%, and 98.57%, respectively. Remarkably, significant increases in accuracy and AUC, reaching 32.22% and 45.11% respectively, are observed. For k-NN, the best accuracy, AUC, and F-measure values are all perfect at 100%, with noteworthy hikes in accuracy and ROC-AUC values, amounting to 43.89% and 40.83%, respectively.
Conclusion: Optimized SVM exhibits exceptional performance with the Sine Cosine Optimization algorithm, while k-NN attains its peak performance with the Flower Optimization algorithm. Statistical analysis underscores the substantial impact of employing meta-heuristic algorithms for optimizing machine learning classifiers, enhancing their performance significantly. Optimized SVM excels in detecting the God Class, while optimized k-NN is particularly effective in identifying the Data Class. This innovative

Submitted: 28 Nov. 2023; Revised: 4 Mar. 2024; Accepted: 3 Apr. 2024; Available online: 9 Apr. 2024

fusion automates the tuning process and elevates classifier performance, simultaneously addressing multiple longstanding challenges.

Keywords: Code Smell, Machine Learning, Meta-heuristics, Support Vector Machine, k-Nearest Neighbors, Optimization

1. Introduction

Using software is an integral part of our lives. They are embedded in every aspect of our existence, like education, transportation, entertainment, communication, healthcare, and security. Software systems have become complex and colossal with advancements in science and technology [1]. Designing and developing them takes a mere 30–40% of effort in the complete life cycle of software; the rest is dedicated to maintaining them [2]. Maintaining it includes continuously adding, deleting, and changing artefact functionalities to meet users' needs and satisfaction, which requires more resources and effort [3]. The un-involvement of the maintenance team in the development phase, time crunch, implementation of substandard design practices, tight deadlines, and inexperience of developers provides bedding ground for the spread of code smells throughout the system [4, 5]. A code smell is not a syntax error but may lead to it. It is a structural flaw that violates fundamental design principles and deteriorates code quality [6]. Furthermore, it makes code more complicated to understand and maintain and prevents code from changing, contributing to technical debts. So, it is best to identify and eradicate them whenever a new feature is added, while fixing a bug or during code reviews. It can be corrected by small and disciplined changes in code called refactoring. It is restructuring internal design but ensuring no change in external behaviour. Refactoring improves code quality and reverses software entropy. It makes the system more readable, understandable, efficient, flexible, and maintainable. Identifying and detecting smells is the first step in refactoring, making the system more robust and contemporary.

Code smell detection has several real-world applications in software development and maintenance. Smell detection helps developers identify code areas that may benefit from refactoring. By addressing code smells, developers can enhance code maintainability and readability, reducing technical debt and making the code base more sustainable. Incorporating smell detection as part of the quality assurance process ensures that newly developed or modified code adheres to best practices. In legacy systems where code has accumulated over time, identifying and mitigating code smells can be crucial for improving the health of the code base. It is essential when introducing new features, fixing bugs, or integrating modern technologies. Integrating code smell detection into Continuous Integration (CI) and Continuous Deployment (CD) pipelines ensures that any new changes introduced to the code base adhere to coding standards and best practices. Certain code smells are indicative of potential sources of bugs or errors. By proactively addressing these smells, developers can reduce the likelihood of introducing bugs and enhance the overall reliability of the software. Some code smells, such as duplicated code or inefficient algorithms, can impact the performance of the software. Detecting and addressing these smells can lead to performance improvements in the application. Code smell detection tools can be integrated into various development environments and IDEs, making it convenient for developers to identify and address issues during the coding process. In summary, code smell detection is a valuable practice with tangible benefits regarding code quality, maintainability, and team collaboration. It contributes to the overall improvement of software development processes and the longevity of software systems.

Various code smell detection tools based on the visualization [7], machine based approach [8], and metric evaluation [9] are available in the market and they are of manual [10], automatic [11], and semi-automatic [12] in nature. Although code smell detection tools function effectively, subsequent research has shown essential flaws that jeopardize their widespread use. The agreement between different detectors is also impaired. Tools' strategies rely heavily on setting up detection rules and threshold values. Engineers need in-depth technical knowledge of code smells in order to define these rules [13]. Another problem is that code smells picked up by current detectors can be interpreted differently by specialists. More crucially, to identify smelly code components from non-smelly ones, the majority of smells require the specific threshold values, and naturally, the choice of threshold significantly impacts their count. Additionally, full consideration of size, domain, design, and complexity is typically lacking, which casts doubt on the veracity of other performance indicators [14]. The usage of code smell rules, using insufficient information, and metrics threshold levels are all overcome by supervised machine learning approach.

A supervised machine learning algorithm feeds in independent variables, commonly called training data, to determine the dependent variable's value and improves by learning through examples [15]. Performance measures are assessed, and the algorithm improves response from the difference between expected and generated output. Techniques like hyperparameter tuning [16], SMOTE [17], feature engineering [18], feature selection [19], etc., can be used to enhance results further. When using a machine learning approach, establishing rules and setting thresholds is left up to the algorithm rather than experts, significantly reducing time and effort [20].

This study uses two supervised machine learning classifiers, Support Vector Machine and k-Nearest Neighbors, to identify smelly instances. These classifiers employ a set of hyperparameters to enhance their results, and by choosing the appropriate values of hyperparameters, one may minimize error [21]. A hyperparameter is an external configuration to the model whose value must be defined by an expert as it cannot be determined from the data. The grid search technique can also improve the performance of machine learning algorithms [22]. However, it has many disadvantages, and for an algorithm to work successfully, a specialist must choose hyperparameter values. It takes specialized knowledge, intuition, and frequent trial and error for the best outcomes. It becomes impossible when the number of hyperparameters grows as evaluations grow exponentially. Therefore, meta-heuristic algorithms are employed to choose the appropriate values for the hyperparameters of machine learning algorithms to overcome this challenge and do away with the requirement for experts [23].

Meta-heuristic algorithms are high-level, problem-independent techniques that use gradient-free mechanisms and provide near-optimal solutions to highly complex real-world problems within limited computing time [24]. They search for a solution(s) in a search space that minimizes or maximizes an objective function while fulfilling certain constraints. The success of a meta-heuristic algorithm depends on two processes: exploration and exploitation. Diversification ensures that the whole search space is explored and not confined to specific areas, whereas, in intensification, certain better regions are explored more thoroughly to find a better solution. We have employed various meta-heuristic algorithms, which are stochastic in nature, exploring the search space and exploiting it for the best solution [25].

The no Free Lunch theorem states that no single optimization technique can solve all optimization problems [26]. This theorem underscores that the efficacy of an algorithm for one application may not necessarily translate to success in another optimization problem. It led to the development of more than three hundred meta-heuristic algorithms for conquering numerous optimization problems. Consequently, the prudent approach involves implementing and comparing optimization algorithms to identify the most apt solution for a given context. Their categorization will guide us in understanding their basic work principles and strategies. They are categorized as follows:

– **Evolutionary:** These algorithms are inspired by Darwin's theory of survival of the fittest. The iterative selection, crossover, and mutation process make the stochastically generated population fitter. Evolutionary Programming [27], Genetic Algorithms [28], and Differential Evolution [29] are some evolutionary algorithms.
– **Swarm:** These algorithms utilize the social behaviour and hunting strategies of the genus of animals. Animals or insects work together in an organized manner and constantly interact to explore the entire search space and converge when necessary [30]. Examples of swarm-based algorithms are Particle Swarm Optimization [31], Ant Colony Optimization [32], etc.
– **Physics:** These algorithms imitate physical principles of the universe, such as gravitation, kinematics, fluid mechanics, and electromagnetism [33]. They can be categorized into thermodynamics, classical mechanics, optics, etc. Some physics-based algorithms are Multi-Verse Optimizer [34], Nuclear Reaction Optimization [35], etc.
– **Human:** These algorithms are inspired by the characteristics and behaviour of the human population. Brain Storm Optimization [36] and Battle Royale Optimization [37] are some examples.
– **Others:** Bio-inspired algorithms are based on interactions or biological processes observed in nature. Examples are Virus Colony Search [38], Earthworm Optimization [39], etc. Math-based algorithms such as Hill Climbing always move towards the peak to aim for a better solution [40]. Moreover, the Sine Cosine Algorithm explores and exploits search space using a mathematical model based on sine and cosine functions [41].

1.1. Motivation

Code smell detection has long been a focal point in software engineering research. This study pioneers a transformative approach by integrating meta-heuristic algorithms to amplify the performance of machine learning classifiers, offering a groundbreaking solution to enduring challenges. Employing an optimization algorithm eliminates the need for an expert and the painful task of finding the best hyperparameter values, automating and simplifying the whole process. This innovative fusion elevates classifier performance and presents a profound breakthrough in the field, addressing multiple long-standing issues concurrently.

The following research underscores the influence of meta-heuristic algorithms to optimize machine learning classifiers for code smell detection. The research delves into a comprehensive comparison of sixteen distinct meta-heuristic techniques, evaluating their efficacy in identifying and addressing smells within source code. In this research, the focus lies on the utilization of optimization algorithms to obtain optimal hyperparameter values for SVM and k-NN. A diverse range of optimization algorithms, encompassing Arithmetic, Jellyfish Search, Flow Direction, Student Psychology Based, Pathfinder, Sine Cosine, Jaya, Crow Search, Dragonfly, Krill Herd, Multi-Verse, Symbiotic Organisms Search, Flower Pollination, Teaching Learning Based, Gravitational Search, and Biogeography-Based Optimization, have been implemented.

A comprehensive comparative analysis examines the performance of machine learning classifiers across three scenarios: absence of optimization, grid search application, and optimization implementation. Key performance metrics, including Accuracy, ROC Area Under the Curve (ROC-AUC), F-measure, and execution time, are meticulously documented for analytical purposes. The techniques are implemented 25 times, Acknowledging the stochastic nature of meta-heuristic algorithms. The resultant best and average values are considered for evaluation. Furthermore, a juxtaposition is drawn between these novel meta-heuristic methods and classical algorithms, such as Differential Evolution, Particle Swarm Optimization, Genetic Algorithm, and Simulated Annealing, enhancing the breadth and depth of the comparative study.

The overall contribution of this paper is:

– Enhancing the performance of machine learning classifiers through the utilization of diverse meta-heuristic algorithms.
– Demonstrating the profound influence of meta-heuristic algorithms in optimizing machine learning classifiers, specifically in the context of code smell detection.
– Identifying the optimal meta-heuristic technique for effectively detecting code smells within source code.
– Disclosing the most readily detectable code smell instances.
– Establishing a foundational reference study for prospective qualitative and quantitative comparative research across various domains.

The research paper is structured across seven distinct sections. Commencing with an introductory segment, the paper outlines fundamental concepts, underscores the study's necessity, and articulates its contributions. The introduction is followed by an overview of related research and the pivotal role of the current study. The third section comprehensively details the experiment setup and methodology, accompanied by an illustrative workflow. Results, their in-depth analysis, and statistical tests comprise the fourth section, followed by an expansive discussion in the fifth. The sixth section meticulously examines potential threats to validity, providing corresponding mitigation strategies. The paper culminates in a conclusive seventh section, encapsulating final thoughts and avenues for future exploration. Additionally, the machine learning algorithms employed and concise profiles of the sixteen meta-heuristic algorithms employed are expounded upon within the appendices.

2. Related work

The use of machine learning and optimization algorithms represent highly sought-after and crucial areas of research. Extensive investigations have been undertaken to enhance the efficiency of machine learning algorithms employing diverse techniques, among which the utilization of meta-heuristic algorithms holds significance. Optimization algorithms serve a dual purpose within this landscape like facilitating feature selection and hyperparameter tuning for machine learning algorithms. Moreover, these algorithms find utility in establishing detection rules for code smells, employing tailored threshold values and metrics. Applying optimization algorithms extends to prioritizing refactoring for multiple code smells, predicated on factors such as severity and risk in the context of extensive software systems. The following section outlines pertinent research efforts in this domain.

Hassaine et al. utilized a machine learning-inspired technique called an Immune-based Detection Strategy that imitated the immune system of the human body [42]. IDS is based on the Artificial Immune Systems (AIS) algorithm, which mimics the defense mechanisms of the human immune system. The authors drew a parallel between the human body and system design to develop a detection method that identifies smelly classes, equivalent to pathogens, using some features of the classes in the form of metrics. Compared to DECOR [43] and Bayesian Belief Networks, IDS outperformed in precision and computation time.

Maiga et al. introduced SMURF – Support Vector Machines that consider practitioners' feedback [44]. SMURF was compared with DETEX [43] and BDTEX [45]; it performed better in accuracy, precision, and recall. Fontana et al. implemented multiple variations and boosted versions of J48, Random Forest, Naive Bayes, JRip, SMO, and SVM, constituting 32 machine-learning algorithms to detect four code smells. They concluded that J48 and Random Forest yield the highest performance, and support vector machines are the worst. Boosting only sometimes helps; in some cases, it diminishes performance [46].

Kessentini et al. proposed a multi-objective genetic programming algorithm (MOGP) to generate rules for automatically detecting code smells in Android applications. They identified detection rules for ten smells and evaluated their technique on 184 Android projects. Results projected that average correctness was more than 82% and an average relevance of 77% based on the feedback of active developers of mobile apps [47]. Kaur et al. designed a new meta-heuristic optimization algorithm inspired by sandpipers' searching and attacking behaviours, known as the Sandpiper Optimization Algorithm (SPOA). They collaborated SPOA with B-J48 pruned machine-learning approach to detect five code smells in three open-source software [48].

Jain et al. applied three hybrid feature selection techniques with ensemble machine learning algorithms to improve the performance in detecting code smells. Seven machine learning classifiers with different kernel variations, along with three boosting designs, two stacking methods, and bagging, were implemented. Combining filter-wrapper, filter-embedded, and wrapper-embedded methods was executed for feature selection. After application of hybrid feature selection, performance measure increased, accuracy by 21.43%, ROC AUC value by 53.24%, and F-measure by 76.06% [16].

In other work, Jain et al. implemented 32 machine learning algorithms with feature selection that drastically eliminated the dimensionality curse and improved performance measures. Two correlation methodologies, brute force and random forest, were used to discard irrelevant features with three filter methods: mutual information, fisher score and univariate ROC-AUC. Results showed that the accuracy of machine learning models had surged up to 26.5%, F-measure by 70.9%, the area under the ROC curve had levelled up to 26.74%, and average training time has reduced up to 62 secs as compared to measures of models without feature selection [49].

Boussaa et al. proposed a promising technique to identify detection rules for code smell detection. Two populations evolved simultaneously. The first produced a set of detection rules for detecting code smells, and the second introduced artificial code smells to support the main objective of the first population. When tested on four open-source Java systems, this technique outperformed two single population-based meta-heuristics, Genetic Programming and Artificial Immune Systems [50].

Similarly, Kessentini et al. parallelly used genetic programming to generate code smell detection rules and genetic algorithms to produce code smell examples. Cooperative P-EA outperforms single population evolution and random search [51]. Sahin et al. implied code smell detection as a bilevel problem [52]. They used genetic programming for the upper-level problems, i.e., detection rules and generated artificial code smells for lower-level problems. However, there was no parallelism in this bilevel approach; levels were executed serially. This technique outperformed Genetic Programming, Competitive Coevolutionary Search [50] and non-search-based methods.

Mansoor et al. used multi-objective genetic programming (MOGP) to find the most optimized detection rules to maximize the detection of smells and minimize false detection problems. Five code smells were inspected on seven large open-source systems, and the algorithm achieved 87% precision and 92% recall [53]. Saranya et al. proposed Euclidean distance-based Genetic Algorithm and Particle Swarm Optimization (EGAPSO) to develop detection rules that outperformed other detection methods like Genetic Algorithm, DECOR, Parallel Evolutionary Algorithm, and Multi-Objective Genetic Programming. The approach was tested on open-source projects like the Gantt Project and Log4j to identify the five code smells [54].

Kannan developed hybrid particle swarm optimization with mutation (HPSOM) to formulate detection rules using appropriate metrics and thresholds. He then compared its performance with other evolutionary techniques like the parallel evolutionary algorithm, genetic algorithm, genetic programming, and particle swarm optimization. HPSOM outperformed all of them by achieving a precision of 94% and recall of 92%. He worked with nine open-source projects and detected five code smells: blob, data class, spaghetti code, functional decomposition, and feature envy [55]. Moatasem et al. used a whale optimization algorithm to formulate ideal detection rules for nine code smells. Equations were tested on five medium and large-size open-source projects. Results were better than other search-based algorithms; 94.24% precision and 93.4% recall were observed [56].

Amal et al. evaluated a refactoring series to make the system more robust using a genetic algorithm and artificial neural network (ANN) [57]. They compared their techniques with other search-based refactoring techniques, such as the IGA technique presented by Ghannem et al. [58] and a design defect detection and correction tool called JDeodorant [59]. Dea et al. used distributed evolutionary algorithms where many evolutionary algorithms with different adaptations (fitness functions, solution representation, and change operators) are implemented in parallel to get a series of refactoring. Cooperative D-EA outperforms single population evolution and random search based on a benchmark of eight sizable open-source systems where more than 86% of code smells are fixed using the suggested refactoring [60].

Saranya et al. used the Strength Pareto Evolutionary Algorithm (SPEA) to prioritize the list of refactorings. Blob, Functional Decomposition, Shotgun Surgery, Data Class, Schizophrenic Class, and Swiss Army Knife were considered and tested on two open-source systems, Xerces-J and J Hot Draw. SPEA outperformed Chemical Reaction Optimization (CRO) and Non-dominated Sorting Genetic Algorithm in prioritizing code smell correction tasks [61].

Large-scale systems have a volume of code smells, and prioritizing them according to risk, impact, importance, and severity is an efficient way to eliminate them. Ouni et al. used chemical reaction optimization to find a series of refactoring to remove smells according to the risk and other factors involved. Seven code smells were tested on five medium to large-scale open-source systems. The proposed technique outsmarted existing methods compared to Genetic Algorithm, Simulated Annealing, and Particle Swarm Optimization [62]. Using the Sandpiper Optimization Algorithm, Kaur et al. detected the severity of five harmful code smells, namely blob, feature envy, data class, functional decomposition, and spaghetti code. The approach was tested on four open-source Java software: Gantt-Project, Log4j, and two different versions of Xerces. Studies showed that many code smells could be refactored with a severe decrease in refactoring effort [63].

This work utilizes the influence of sixteen powerful meta-heuristic algorithms to optimize machine learning algorithms. This approach addresses problems such as evaluating the best hyperparameter values of machine learning algorithms to elevate their performance. This fusion eliminates needing an expert, reduces time and effort, and effectively deciphers complex software engineering challenges.

3. Research methodology

The following section delves into the research questions, studies answers, description of code smells analyzed, datasets used, and complete experimentation settings.

3.1. Research questions addressed

This study aims to investigate the following research questions:
- **RQ 1:** Does using meta-heuristic algorithms for optimizing machine learning classifiers boost their performance for detecting code smell in complex software systems?
- **RQ 2:** How significant is the impact of optimization of machine learning algorithms with meta-heuristic techniques on its overall performance?
- **RQ 3:** Given the meta-heuristic algorithms, which yields the best performance in optimizing classifiers to detect code smell and why?
- **RQ 4:** How does our approach perform compared to existing machine learning based techniques?

3.2. Code Smells investigated

This study entails the optimization of two distinct machine learning classifiers by utilizing a comprehensive array of sixteen selected meta-heuristic algorithms, a strategy aimed at refining performance metrics. The primary focus of this optimization effort is detecting four distinct types of code smells, each of which bears distinctive characteristics and implications. Specifically, two class-level code smells [64] under scrutiny are the Data Class and the God Class. Data Class is a passive container for data, housing attributes, getters, and setters intended for use by other encapsulating classes. This class does not engage in the execution of substantial operations on its stored data. It impacts data abstraction and encapsulation properties of the system. It can be refactored with the Encapsulate Collection, Move Method, Extract Method, Encapsulate Field, and Hide Method.

God Class is characterized by its tendency for extensive functionality implementation, leveraging attributes sourced from various other classes. This behaviour results in a notably intricate and expansive class structure that is difficult to understand and maintain. It promotes code duplication and complex methods. It affects cohesion, coupling, complexity, and size of the system. It can be refactored with Extract Class, Extract Subclass, Extract Interface, and Duplicate Observed Data. Further delving into the method-level code smells [65], two distinct categories are investigated: Feature Envy and Long Method. Feature Envy manifests when a method tends to access attributes originating from external classes while interacting with data derived from these classes. It harms the coupling and data abstraction properties of code. It can be treated with Extract Method and Move Method refactoring.

The long method is overly extensive and draws information from other methods. These methods often seek to centralize a class's intelligence and encompass many features. It impacts coupling, cohesion complexity, and the size of the whole system. One can eliminate the Long Method with the Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, Replace Method with Method Object, and Decompose Conditional refactoring techniques. While individually diverse in their manifestations, these code smells collectively embody some of the most insidious and prevalent issues encountered within software code bases [66]. Their systematic detection and subsequent remediation are pivotal to enhancing software quality, maintainability, and comprehensibility.

3.3. Datasets used

In this research, we have used four datasets curated by Fontana et al. [46] to facilitate the classification of specific code smells. These datasets have been assembled from 74 compilable Java systems sourced from the Qualitas Corpus [67]. Collectively, these systems span a diverse spectrum of application domains and exhibit a wide range of sizes, thereby endowing our research with a robust and comprehensive foundation. Datasets included an intra-system setup to prevent machine learning models from succumbing to over-fitting. List of all heterogeneous systems is included in the Table 1 of supplementary file[1]. They developed the Design Features and Metrics for Java (DFMCFJ) tool, underpinned by the Eclipse JDT Library, which extracts a rich array of object-oriented metrics at multiple granularities, spanning project, package, class, and method levels.

Each dataset comprises 420 data points, partitioned into 280 negative samples, signifying the absence of code smell, and 140 positive samples, denoting presence. Datasets are rooted in a comprehensive assessment of object-oriented metrics, spanning multiple strata of code design, such as coupling, complexity, cohesion, and size. Details of all metrics used in datasets are mentioned in the Table 2 of supplementary file. The datasets are judiciously leveraged by stratified sampling techniques, thus generating balanced and labeled datasets. It is imperative to note that each entry within these datasets is expressly associated with either a method or a class. Each row is labeled with the help of Advisor (Code smell detection tools such as PMD, iPlasma, Fluid Tool, and Antipattern Scanner) and validated by trained MSc students after thorough discussion.

The metrics encompass a comprehensive view of code design focused on method-level code smells, extending across project, package, class, and method levels. The method-level datasets harnessed 82 distinct metrics. Conversely, the datasets dedicated to class-level code smells have a set of metrics spanning project, package, and class levels, totaling 61 in number. This approach ensures that our research is firmly grounded in a wealth of empirical data, encompassing a multifaceted view of code characteristics. Thus, it is poised to yield comprehensive insights into code smell detection. Datasets are made available by Fontana et al.[2].

3.4. Experimentation setup

This research aims to enhance the performance of two prominent machine learning classifiers: Support Vector Machine (SVM) and k-Nearest Neighbors (k-NN). This enhancement is pursued by utilizing sixteen distinct meta-heuristic algorithms, expertly calibrated to fine-tune the hyperparameters governing these classifiers. A comprehensive assessment and comparative analysis of the efficacy and impact of these meta-heuristic algorithms within

[1]Details of Datasets – https://drive.google.com/file/d/1Jt3jnRDUKgCvN-ZUM6xwtZTut8GuIFcL/view?usp=sharing

[2]Datasets – https://drive.google.com/file/d/15aXc_el-nx4tQwU3khunQ-I5ObSA1-Zb/

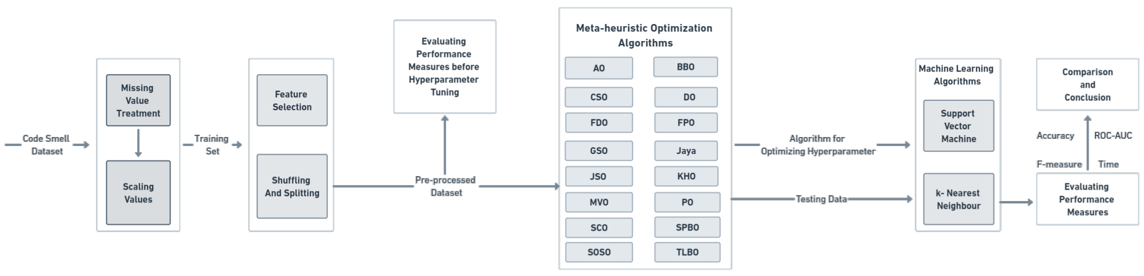

Figure 1. Workflow

machine learning is pursued. This research unfolds within the computational domain of Python [68], with the scikit-learn [69] framework serving as the foundational infrastructure. To visually represent the holistic research process, we have encapsulated the workflow of our study in Figure 1.

The data pre-processing ensures the cleanliness and readiness of datasets before they are divided into training and test sets [70]. This preliminary data grooming is essential, as it directly influences the quality and appropriateness of the data employed for model training. It ameliorates model performance, reduces training duration, mitigates over-fitting risks, and enhances model interpretability [71]. The following steps are taken to prepare datasets. The missing data values are replaced with a zero value due to intra-system settings. A scaling operation establishes an equitable ground for our independent variables. This normalization strategy serves the dual purpose of bridging any inherent gaps between features and curbing the potential introduction of bias.

The next step in the data refinement entails identifying and eliminating constant, quasi-constant, and duplicated features. These categories encapsulate features that either exhibit an unchanging value across instances (constant features), furnish redundant or repetitive information (duplicate features), or verge on maintaining nearly identical values for every instance (quasi-constant features) [72]. The independent variables should not be correlated and reasonably correlate with the dependent variable. Thus, correlated independent variables are precisely identified and eliminated. Pearson's correlation coefficient [73] is employed for the same, a well-established statistical metric renowned for its adeptness in quantifying the linear relationship between two variables.

The dataset is randomly partitioned into two sets for training and testing purposes. The training dataset is formulated, constituting 80% of the entire dataset, while the test dataset accounts for the remaining 20%. K-fold cross-validation is employed to assess a predictive model's performance and generalization ability. It scrutinizes the efficacy and proficiency of machine learning models when confronted with previously unseen data. This method divides the dataset into K parcels of identical size, denoted as "folds". We set the value of K to 10, signifying ten equivalent folds [74]. The model is iteratively trained on $k-1$ folds while reserving one fold for validation. This cyclic process iterates K times, each fold having a turn as the validation set. The final model's performance is the aggregation of performance scores garnered across all K iterations. Machine learning algorithms are implemented, and their hyperparameters are obtained from meta-heuristic optimization techniques. Employing meta-heuristic algorithms explores and scrutinizes the parameter space to identify the optimal hyperparameters that give the peak performance of the machine learning classifier. The working of meta-heuristic algorithms is presented in Figure 2, which is explained in the following section:

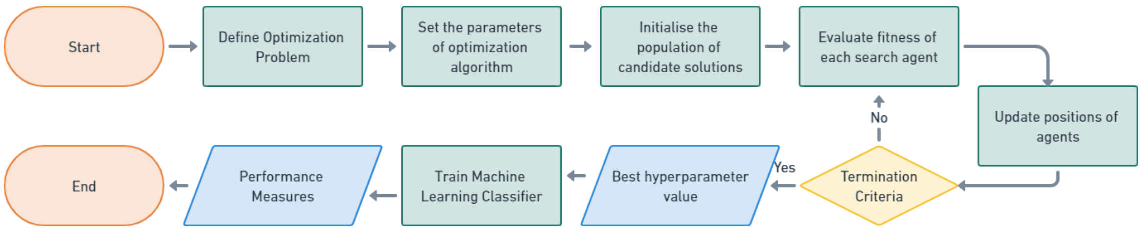
Start
Define Optimization Problem
Set the parameters of optimization algorithm
Initialise the population of candidate solutions
Evaluate fitness of each search agent
Update positions of agents
Termination Criteria
No
Yes
Best hyperparameter value
Train Machine Learning Classifier
Performance Measures
End

Figure 2. Flowchart: Steps involved in working of meta-heuristic algorithms

1. **Define the optimization problem.** At its core, the optimization task revolves around minimizing errors in machine-learning classifiers. This entails identifying optimal hyperparameter values, a prerequisite before the classifier's training phase, and directing and shaping its behaviour. The Support Vector Machine (SVM) has two critical hyperparameters, "C" and "gamma," for optimization. Simultaneously, the k-Nearest Neighbors (k-NN) classifier undergoes refinement by selecting optimal "n_neighbors" and "p" values. In the case of SVM, the Radial Basis Function (RBF) kernel is a natural choice, endowed with a track record of superior performance.
2. **Set parameter values for meta-heuristic algorithm.** The parameter values, such as population size, iteration count, generation specifications, etc., are initialized. These selected parameter values for meta-heuristic algorithms and the hyperparameter spectrum of machine learning classifiers are thoughtfully detailed in Table 1. Arriving

Table 1. Parameter values used for meta-heuristics algorithms and range of hyperparameters for machine learning classifiers

Ref	Meta-heuristic algorithms	Year	Category	Parameters
[75]	Arithmetic Optimization	2021	Math	size = 5, alpha = 5, mu = 0.5
[76]	Jellyfish Search Optimization	2021	Swarm	jellyfishes = 5, eta = 4, beta = 3, gamma = 0.1, c_0 = 0.5
[77]	Flow Direction Optimization	2021	Physics	size = 5, beta = 8
[78]	Student Psychology Based Optimization	2020	Human	size = 5, generations = 50
[79]	Pathfinder Optimization	2019	Swarm	size = 5, generations = 50
[80]	Sine Cosine Optimization	2016	Math	solutions = 5, a_linear_component = 2, r1 = 2
[81]	Jaya Optimization	2016	Swarm	size = 5, generations = 50
[82]	Crow Search Optimization	2016	Swarm	size = 5, ap = 0.02, fL = 0.02
[83]	Dragonfly Optimization	2016	Swarm	size = 3, generations = 50
[84]	Krill Herd Optimization	2016	Swarm	size = 5, generations = 50, mutation_rate = 0.1, eta = 1, c_t = 1, mu = 1, elite = 0
[85]	Multi-Verse Optimization	2015	Physics	universes = 5
[86]	Symbiotic Organisms Search	2014	Bio	size = 5, eta = 1, generations = 50, mutation_rate = 0.1
[87]	Flower Pollination Optimization	2012	Evolutionary	flowers = 3, gamma = 0.5, lamb = 1.4, p = 0.8, beta = 1.5
[88]	Teaching Learning Based Optimization	2012	Human	size = 5, generations = 50
[89]	Gravitational Search Optimization	2009	Physics	swarm_size = 5
[90]	Biogeography-Based Optimization	2008	Bio	size = 5, mutation_rate = 0.1, elite = 0, eta = 1, gens = 50
[91]	Support Vector Machine	–	–	C = [10, 1000], gamma = [0.05, 10], kernel = rbf
[92]	k-Nearest Neighbors	–	–	n_neighbors = [3, 50], p = [1, 2]

at these optimal values is underpinned by an exhaustive investigative process involving comprehensive research, literature survey, and methodical empirical experimentation. These chosen values are supported by foundational research. Furthermore, the machine learning algorithm's hyperparameter range is selected by drawing insights from research, experimentation, and practical experience.

3. **Generate initial population.** During this phase, the algorithm initializes the population of the candidate solution, a process that can involve randomized generation or employ alternative strategies [93]. For optimization, a comprehensive ensemble of sixteen meta-heuristic optimization algorithms is harnessed. These encompass Arithmetic, Jellyfish Search, Flow Direction, Student Psychology Based, Pathfinder, Sine Cosine, Jaya, Crow Search, Dragonfly, Krill Herd, Multi-Verse, Symbiotic Organisms Search, Flower Pollination, Teaching Learning Based, Gravitational Search, and Biogeography-Based Optimization. The objective is determining the optimal hyperparameter values for classifiers to augment the performance. The behaviour, working principle, and learning equation of each meta-heuristic algorithm are mentioned in the Appendix B.
4. **Fitness evaluation.** In this phase, the fitness of each candidate solution undergoes scrutiny. This undertaking entails training the machine learning classifier, which utilizes the hyperparameters intrinsic to each candidate solution. Subsequently, their performance is appraised primarily on a validation dataset. A designated performance metric, accuracy or F-measure, is wielded as the discerning fitness function, serving as the threshold to quantify the efficacy of each solution.
5. **Updating Position Vectors.** The candidate solutions are updated based on the information obtained in the fitness evaluation step and the rules of the meta-heuristic algorithm. This involves adjusting the position of each search agent, updating the best-performing agent, or selecting new agents to replace under-performing ones.
6. **Termination Criteria.** The position updation continues until a stopping criterion is met. This could involve checking if the maximum number of iterations has been reached, if the best-performing solution has not improved in a certain number of iterations, or if the solutions have converged to a certain level of accuracy. For this study, stopping criteria are set to 50 iterations because experimentation found that this number is sufficient to converge to an appropriate solution. As optimization algorithms are stochastic, the process is repeated 25 times to achieve the best and average values.
7. **Output.** If the termination criteria have been met, the algorithm finds the best-performing hyperparameters found; otherwise, it returns to step 4 and continues the optimization process.
8. **Training.** In this step, the machine learning classifier is trained with the best hyperparameter values derived from the above step. Our study has chosen SVM and k-NN for optimization, and its working is mentioned in the Appendix A.
9. **Evaluation.** The performance measures of machine learning classifiers, such as accuracy, F-measure, and ROC-AUC, are evaluated for analysis and comparison purposes.

By using meta-heuristic algorithms to search for the best hyperparameter values, we can avoid the time-consuming and error-prone process of manual tuning [94]. The aim is to minimize the error component, that is, the difference between the predicted and actual value of the target variable. Accuracy, ROC Area Under the Curve (ROC-AUC), F-measure, and execution time are recorded for analysis after optimization. Deviation from the standard and time for one iteration is an average of 25 executions. Further, the final performance is compared with the scenario when no such optimization technique is used, the grid search method is used, and classic meta-heuristic algorithms such as Differential

Evolution, Particle Swarm Optimization, Genetic Algorithm, and Simulated Annealing are used. Results are presented in tabular form in the next section with a qualitative and quantitative analysis.

3.5. Performance measures

Following are the performance measures for classification problems that have been used to analyze and compare machine learning algorithms:

3.5.1. Accuracy

It is the sum of all correctly predicted code smells divided by the total number of smells present in the code. It can be calculated by the formula:

$$\frac{\textit{Detected Code Smells}}{\textit{Total Code Smells Present}}$$

3.5.2. *F*-measure

F-measure is the weighted harmonic mean of recall and precision.

$$F\text{-measure} = 2 \cdot \frac{\textit{precision} \cdot \textit{recall}}{\textit{precision} + \textit{recall}}$$

Precision is the number of smells predicted as smelly and are also actually smelly. It is calculated as:

$$\frac{(\textit{Present Code Smells}) \bigcap (\textit{Detected Code Smells})}{(\textit{Detected Code Smells})}$$

Recall is the number of instances that are actually smelly and are also predicted correctly as smelly. It is calculated as:

$$\frac{(\textit{Present Code Smells}) \bigcap (\textit{Detected Code Smells})}{(\textit{Present Code Smells})}$$

3.5.3. ROC-AUC

Receiver Operating Characteristics (ROC) is a plot of the False Positive Rate (on the x-axis) versus the True Positive Rate (on the y-axis) for every possible classification threshold. The area calculated under the ROC curve is known as ROC-AUC.

$$\text{ROC-AUC} = \frac{1 + TP_{\text{rate}} - FP_{\text{rate}}}{2}$$

It represents the probability that a machine learning model ranks a randomly chosen positive observation higher than a randomly chosen negative observation, and thus it is a useful metric even for skewed datasets [95].

4. Results and analysis

Leveraging meta-heuristic algorithms to attain the optimal hyperparameter values for machine learning algorithms is a prominent strategy for achieving peak performance. In this study, the Support Vector Machine and k-Nearest Neighbors undergo optimization by using sixteen meta-heuristic methods. The following evaluation encompasses a comprehensive comparison and scrutiny of their respective performance measures, facilitating the identification of the most effective optimization algorithm. This analysis accounts for four distinct code smells, categorized into class-level and method-level varieties. The research findings present results via tables and in-depth analytical modules.

Tables 3 through 6 comprehensively examine the performance measures for the Support Vector Machine (SVM) in conjunction with various meta-heuristic algorithms across all four distinct code smells. Tables 8 to 11 meticulously present the performance metrics of k-Nearest Neighbors (k-NN) when optimized with various meta-heuristic algorithms, encompassing all four distinct code smells. Notably, all sixteen meta-heuristic algorithms are executed twenty-five times to derive the average values for all performance measures. Across these twenty-five iterations, the most exceptional performance measure is recorded for each code smell. This measure is compared against the original performance metrics obtained when no optimization technique is applied. This comparative analysis seeks to elucidate the precise impact that optimization algorithms wield over machine learning processes.

It is important to note that in cases where the F-measure is not always measurable, the difference related to this metric is omitted from consideration. Additionally, standard deviation values are computed, providing insights into the degree of variation within the data. Furthermore, the time required per iteration is documented, offering a glimpse into the computational efficiency of these optimization algorithms. As part of this comprehensive evaluation, the performance of the selected meta-heuristic algorithms is juxtaposed with that of four widely recognized and fundamental techniques: Genetic Algorithm, Differential Evolution, Particle Swarm Optimization, and Simulated Annealing. The best performance measures are denoted in bold to highlight the most outstanding results. Subsequently, this narrative will delve into a detailed and systematic analysis of how optimization impacts the performance of classifiers in the context of each specific code smell.

4.1. Support Vector Machine

Table 2 presents an extensive evaluation of the performance metrics, including accuracy, ROC-AUC, and F-measure, of the Support Vector Machine (SVM). This evaluation encompasses instances without optimization and when grid search is executed. In grid search, range of hyperparameter selected is as follows – C: [0.1, 1, 10, 100, 1000], gamma: [1, 0.1, 0.01, 0.001, 0.0001]. Without optimization, the God Class exhibits the highest values, recording accuracy and ROC-AUC of 73.89% and 84.88%, respectively. When employing grid search, best metrics are noted as 75% for accuracy and 73.69% for ROC-AUC. Notably, the Data Class demonstrates some gains in accuracy of 3.61%. With grid search ROC-AUC value always decreased, even up to 11.19%. It's noteworthy that grid search fails to improve results in all cases, even decreased in some, underscoring the necessity for alternative techniques to achieve optimal outcomes.

Table 2. Performance measures of Support Vector Machine (SVM)

	Without optimization			Grid search		
Code smells	Accuracy	ROC-AUC	F-measure	Accuracy	ROC-AUC	F-measure
Data class	67.92	62.78	0	71.53	55.83	**21.67**
Feature envy	64.17	68.72	**18**	61.94	67.25	0
God class	**73.89**	**84.88**	0	**75**	**73.69**	0
Long method	64.17	66.22	**18**	64.31	61.92	13

4.1.1. Data class

The Data Class detection outcomes are carefully presented in Table 3. Among the meta-heuristic algorithms, **Symbiotic Organisms Search Optimization** emerges as the unequivocal champion, consistently exhibiting the maximum, optimal average, and most substantial improvements in all three performance metrics. Symbiotic Organisms Search Optimization attains a peak accuracy of 97.64% and the finest average accuracy of 97.64% while achieving an impressive increase of 29.72% (Δ_1) when juxtaposed with non-optimized results. Furthermore, this algorithm achieves highest ROC-AUC of 100% and a remarkable 99.96% as the best average value, with a minimal deviation of 0.15; these achievements correspond to a notable surge of 37.22% (Δ_2) compared to the non-optimized baseline. For F-measure, both Jellyfish Search and Symbiotic Organisms Search Optimization emerge as front runners, securing the highest maximum and optimal average value of 96%. Among the algorithmic contenders, Dragonfly Optimization appears the swiftest, boasting an execution time of 6.97 seconds per iteration. In contrast, Pathfinder Optimization is the most time-consuming option for detecting Data Class.

4.1.2. Feature envy

Table 4 encapsulates the findings identifying Feature Envy through SVM and diverse meta-heuristic algorithms. Crow Search Optimization stands out with its highest recorded accuracy of 86.11%. In parallel, Dragonfly Optimization attains the highest ROC-AUC and F-measure, registering remarkable values of 99.33% and 94.29%, respectively. When considering the average performance metrics, Symbiotic Organisms Search Optimization emerges as the frontrunner, achieving the best average accuracy of 94.36% and the highest average F-measure, 92.37%. These achievements come with minimal deviations of 1.34 and 1.18, respectively. Pathfinder Optimization secures the best average ROC-AUC value, an impressive 98.57%. Furthermore, Dragonfly Optimization is characterized by the most substantial improvements in accuracy and ROC-AUC values, attaining increments of 32.08% (Δ_1) and 30.61% (Δ_2), respectively, compared to the non-optimized baseline. It is also the swiftest optimization algorithm, with an execution time of merely 7.09 seconds per iteration. In stark contrast, Pathfinder Optimization ranks as the slowest algorithm in execution time, with an enduring 242.03 seconds per iteration. The results underscore **Dragonfly Optimization** as the most proficient algorithm for Feature Envy detection when coupled with SVM.

4.1.3. God class

Table 5 furnishes the outcomes pertinent to identifying the God Class, showcasing the results obtained when utilizing SVM with diverse meta-heuristic algorithms. **Sine Cosine**

Table 3. Performance measures of SVM optimized with meta-heuristic algorithms for Data Class

Performance measures	Accuracy				ROC-AUC				F-measure			Time of one
Optimization Algorithms	Best [%]	Avg [%]	Std	Δ_1	Best [%]	Avg [%]	Std	Δ_2	Best [%]	Avg [%]	Std	iteration [s]
Arithmetic Optimization	91.67	75.88	7.76	23.75	95.83	90.33	2.84	33.05	83.90	16.87	24.77	15.31
Jellyfish Search Optimization	97.50	97.50	0.00	29.58	98.67	98.67	0.00	35.89	**96.00**	**96.00**	0.00	36.65
Flow Direction Optimization	90.28	90.28	0.00	22.36	94.00	94.00	0.00	31.22	88.06	88.06	0.00	203.12
Student Psychology Based Optimization	96.39	93.72	9.04	28.47	99.33	97.92	3.83	36.55	94.67	90.88	18.55	47.70
Pathfinder Optimization	90.56	90.51	0.22	22.64	97.83	97.83	0.00	35.05	84.71	84.71	0.00	214.55
Sine Cosine Optimization	94.17	94.17	0.00	26.25	98.89	98.89	0.00	36.11	88.00	88.00	0.00	32.08
Jaya Optimization	90.28	87.10	7.28	22.36	96.11	94.72	3.76	33.33	77.90	65.44	28.56	13.54
Crow Search Optimization	85.83	74.39	7.09	17.91	96.83	89.96	3.20	34.05	87.71	34.63	24.08	25.38
Dragonfly Optimization	95.42	93.95	3.96	27.50	99.17	98.44	0.59	36.39	95.14	88.18	11.68	**6.97**
Krill Herd Optimization	82.78	80.98	1.41	14.86	90.17	89.05	0.99	27.39	78.81	75.53	1.93	21.30
Multi-Verse Optimization	91.53	86.73	9.48	23.61	98.11	98.11	0.00	35.33	86.71	71.32	30.66	29.21
Symbiotic Organisms Search Optimization	**97.64**	**97.64**	0.00	**29.72**	**100.00**	**99.96**	0.15	**37.22**	**96.00**	**96.00**	0.00	85.12
Flower Pollination Optimization	95.42	89.70	10.25	27.50	97.50	97.01	0.97	34.72	91.33	72.40	34.80	8.14
Teaching Learning Based Optimization	96.39	96.39	0.00	28.47	99.17	99.17	0.00	36.39	91.33	91.33	0.00	51.06
Gravitational Search Optimization	95.28	76.20	8.46	27.36	99.44	98.24	1.63	36.66	91.24	32.44	32.99	20.07
Biogeography-Based Optimization	94.03	83.57	12.92	26.11	97.56	96.79	1.76	34.78	91.14	41.58	41.08	37.80
Differential Evolution	66.67	66.67	0.00	−1.25	67.64	60.72	8.49	4.86	0.00	0.00	0.00	14.03
Particle Swarm Optimization	63.06	63.06	0.00	−4.86	65.44	60.03	7.23	2.66	0.00	0.00	0.00	36.5154
Genetic Algorithm	65.42	59.74	1.61	−2.50	68.42	56.70	5.16	5.64	9.00	5.76	4.32	9.23011
Simulated Annealing	66.67	64.37	0.68	−1.25	58.00	54.36	1.53	−4.78	10.00	0.80	2.71	13.14

Table 4. Performance measures of SVM optimized with meta-heuristic algorithms for Feature Envy

Performance measures	Accuracy				ROC-AUC				F-measure			Time of one
Optimization Algorithms	Best [%]	Avg [%]	Std	Δ_1	Best [%]	Avg [%]	Std	Δ_2	Best [%]	Avg [%]	Std	iteration [s]
Arithmetic Optimization	79.58	78.74	0.96	15.41	82.86	71.12	4.25	14.14	13.33	11.73	4.33	14.77
Jellyfish Search Optimization	85.69	83.88	6.14	21.52	93.06	93.06	0.00	24.34	82.05	82.05	0.00	39.67
Flow Direction Optimization	85.83	84.90	0.41	21.66	97.67	97.67	0.00	28.95	67.00	65.86	0.57	228.73
Student Psychology Based Optimization	84.58	80.31	5.24	20.41	92.50	91.69	2.12	23.78	51.00	46.92	13.84	51.60
Pathfinder Optimization	88.19	88.19	0.00	24.02	98.57	**98.57**	0.00	29.85	58.33	58.33	0.00	242.03
Sine Cosine Optimization	84.44	84.44	0.00	20.27	98.33	98.33	0.00	29.61	55.67	55.67	0.00	16.10
Jaya Optimization	84.44	79.33	6.26	20.27	96.94	96.94	0.00	28.22	61.33	49.07	24.53	14.49
Crow Search Optimization	86.11	58.20	8.91	21.94	92.50	82.43	4.76	23.78	81.13	13.18	27.95	19.21
Dragonfly Optimization	**96.25**	88.27	10.85	**32.08**	**99.33**	97.00	1.96	**30.61**	**94.29**	82.00	29.10	**7.09**
Krill Herd Optimization	65.42	65.42	0.00	1.25	57.83	57.69	0.34	−10.89	0.00	0.00	0.00	29.36
Multi-Verse Optimization	91.67	86.42	10.50	27.50	97.00	97.00	0.00	28.28	88.95	46.26	44.44	14.31
Symbiotic Organisms Search Optimization	95.56	**94.36**	1.34	31.39	99.00	98.25	0.75	30.28	93.81	**92.37**	1.18	88.54
Flower Pollination Optimization	77.22	74.97	3.61	13.05	93.61	91.26	7.35	24.89	41.33	24.80	20.25	8.98
Teaching Learning Based Optimization	84.72	83.57	0.34	20.55	97.56	97.56	0.00	28.84	61.33	59.41	0.39	54.32
Gravitational Search Optimization	79.03	65.16	3.43	14.86	92.56	89.52	4.94	23.84	64.57	2.58	12.65	16.40
Biogeography-Based Optimization	84.31	73.36	6.32	20.14	95.50	91.32	6.78	26.78	60.67	26.11	28.73	27.83
Differential Evolution	61.94	61.94	0.00	−2.23	67.00	65.80	2.75	−1.72	0.00	0.00	0.00	16.26
Particle Swarm Optimization	71.67	71.67	0.00	7.50	61.67	59.53	3.48	−7.05	0.00	0.00	0.00	51.15
Genetic Algorithm	72.78	72.78	0.00	8.61	71.73	61.39	5.22	3.01	0.00	0.00	0.00	13.05
Simulated Annealing	63.06	63.06	0.00	−1.11	62.17	60.33	1.04	−6.55	0.00	0.00	0.00	17.17

Optimization is the most proficient performer, consistently achieving the highest values across all three performance metrics. Sine Cosine Optimization attains a remarkable maximum accuracy of 98.75%, with the best average accuracy standing at 98.75%, as well. This impressive achievement represents a substantial improvement of 24.86%($\varDelta_1$) compared to the non-optimized baseline. Furthermore, the algorithm achieves a maximum ROC-AUC of 100%, with the best average ROC-AUC reaching 100%. Notably, Student Psychology Based and Jaya Optimization also attain maximum and best average ROC-AUC values of 100%. Symbiotic Organisms Search Optimization also secures the best ROC-AUC value of 100%, effectively tying with its counterparts. Highest gain in ROC-AUC value observed is 15.12%. Regarding the F-measure, the maximum and best average value achieved is 98.57%. Notably, Krill Herd Optimization consistently ranks as the poorest-performing algorithm across all cases, exhibiting subpar results. Regarding computational efficiency, Dragonfly Optimization is the fastest algorithm in this context, with an execution time of 9.17 seconds per iteration. The results highlight Sine Cosine Optimization as the preeminent algorithm for detecting the God Class when combined with SVM.

4.1.4. Long method

Table 6 presents the findings of detecting Long Method using the Support Vector Machine (SVM) in conjunction with various meta-heuristic algorithms. Among these algorithms, **Symbiotic Organisms Search Optimization** stands out as the top-performing meta-heuristic, consistently exhibiting the highest values across all three performance metrics. Symbiotic Organisms Search Optimization attains an impressive maximum accuracy of 96.39%, a maximum ROC-AUC of 100%, and a maximum F-measure of 94.57%. It's also worth noting that Flower Pollination Optimization achieves a perfect ROC-AUC score of 100%. When considering the best average performance, Symbiotic Organisms Search Optimization secures the highest average accuracy of 95.34%, with a deviation of 0.75. Additionally, it achieves a best average ROC-AUC of 99.36% with a deviation of 0.40 and a best average F-measure of 93.50% with a deviation of 0.53. These findings underscore the algorithm's consistent and robust performance. Regarding improvements over the non-optimized baseline, Symbiotic Organisms Search Optimization achieves the maximum hike in accuracy and ROC-AUC, with increases of 32.22% ($\varDelta_1$) and 33.78% ($\varDelta_2$), respectively. Conversely, Krill Herd Optimization consistently ranks as the poorest-performing algorithm across all scenarios. Regarding computational efficiency, Dragonfly Optimization is the fastest technique, with an execution time of 8.17 seconds per iteration. To summarize, these results emphasize the superiority of Symbiotic Organisms Search Optimization for detecting Long Method when paired with SVM.

4.2. k-Nearest neighbors

Table 7 comprehensively presents the performance metrics encompassing the best and average values for accuracy, ROC-AUC, and F-measure concerning k-Nearest Neighbors (k-NN). The results are categorized into two scenarios: one when no optimization is applied and another when grid search is employed. In grid search, the hyperparameter spectrum is as follows – k: [ranges from 1 to 60], p: [1, 1.2, 1.5, 2]. In the absence of optimization, it is evident that the God Class stands out with the best accuracy of 71.81%, ROC-AUC of 68.25% and an F-measure of 49.33%. Upon the introduction of the grid search, the Long Method emerged as the leader in accuracy, achieving a notable 77.78%. Simultaneously, the

Table 5. Performance measures of SVM optimized with meta-heuristic algorithms for God Class

Performance measures	Accuracy				ROC-AUC				F-measure			Time of one
Optimization Algorithms	Best [%]	Avg [%]	Std	Δ_1	Best [%]	Avg [%]	Std	Δ_2	Best [%]	Avg [%]	Std	iteration [s]
Arithmetic Optimization	71.67	71.67	0.00	−2.22	97.78	91.59	5.23	12.90	33.00	1.79	6.77	16.56
Jellyfish Search Optimization	93.89	90.32	9.66	20.00	95.11	94.95	0.54	10.23	92.29	92.29	0.00	33.53
Flow Direction Optimization	95.56	95.56	0.00	21.67	98.57	98.57	0.00	13.69	91.33	91.33	0.00	237.48
Student Psychology Based Optimization	94.03	91.14	7.66	20.14	**100.00**	**100.00**	0.00	**15.12**	89.24	71.39	35.70	29.86
Pathfinder Optimization	90.42	90.42	0.00	16.53	97.94	97.94	0.00	13.06	85.95	85.87	0.11	226.15
Sine Cosine Optimization	**98.75**	**98.75**	0.00	**24.86**	**100.00**	**100.00**	0.00	**15.12**	**98.57**	**98.57**	0.00	15.41
Jaya Optimization	93.75	84.90	11.80	19.86	**100.00**	**100.00**	0.00	**15.12**	91.24	58.39	43.79	13.98
Crow Search Optimization	78.75	70.94	1.76	4.86	95.56	81.78	5.45	10.68	57.00	6.95	15.14	20.07
Dragonfly Optimization	95.00	82.34	12.14	21.11	100.00	94.02	2.84	**15.12**	92.29	76.71	28.44	**9.17**
Krill Herd Optimization	59.72	59.72	0.00	−14.17	56.00	56.00	0.00	−28.88	0.00	0.00	0.00	24.75
Multi-Verse Optimization	95.14	80.67	12.52	21.25	95.44	93.96	1.21	10.56	93.24	50.13	44.50	27.49
Symbiotic Organisms Search Optimization	97.78	96.94	0.57	23.89	**100.00**	99.98	0.10	15.12	97.14	96.23	0.69	106.65
Flower Pollination Optimization	94.44	83.59	14.48	20.55	99.44	97.07	5.23	14.56	93.00	59.52	44.64	10.01
Teaching Learning Based Optimization	97.78	96.08	6.53	23.89	99.44	99.13	0.50	14.56	96.00	83.57	30.87	73.37
Gravitational Search Optimization	92.64	63.33	8.30	18.75	99.00	97.97	2.31	14.12	0.00	0.00	0.00	25.58
Biogeography-Based Optimization	93.06	67.63	7.50	19.17	97.56	95.50	3.40	12.68	90.86	21.65	38.53	30.62
Differential Evolution	67.92	67.92	0.00	−5.97	65.83	64.44	3.78	−19.05	0.00	0.00	0.00	16.41
Particle Swarm Optimization	69.17	69.17	0.00	−4.72	51.67	51.00	0.82	−33.21	0.00	0.00	0.00	35.02
Genetic Algorithm	65.42	65.42	0.00	−8.47	80.75	64.19	9.19	−4.13	0.00	0.00	0.00	14.94
Simulated Annealing	60.83	60.83	0.00	−13.06	66.33	60.05	3.80	−18.55	0.00	0.00	0.00	20.14

Table 6. Performance measures of SVM optimized with meta-heuristic algorithms for Long Method

Performance measures	Accuracy				ROC-AUC				*F*-measure			Time of one
Optimization Algorithms	Best [%]	Avg [%]	Std	Δ_1	Best [%]	Avg [%]	Std	Δ_2	Best [%]	Avg [%]	Std	iteration [s]
Arithmetic Optimization	89.17	72.67	6.09	25.00	98.61	86.32	7.11	32.39	64.33	2.77	12.60	18.61
Jellyfish Search Optimization	80.83	80.42	2.04	16.66	92.50	92.50	0.00	26.28	50.00	46.00	13.56	34.12
Flow Direction Optimization	91.94	90.92	0.30	27.77	97.22	97.22	0.00	31.00	88.38	86.70	0.62	228.81
Student Psychology Based Optimization	87.64	85.27	6.41	23.47	96.00	95.44	1.12	29.78	76.38	54.99	34.30	31.70
Pathfinder Optimization	87.22	87.22	0.00	23.05	97.00	97.00	0.00	30.78	84.88	84.88	0.00	229.40
Sine Cosine Optimization	87.08	87.08	0.00	22.91	97.50	97.50	0.00	31.28	65.00	65.00	0.00	16.19
Jaya Optimization	91.53	79.59	12.42	27.36	98.11	98.11	0.00	31.89	88.48	60.16	41.27	16.09
Crow Search Optimization	69.17	69.17	0.00	5.00	95.50	86.61	9.58	29.28	18.33	0.93	3.68	28.30
Dragonfly Optimization	94.03	82.51	10.45	29.86	97.33	92.87	2.64	31.11	91.14	76.57	22.64	**8.17**
Krill Herd Optimization	66.67	66.67	0.00	2.50	72.83	72.83	0.00	6.61	0.00	0.00	0.00	22.72
Multi-Verse Optimization	91.39	68.72	17.00	27.22	95.13	95.13	0.00	28.91	91.94	36.78	45.04	15.22
Symbiotic Organisms Search Optimization	**96.39**	**95.34**	0.75	**32.22**	**100.00**	**99.36**	0.40	**33.78**	**94.57**	**93.50**	0.53	82.00
Flower Pollination Optimization	90.42	77.42	12.49	26.25	**100.00**	96.90	9.68	**33.78**	87.43	66.45	37.34	9.72
Teaching Learning Based Optimization	94.03	94.03	0.00	29.86	99.33	98.89	0.16	33.11	92.13	77.39	33.77	75.38
Gravitational Search Optimization	71.67	71.67	0.00	7.50	97.22	95.11	4.45	31.00	36.67	1.47	7.19	19.51
Biogeography-Based Optimization	91.81	67.87	11.86	27.64	99.33	97.81	2.38	33.11	90.10	21.57	38.38	32.01
Differential Evolution	64.17	64.17	0.00	0.00	66.00	63.51	5.55	−0.22	0.00	0.00	0.00	10.36
Particle Swarm Optimization	67.92	67.92	0.00	3.75	64.33	62.29	3.10	−1.89	0.00	0.00	0.00	43.82
Genetic Algorithm	69.17	69.17	0.00	5.00	80.11	62.16	8.20	13.89	0.00	0.00	0.00	16.86
Simulated Annealing	70.42	70.42	0.00	6.25	63.50	60.47	0.91	−2.72	0.00	0.00	0.00	20.20

Table 7. Performance measures of k-Nearest Neighbors (k-NN)

	Without optimization			Grid search		
Code Smells	Accuracy	ROC-AUC	F-measure	Accuracy	ROC-AUC	F-measure
Data class	56.11	59.17	29.86	66.81	59.03	21.67
Feature envy	63.33	63.78	43.24	63.89	61.83	50.98
God class	71.81	68.25	49.33	74.17	67.33	51.86
Long method	68.06	65.62	15.67	**77.78**	65	42

God Class maintains prominence with the best ROC-AUC and F-measure values, amounting to 67.33% and 51.86%, respectively. While examining the magnitude of improvements by grid search, it becomes apparent that the uplift in performance measures is not particularly substantial. The most noteworthy enhancements include a 10.69% increase in accuracy for Data Class and 9.72% for Long Method. ROC-AUC always decreased if grid search is applied. A significant boost of 26.33% in F-measure for the Long Method is observed, but F-measure degraded by 8.19% for Data Class. In summary, the findings suggest that grid search, while functional, may not induce significant improvements in performance measures across all scenarios. Results indicate that there is a need for alternative strategy to boost performance.

4.2.1. Data class

The outcomes related to detecting the Data Class are thoughtfully presented in Table 8. Among the array of employed meta-heuristic algorithms, it's evident that **Flower Pollination Optimization** emerges as the most effective. It remarkably attains the maximum accuracy score of 100%, thereby exhibiting a substantial increase of 43.89% (Δ_1) compared to scenarios without optimization. Regarding average accuracy, Pathfinder seizes the top position, achieving a commendable accuracy of 97.62% with a negligible deviation of 0.05. Furthermore, for ROC-AUC values, Pathfinder, Sine Cosine, Jaya, Crow Search, Teaching Learning Based, Multi-Verse, and Flower Pollination Optimization jointly secure the highest value at 100%, reflecting an impressive hike of 40.83% (Δ_2). Pathfinder and Sine Cosine Optimization maintain this elevated performance level by achieving the best average ROC-AUC of 100%, with no deviations observed. Regarding the F-measure metric, Flower Pollination Optimization stands out, boasting a maximum value of 100% and a best average performance score of 96.45%. Finally, from an efficiency perspective, Dragonfly Optimization demonstrates its prowess by completing each iteration in a mere 4.90 seconds, rendering it the fastest among the considered optimization algorithms.

4.2.2. Feature envy

Table 9 presents the comprehensive results for detecting the Feature Envy with optimized k-NN. Remarkably, Symbiotic Organisms Search Optimization emerges as a standout performer, achieving the maximum accuracy and F-measure scores of 96.39% and 92.67%, respectively. Additionally, it boasts the best average accuracy and F-measure, securing impressive values of 96.30% and 92.32%. Notably, Symbiotic Organisms Search Optimization demonstrates a significant increase in accuracy, registering a top hike of 33.06% (Δ_1). Conversely, Multi-Verse Optimization excels in the ROC-AUC metric, showcasing the highest promenade of 35% (Δ_2). The ROC-AUC metric further reveals that Multi-Verse and Jaya Optimization jointly achieve the maximum and best average ROC-AUC scores at

Table 8. Performance measures of k-NN optimized with meta-heuristic algorithms for Data Class

Performance measures	Accuracy				ROC-AUC				F-measure			Time of one
Optimization Algorithms	Best [%]	Avg [%]	Std	Δ_1	Best [%]	Avg [%]	Std	Δ_2	Best [%]	Avg [%]	Std	iteration [s]
Arithmetic Optimization	95.28	92.47	2.16	39.17	99.33	98.87	0.27	40.16	94.29	90.56	2.79	40.59
Jellyfish Search Optimization	92.92	92.47	0.60	36.81	98.57	98.34	0.22	39.40	86.00	85.84	0.37	24.40
Flow Direction Optimization	96.25	96.20	0.24	40.14	99.58	99.37	0.21	40.41	95.14	95.12	0.11	266.47
Student Psychology Based Optimization	96.53	95.43	0.78	40.42	99.17	98.63	0.43	40.00	94.57	92.35	2.52	47.77
Pathfinder Optimization	97.64	**97.62**	0.05	41.53	**100.00**	**100.00**	0.00	**40.83**	95.71	95.28	0.35	186.47
Sine Cosine Optimization	95.42	94.88	0.64	39.31	**100.00**	**100.00**	0.00	**40.83**	94.29	93.77	0.54	14.72
Jaya Optimization	94.31	93.98	0.44	38.20	**100.00**	99.78	0.21	**40.83**	91.24	90.82	0.63	7.06
Crow Search Optimization	96.53	95.73	1.09	40.42	**100.00**	98.62	0.69	**40.83**	96.57	92.21	2.26	24.05
Dragonfly Optimization	96.25	93.67	1.57	40.14	99.58	98.72	1.02	40.41	92.67	89.16	2.70	**4.90**
Krill Herd Optimization	67.92	67.92	0.00	11.81	55.72	55.72	0.00	−3.45	0.00	0.00	0.00	38.23
Multi-Verse Optimization	94.17	93.18	1.23	38.06	**100.00**	99.78	0.18	**40.83**	93.21	92.35	0.90	13.71
Symbiotic Organisms Search Optimization	96.25	95.09	1.57	40.14	97.92	97.64	0.49	38.75	93.00	91.45	1.68	61.71
Flower Pollination Optimization	**100.00**	96.68	6.00	**43.89**	**100.00**	99.97	0.10	**40.83**	**100.00**	**96.45**	4.35	8.95
Teaching Learning Based Optimization	97.64	95.01	1.33	41.53	**100.00**	99.98	0.08	**40.83**	96.57	91.98	2.08	46.49
Gravitational Search Optimization	91.53	89.69	2.41	35.42	98.83	98.31	0.33	39.66	91.63	88.99	1.91	13.00
Biogeography-Based Optimization	95.14	94.37	0.83	39.03	99.44	97.84	0.67	40.27	91.67	89.74	2.06	27.12
Differential Evolution	77.50	70.93	5.07	21.39	84.50	77.03	5.22	25.33	66.19	61.22	5.42	8.33
Particle Swarm Optimization	83.19	80.14	3.69	27.08	87.36	86.55	0.94	28.19	81.35	77.25	4.44	7.31
Genetic Algorithm	79.86	75.53	2.81	23.75	86.39	81.49	2.67	27.22	69.19	57.34	4.50	26.45
Simulated Annealing	78.47	75.73	2.22	22.36	81.94	80.59	0.83	22.77	69.81	59.71	15.37	17.41

Table 9. Performance measures of k-NN optimized with meta-heuristic algorithms for Feature Envy

Performance measures	Accuracy				ROC-AUC				F-measure			Time of one
Optimization algorithms	Best [%]	Avg [%]	Std	Δ_1	Best [%]	Avg [%]	Std	Δ_2	Best [%]	Avg [%]	Std	iteration [s]
Arithmetic Optimization	91.81	86.67	3.51	28.48	96.17	93.72	1.14	32.39	80.67	74.26	9.51	18.81
Jellyfish Search Optimization	88.19	88.05	0.51	24.86	98.33	98.07	0.23	34.55	75.67	73.96	73.96	32.04
Flow Direction Optimization	87.22	86.80	0.99	23.89	95.56	95.39	0.20	31.78	75.24	74.07	1.55	285.53
Student Psychology Based Optimization	88.33	83.99	3.63	25.00	95.83	93.62	1.26	32.05	70.67	56.63	8.68	49.74
Pathfinder Optimization	94.17	94.13	0.19	30.84	98.47	98.03	0.22	34.69	88.17	88.17	0.00	193.51
Sine Cosine Optimization	90.69	90.67	0.05	27.36	96.00	95.49	0.27	32.22	79.57	79.40	0.44	10.53
Jaya Optimization	90.69	90.50	0.45	27.36	98.75	**98.11**	0.30	34.97	79.00	78.59	0.86	5.91
Crow Search Optimization	83.33	80.06	3.45	20.00	94.61	93.24	0.74	30.83	63.00	47.13	14.07	25.11
Dragonfly Optimization	85.83	82.16	2.89	22.50	96.67	95.19	0.98	32.89	67.00	54.67	11.83	**4.27**
Krill Herd Optimization	74.86	70.57	2.03	11.53	78.67	74.72	1.68	14.89	48.72	39.52	6.13	9.61
Multi-Verse Optimization	89.31	88.69	1.01	25.98	**98.78**	97.42	0.55	**35.00**	80.00	78.50	2.52	11.45
Symbiotic Organisms Search Optimization	**96.39**	**96.30**	0.30	**33.06**	98.19	97.52	1.50	34.41	**92.67**	**92.32**	0.61	76.47
Flower Pollination Optimization	94.03	91.57	1.65	30.70	98.50	97.33	1.04	34.72	89.81	74.44	21.30	9.05
Teaching Learning Based Optimization	92.92	91.00	1.90	29.59	97.58	96.77	0.77	33.80	90.38	87.76	2.34	47.05
Gravitational Search Optimization	88.19	86.73	2.56	24.86	96.58	94.98	0.91	32.80	82.48	79.49	7.66	17.54
Biogeography-Based Optimization	90.00	86.07	2.88	26.67	95.67	94.78	0.46	31.89	81.63	77.94	4.76	33.56
Differential Evolution	77.50	74.86	2.74	14.17	83.39	81.83	1.33	19.61	61.50	50.18	9.69	10.79
Particle Swarm Optimization	74.03	73.33	0.62	10.70	81.11	80.78	0.35	17.33	51.43	49.78	0.91	7.21
Genetic Algorithm	75.42	73.42	1.73	12.09	82.97	79.57	2.96	19.19	45.00	28.24	14.53	33.69
Simulated Annealing	74.17	71.87	1.41	10.84	76.81	75.51	0.96	13.03	35.67	21.11	11.39	12.22

98.78% and 98.11%, respectively, with a minimal deviation of 0.30. From an operational efficiency standpoint, Dragonfly Optimization exhibits remarkable swiftness, completing each iteration in a mere 4.27 seconds, thereby asserting itself as the fastest-performing algorithm in the context of this study. In conclusion, optimizing k-NN with **Symbiotic Organisms Search Optimization** is the most effective approach for detecting the Feature Envy.

4.2.3. God class

The results for detecting the God Class are meticulously outlined in Table 10. Notably, Flow Direction Optimization delivers outstanding performance, clinching the maximum accuracy, best average accuracy, and the highest hike in accuracy, reaching impressive scores of 97.64%, 97.64% (with zero deviations), and a remarkable 25.83% hike (Δ_1). Moreover, the ROC-AUC metric showcases exceptional results, with a maximum ROC-AUC score of 100% and the most significant hike, reaching 31.75% (Δ_2). These outstanding achievements are credited to Arithmetic, Jellyfish Search, Flow Direction, Pathfinder, Jaya, Crow Search, Multi-Verse, Symbiotic Organisms Search, Flower Pollination, Teaching Learning Based, Gravitational Search, and Biogeography-Based Optimization. In addition, Jellyfish Search, Flow Direction, Pathfinder, Multi-Verse, Teaching Learning Based, and Biogeography-Based Optimization collectively secure a best average ROC-AUC of 100%, accompanied by zero deviations. Regarding the F-measure, Flow Direction Optimization stands out with a maximum and average value of 96%, with no deviations. In terms of operational efficiency, Jaya Optimization demonstrates impressive speed, completing each iteration in 5.08 seconds. Additionally, it's worth noting that both Differential Evolution and Simulated Annealing achieve perfect scores of 100% for both best and average ROC-AUC values. In conclusion, utilizing k-NN in conjunction with **Flow Direction Optimization** is the most effective approach to detecting the God Class despite a slightly slower execution time.

4.2.4. Long method

Table 11 comprehensively presents the outcomes concerning detecting the Long Method code. Notably, Biogeography-Based Optimization stands out with the maximum accuracy, the best average accuracy, and the most significant accuracy hike, attaining remarkable scores of 96.39%, 96.39%, and 28.33% hike (Δ_1), respectively. Moreover, the ROC-AUC metric showcases exceptional results, with a maximum ROC-AUC score of 100% and the most substantial hike, reaching 34.38% (Δ_2). These exceptional achievements are attributed to Jellyfish Search, Student Psychology Based, Sine Cosine, Jaya, Crow Search, Multi-Verse, Symbiotic Organisms Search, Flower Pollination, Teaching Learning Based, Gravitational Search, and Biogeography-Based Optimization. Furthermore, Jellyfish Search, Student Psychology Based, and Symbiotic Organisms Search Optimization collectively secure the best average ROC-AUC of 100%. Regarding the F-measure, Biogeography-Based Optimization achieves the maximum value of 94%, while Multi-Verse Optimization secures the best average F-measure of 88.44%. Dragonfly Optimization is the fastest, completing each iteration in 5.84 seconds. It's worth highlighting that the other elementary algorithms also deliver commendable performance in this context. In summary, when it comes to detecting the Long Method, using k-NN in conjunction with **Biogeography-Based** or

Table 10. Performance measures of k-NN optimized with meta-heuristic algorithms for God Class

Performance measures	Accuracy				ROC-AUC				F-measure			Time of one
Optimization Algorithms	Best [%]	Avg [%]	Std	Δ_1	Best [%]	Avg [%]	Std	Δ_2	Best [%]	Avg [%]	Std	iteration [s]
Arithmetic Optimization	93.89	88.99	5.47	22.08	**100.00**	99.84	0.27	**31.75**	86.00	70.50	15.13	33.46
Jellyfish Search Optimization	93.89	93.84	0.22	22.08	**100.00**	**100.00**	0.00	**31.75**	87.67	87.19	1.30	29.67
Flow Direction Optimization	**97.64**	**97.64**	0.00	**25.83**	**100.00**	**100.00**	0.00	**31.75**	**96.00**	**96.00**	0.00	240.65
Student Psychology Based Optimization	94.03	92.92	1.24	22.22	98.61	98.51	0.16	30.36	87.33	85.03	5.18	47.80
Pathfinder Optimization	92.92	92.92	0.00	21.11	**100.00**	**100.00**	0.00	**31.75**	91.43	90.81	0.60	185.17
Sine Cosine Optimization	90.69	90.69	0.00	18.88	99.67	99.27	0.19	31.42	82.67	82.67	0.00	12.47
Jaya Optimization	96.39	94.99	0.79	24.58	**100.00**	99.73	0.29	**31.75**	90.00	89.27	1.19	**5.08**
Crow Search Optimization	91.53	87.58	3.61	19.72	**100.00**	99.76	0.31	**31.75**	81.00	73.71	10.30	17.61
Dragonfly Optimization	92.92	85.54	6.06	21.11	99.44	98.68	0.45	31.19	82.67	60.81	22.08	5.57
Krill Herd Optimization	72.36	67.23	2.60	0.55	85.23	78.59	3.37	16.98	51.79	40.31	7.24	10.90
Multi-Verse Optimization	94.03	93.53	0.56	22.22	**100.00**	**100.00**	0.00	**31.75**	91.24	90.25	1.13	13.27
Symbiotic Organisms Search Optimization	91.81	91.57	0.43	20.00	**100.00**	99.99	0.05	**31.75**	82.33	79.56	2.51	62.28
Flower Pollination Optimization	93.89	88.12	5.63	22.08	**100.00**	99.64	0.33	**31.75**	89.00	81.72	7.94	10.41
Teaching Learn ing Based Optimization	88.06	86.82	1.61	16.25	**100.00**	**100.00**	0.00	**31.75**	80.71	74.73	6.24	46.68
Gravitational Search Optimization	95.42	93.20	1.62	23.61	**100.00**	99.93	0.16	**31.75**	93.71	90.12	4.46	17.06
Biogeography-Based Optimization	92.64	87.37	6.18	20.83	**100.00**	**100.00**	0.00	**31.75**	79.33	61.85	19.78	66.86
Differential Evolution	96.39	95.79	0.94	24.58	100.00	100.00	0.00	31.75	92.67	90.75	4.02	10.06
Particle Swarm Optimization	96.53	96.53	0.00	24.72	100.00	99.96	0.11	31.75	92.67	92.67	0.00	6.88
Genetic Algorithm	96.39	94.17	0.72	24.58	100.00	99.77	0.26	31.75	93.14	90.45	2.29	33.48
Simulated Annealing	96.39	93.43	3.92	24.58	100.00	99.33	1.33	31.75	93.00	87.15	10.04	15.01

Table 11. Performance measures of k-NN optimized with meta-heuristic algorithms for Long Method

Performance measures	Accuracy				ROC-AUC				F-measure			Time of one
Optimization Algorithms	Best [%]	Avg [%]	Std	Δ_1	Best [%]	Avg [%]	Std	Δ_2	Best [%]	Avg [%]	Std	iteration [s]
Arithmetic Optimization	88.33	80.65	4.68	20.27	99.67	99.25	0.25	34.05	81.81	69.82	10.86	15.23
Jellyfish Search Optimization	90.69	90.36	0.85	22.63	**100.00**	**100.00**	0.00	**34.38**	84.24	83.62	1.29	30.59
Flow Direction Optimization	84.44	84.44	0.00	16.38	96.94	96.72	0.27	31.32	63.00	63.00	0.00	272.84
Student Psychology Based Optimization	90.69	89.53	2.63	22.63	**100.00**	**100.00**	0.00	**34.38**	84.90	80.72	5.87	31.00
Pathfinder Optimization	84.31	83.89	0.66	16.25	99.67	99.39	0.36	34.05	68.67	68.67	0.00	203.61
Sine Cosine Optimization	89.44	89.14	0.53	21.38	**100.00**	99.92	0.13	**34.38**	79.00	78.43	1.02	28.44
Jaya Optimization	84.86	82.86	1.05	16.80	**100.00**	99.26	0.84	**34.38**	76.71	74.30	1.86	14.56
Crow Search Optimization	87.78	81.33	4.72	19.72	**100.00**	99.29	0.47	**34.38**	70.33	58.02	11.43	27.80
Dragonfly Optimization	88.06	83.47	2.12	20.00	99.29	98.12	0.68	33.67	50.00	18.40	13.60	**5.84**
Krill Herd Optimization	89.17	80.89	5.65	21.11	97.50	92.91	2.43	31.88	80.57	66.27	10.47	7.82
Multi-Verse Optimization	94.17	**93.88**	0.42	26.11	**100.00**	99.73	0.24	**34.38**	89.00	**88.44**	0.59	31.89
Symbiotic Organisms Search Optimization	90.42	89.38	0.93	22.36	**100.00**	**100.00**	0.00	**34.38**	71.67	68.65	2.76	69.10
Flower Pollination Optimization	88.89	83.81	6.06	20.83	**100.00**	99.38	0.30	**34.38**	76.57	67.91	10.68	28.75
Teaching Learning Based Optimization	91.67	90.16	1.74	23.61	**100.00**	99.98	0.08	**34.38**	86.24	83.22	4.89	66.39
Gravitational Search Optimization	89.31	87.77	1.31	21.25	**100.00**	99.82	0.25	**34.38**	82.24	75.16	7.31	35.45
Biogeography-Based Optimization	**96.39**	90.43	6.41	**28.33**	**100.00**	99.68	0.20	**34.38**	**94.00**	83.42	12.64	65.35
Differential Evolution	86.67	84.17	1.57	18.61	96.33	94.94	0.91	30.71	69.00	61.91	13.58	11.08
Particle Swarm Optimization	87.08	86.06	0.30	19.02	95.00	93.77	2.21	29.38	69.67	63.29	3.58	22.47
Genetic Algorithm	86.94	80.01	6.70	18.88	94.03	91.60	1.33	28.41	76.00	61.67	19.06	47.63
Simulated Annealing	94.03	93.21	0.98	25.97	98.11	98.04	0.24	32.49	92.07	91.03	1.51	31.66

Multi-Verse Optimization emerges as the most effective approach, offering excellent accuracy and ROC-AUC outcomes.

5. Discussion

The following section addresses the research questions and discusses the important findings of the research study.

5.1. Does using meta-heuristic algorithms for optimizing machine learning classifiers boost their performance to detect code smell in complex software systems?

The described experiment has been executed to address RQ 1, yielding noteworthy advancements in performance metrics. In the case of Support Vector Machine (SVM), the highest accuracy is 98.75%, achieved by the Sine Cosine Optimization algorithm when applied to detect God Class. The Symbiotic Organisms Search elevates the accuracy by 32.22% when deployed for Long Method. Regarding ROC-AUC metrics, a flawless 100% is reached by Symbiotic Organisms Search across multiple code smell detection, specifically for Data Class, God Class, and Long Method.

In the case of God Class, the ROC-AUC value of 100% is also secured by Student Psychology Based, Sine Cosine, and Jaya Optimization methods. Similarly, Flower Pollination Optimization achieves an impeccable 100% ROC-AUC for Long Method. The average ROC-AUC stands at 100% and is simultaneously attained by Student Psychology Based, Sine Cosine, and Jaya Optimization for God Class. For detecting Data Class, Symbiotic Organisms Search orchestrates a 45.11% hike in ROC-AUC value.

Turning our focus to F-measure, the Sine Cosine Optimization achieved 98.57% to combat God Class. Dragonfly Optimization is acknowledged as the fastest in algorithmic velocity, while Pathfinder Optimization is the slowest. Conclusively, the apex of optimization is occupied by the **Sine Cosine Algorithm**, demonstrably exemplifying its pre-eminence by securing the highest scores, both in terms of maximum and average values, across a spectrum of performance metrics.

For k-Nearest Neighbors (k-NN), a perfect 100% accuracy and 43.89% surge in accuracy is recorded, executed by the Flower Pollination Optimization method when applied to detect Data Class. It also scores a perfect 100% F-measure and the best average F-measure at 96.45%. When applied to detect God Class, the highest average accuracy is 97.64% attained by the Flow Direction Optimization algorithm.

For ROC-AUC metrics, a flawless 100% is not a solitary accomplishment but a shared distinction among several optimization methodologies. Specifically, optimizers for Data Class include Pathfinder, Sine Cosine, Jaya, Crow Search, Multi-Verse, Flower Pollination, and Teaching Learning Based Optimization, concurrently ascending to this pinnacle. Each eminent algorithm also accomplishes a 40.83% ROC-AUC increase. Similarly, the God Class bears witness to the Arithmetic, Jellyfish Search, Flow Direction, Pathfinder, Jaya, Crow Search, Multi-Verse, Symbiotic Organisms Search, Flower Pollination, Teaching Learning Based, Gravitational Search, and Biogeography-Based Optimization, all attaining a flawless 100% ROC-AUC value. The Long Method equally experiences perfection in ROC-AUC, with Jellyfish Search, Student Psychology Based, Sine Cosine, Jaya, Crow Search, Multi-Verse, Symbiotic Organisms Search, Flower Pollination, Teaching Learning

Based, Gravitational Search, and Biogeography-Based Optimization, all registering a 100% ROC-AUC.

The average ROC-AUC yields a harmonious 100% outcome with no deviations for several scenarios: Pathfinder and Sine Cosine Optimization for Data Class, Jellyfish Search, Flow Direction, Pathfinder, Multi-Verse, Teaching Learning Based, and Biogeography-Based Optimization for God Class, and Jellyfish Search, Student Psychology Based, and Symbiotic Organisms Search Optimization for Long Method. Regarding the computational time, Dragonfly Optimization is the fastest, while the Flow Direction Optimization method is the slowest in its computational stride. The paramount optimizer, defined by maximum and average performance measures, emerges as the **Flower Pollination Optimization**, underscoring its dominance in k-NN optimization.

Summary of RQ 1. Employing swarm-based techniques to optimize the hyperparameter values of machine learning classifiers is definitely a beneficial process. It not only improves the performance of a classifier but eliminates the need for an expert, automating the code smell detection process.

5.2. How significant is the impact of optimization of machine learning algorithms with meta-heuristic techniques on its overall performance?

To answer RQ 2, we have conducted statistical tests on experiment results to evaluate the impact of optimization. The Wilcoxon signed-rank test is a non-parametric statistical test used to assess whether the distribution of paired differences between two related groups is symmetric about zero [96]. Experimentation data do not follow a normal distribution, have paired observations, and data can be ranked. Therefore, the Wilcoxon signed-rank test is the best hypothesis statistical test to measure the impact of employing meta-heuristic algorithms for optimizing machine learning algorithms.

To perform the test, a null hypothesis (H0) is set up as – the median difference between paired observations is zero (no difference) and the alternative hypothesis (H1) as the median difference between paired observations is not zero. Data is gathered for the paired observations we want to compare, and the differences between paired observations are calculated. The absolute values of the differences are ranked, and the test statistic (W) using the ranked differences is calculated. For n pairs, the degree of freedom is $n - 1$. The test statistic (p-value) to the critical value from the Wilcoxon signed-rank distribution table is compared. If the p-value is less than the chosen significance level, reject the null hypothesis, indicating a significant difference. If the p-value is greater than the significance level, fail to reject the null hypothesis. The test statistic, degrees of freedom, p-value, and decision regarding the null hypothesis are reported [97].

Table 12–15 results depict the value of z, p, and r from the Wilcoxon signed rank sum test. Values before optimization are paired with best and average values acquired after optimization. The degree of freedom for this test is 15. The confidence level is 95%, and the significance level is 0.05. The null hypothesis is rejected if the p-value is less than 0.05, implying the difference is significant. Based on the results, it can be seen that the p-values for all five performance measures are below 0.05, indicating a significant difference between performance measures before and after optimization. r denotes effect size depicting the magnitude of difference and can be calculated as $\frac{z}{\sqrt{n}}$, where n is the number of paired

Table 12. Wilcoxon Signed Rank Sum test results for Best Values attained by SVM

Performance measures	Accuracy			*F*-measure			ROC-AUC		
Code smell	z	p	r	z	p	r	z	p	r
Data class	3.5180	.000435	0.88	3.5168	.000437	0.88	3.5174	.000436	0.88
Feature envy	3.5168	.000437	0.88	3.4651	.000530	0.87	3.3616	.000775	0.84
God class	3.3099	.000933	0.83	2.6983	.006969	0.67	3.2966	.000979	0.82
Long method	3.5168	.000437	0.88	3.5180	.000435	0.88	3.4128	.000643	0.85

Table 13. Wilcoxon Signed Rank Sum test tesults for Average Values attained by SVM

Performance measures	Accuracy			*F*-measure			ROC-AUC		
Code smell	z	p	r	z	p	r	z	p	r
Data class	3.5162	.000438	0.88	3.5162	.000438	0.88	3.5168	.000437	0.88
Feature envy	3.3611	.000776	0.84	3.4133	.000642	0.85	2.7923	.005234	0.70
God class	2.3786	.017378	0.59	2.6389	.008317	0.66	3.2958	.000982	0.82
Long method	3.5162	.000438	0.88	3.5162	.000438	0.88	2.7923	.005234	0.70

Table 14. Wilcoxon Signed Rank Sum test results for Best Values attained by k-NN

Performance measures	Accuracy			*F*-measure			ROC-AUC		
Code smell	z	p	r	z	p	r	z	p	r
Data class	3.5197	.000432	0.88	3.4980	.000469	0.87	3.4656	.000529	0.87
Feature envy	3.5174	.000436	0.88	3.5162	.000438	0.88	3.5162	.000438	0.88
God class	3.5197	.000432	0.88	3.6973	.000218	0.92	3.5168	.000437	0.88
Long method	3.5168	.000437	0.88	3.6537	.000258	0.91	3.5162	.000438	0.88

Table 15. Wilcoxon Signed Rank Sum test results for Average Values attained by k-NN

Performance measures	Accuracy			*F*-measure			ROC-AUC		
Code smell	z	p	r	z	p	r	z	p	r
Data class	3.5168	.000437	0.88	3.4656	.000529	0.87	3.4651	.000530	0.87
Feature envy	3.5162	.000438	0.88	3.5162	.000438	0.88	3.4645	.000531	0.87
God class	3.4651	.000530	0.87	3.5369	.000405	0.88	3.4645	.000531	0.87
Long method	3.5162	.000438	0.88	3.5185	.000434	0.88	3.5162	.000438	0.88

observations. The effect is considered high if r is greater than 0.5 and 0.8 is recorded r value in the experimentation, yielding promising results.

Summary of RQ 2. Optimizing machine learning algorithms with swarm-intelligent algorithms significantly impacts their performance.

5.3. Given the meta-heuristic algorithms, which yields the best performance in optimizing classifiers to detect code smell and why?

To address RQ 3, the experiment's outcomes are examined and compared to determine the most effective meta-heuristic techniques for optimizing machine learning algorithms for code smell detection. It is important to acknowledge that the "No-Free Lunch" theorem has significantly influenced the landscape of optimization algorithms, driving continuous innovations over the years. This theorem underscores that no single algorithm universally excels in every problem domain. Instead, their efficacy varies, with each demonstrating superior performance in specific problem statements [98]. Implementation remains the most effective means of identifying the optimal technique for a given problem.

The investigation involved systematically applying sixteen meta-heuristic algorithms for hyperparameter optimization on two distinct machine learning algorithms, enhancing their performance metrics. A comprehensive evaluation is conducted across three scenarios: instances without optimization, cases employing grid search, and evaluations utilizing meta-heuristic algorithms. Performance metrics are thoughtfully juxtaposed with other algorithms, ensuring an accurate and effective comparison. These assessments are supplemented by comparisons with foundational algorithms, namely Genetic Algorithm, Differential Evolution, Particle Swarm Optimization, and Simulated Annealing. The empirical results affirm that the foundational algorithms, while competent, do not outshine the implemented optimizers across the board. Instead, they exhibit comparable proficiency in a few cases. The comprehensive evaluation of their performance measures, in conjunction with other algorithms, is methodically documented after the respective tables. Table 16 highlights the highest-performing optimization algorithms for each case based on experimentation.

Table 16. The best performing optimization algorithm for each code smell

Code Smell	SVM	k-NN
Data class	Symbiotic Organisms Search Optimization	Flower Pollination Optimization
Feature envy	Dragonfly Optimization	Symbiotic Organisms Search Optimization
God class	Sine Cosine Optimization	Flow Direction Optimization
Long method	Symbiotic Organisms Search Optimization	Biogeography-Based and Multi-Verse Optimization

Finding the most optimized value for hyperparameters of machine learning algorithms is in the category of non-separable, constrained, and multimodal problems. Non-separable problems refer to scenarios where the relationships and dependencies within the data are too intricate to be accurately represented by simple linear decision boundaries [99]. In classification tasks, linear separability implies that classes can be perfectly distinguished by a straight line, plane, or hyperplane, but non-separable problems defy such simplicity. Dealing with non-separable data requires complex decision boundaries, often necessitating the application of nonlinear models like kernelized support vector machines. Specialized algorithms like meta-heuristics or evolutionary approaches may be needed to navigate such landscapes.

Constrained problems refer to scenarios where the solution space of a problem is subject to certain conditions or limitations [100]. These constraints restrict the set of feasible solutions and play a critical role in shaping the optimization landscape. Optimization algorithms designed for constrained problems must navigate the complex interplay between the objective function and the imposed constraints. Classical optimization methods, like Lagrange multipliers and penalty methods, are often employed to handle equality and inequality constraints.

Multimodal problems refer to scenarios where the objective function or fitness landscape has multiple distinct optimal solutions, known as modes [101]. Each mode represents a set of parameter values that yield an optimal or near-optimal solution to the problem where the algorithm can converge. The presence of multiple modes introduces challenges because traditional optimization methods, which aim to find a single global optimum, may struggle to explore and exploit the diverse modes. Handling multimodal problems requires specialized optimization techniques designed to explore and exploit multiple modes efficiently.

Meta-heuristic algorithms are easy to implement and do not require much domain-specific knowledge. They can optimize both continuous, discrete problems, and multiple objective problems. The performance of a meta-heuristic algorithm hinges upon some of the pivotal factors. The first factor is the precise configuration of parameters, encompassing critical attributes such as the optimal count of search agents, the judicious establishment of the discovery rate, the velocity of the fitness function, etc. Furthermore, a paramount significance revolves around the delicate equilibrium between exploration and exploitation rates. Though they can find the global optima for complex, nonlinear, and non-convex functions, they are prone to get trapped in local optima if the population size is small and the search space is vast. Augmenting this complexity, randomness into the search process emerges as a potent mechanism. By introducing controlled stochasticity, these optimizers foster heightened performance and enhanced exploration of solution spaces, ultimately yielding superior results [102]. The **Symbiotic Organisms Search Optimization, Teaching Learning Based Optimization, and Sine Cosine Optimization** emerge as stellar exemplars of detecting code smells, adeptly navigating the intricate terrain of algorithmic design. They orchestrate a harmonious symphony of parameter tuning, dynamic mode switching, and controlled randomness infusion, culminating in attaining superlative outcomes. Conversely, Krill Herd Optimization is the least effective algorithm for code smell detection.

Summary of RQ 3. The no-free Lunch theorem implies that there is no one-size-fits-all solution; what works best for one optimization problem might not work for another problem. So, the best way to find the most optimal techniques is to implement them and compare their results. Table 16 summarizes the list of best-performing optimization techniques for each case. They performed better because they balanced exploration and exploitation well, avoided early convergence, introduced appropriate randomness, and discovered global optimum solutions required to conquer non-separable, constrained, and multi-modal problems.

5.4. How does our approach perform compared to existing machine learning based techniques?

To answer RQ 4, we have compared our work with Fontana et al. [46]. This is the most extensive study that detects code smells using machine learning and employs the same datasets, allowing for a fair comparison. They created balanced datasets to detect the four most common and perilous code smells. They applied 32 variations of machine learning classifiers, including their boosted versions, for detection. It included pruned, unpruned, and reduced error pruning techniques of J48, a C4.5 decision tree. JRip, Random Forest, Naive Bayes, and SMO with RBF and Polynomial kernel were also included. With that, C and ν SVM were implemented with Linear, Polynomial, RBF, and Sigmoid kernel settings. Implementation was done in Weka, and machine learning classifiers were treated as black-box implementations. No pre-processing or feature selection technique was used except in the case of SVM, where standardization and normalization were done. They employed 10-fold cross-validation techniques and reported average values. Tree-based algorithms like J48 and random forest performed best, whereas SVMs were the worst performers.

For Data Class, C-SVM with RBF has an accuracy of 96%, F-measure of 97.01%, and ROC-AUC of 99.15%. Symbiotic Organisms Search Optimization is the best performer in detecting Data Class with SVM and outperformed accuracy and AUC delivered by Fontana et al., reported accuracy is 97.64%, *F*-measure is 96%, and ROC-AUC is 100%. In the case of Feature Envy, results projected by Fontana et al. were far less superior. The dragonfly optimizer performed best with 96.25% accuracy, 94.29% *F*-measure, and 99.33% ROC-AUC, whereas Fontana et al. yielded 94.14% accuracy, 95.62% *F*-measure, and 98.02% ROC-AUC. For God Class, the Sine Cosine algorithm outperformed all the above-mentioned meta-heuristic algorithms and results of Fontana et al. Accuracy, *F*-measure, and ROC-AUC achieved for God Class in the case of Fontana et al. are 95.76%, 96.87%, and 99.24%, respectively, whereas optimized SVM achieved 98.75% accuracy, 98.57% *F*-measure, and 100% ROC-AUC, all on the higher side. Symbiotic Organisms Search Optimization obtained 96.39% accuracy and 100% ROC-AUC for the Long Method, which is higher compared to the performance measured attained by Fontana et al., i.e., 96.38% accuracy and 99.15% ROC-AUC. One exception is the *F*-measure, which is 94.57% for optimized SVM and 97.22% for the unoptimized version.

Summary of RQ4. Unlike *F*-measure in two out of four cases, utilizing swarm-based algorithms for optimizing SVM is a better option as it delivers elevated performance.

6. Threats to validity

In this section, threats to validity are discussed that might arise concerns and how they are mitigated.

6.1. Threats to internal validity

The assessment of metrics within the datasets [46] is conducted using a proprietary tool known as Design Features and Metrics for Java (DFMC4J). This tool operates by parsing Java code through the Eclipse JDT Library; however, it is important to note that the accuracy of its calculations has not been externally validated, potentially introducing imprecision in metric computations for source code elements. Moreover, the identification of code smell candidates is carried out manually by students rather than seasoned professionals, thereby introducing an inherent margin of error. To mitigate this concern, a comprehensive training program was administered to the students, and the final decisions were made following meticulous deliberation. In addition to this, code smell detection tools like iPlasma, PMD, and Fluid tools were also enlisted to corroborate the presence of code smell instances.

6.2. Threats to external validity

The datasets were meticulously crafted from a collection of 74 open-source Java systems sourced from the Qualitus Corpus [67]. Nonetheless, it's imperative to acknowledge that open-source software might not encompass the entirety of conceivable scenarios, potentially limiting the generalization of findings to industrial contexts. Extending our investigation to encompass industrial, commercial, and private projects is a future endeavor. These systems employ older Java versions and don't include emerging new Java language constructs [103]. The systems included are from 2003-2011, which might not represent the current scenario. These issues can be addressed in future work by employing datasets that include the latest Java constructs, industrial projects, more code smells, severity prospects, etc.

It is important to underscore that this empirical study focuses solely on datasets originating from Java source code, and its findings may not seamlessly translate to other programming languages given the distinct nature of metric values and design paradigms across languages. As part of our ongoing research, we aim to explore additional programming languages to augment the breadth of our insights. Furthermore, while Fowler [6] delineates twenty-two distinct code smells, this study delves into the analysis of only four, a limited subset for generalization. Future investigations could encompass the remaining smells to yield more comprehensive and conclusive outcomes. Similarly, the utilization of merely two classifiers in this study warrants consideration, as the implications derived may not be universally applicable.

6.3. Threats to conclusion validity

The computation of F-measure by certain machine learning algorithms faced limitations, impacting the derived conclusions concerning F-measure values. This issue is of considerable significance, warranting both immediate attention and subsequent in-depth investigation. While maintaining nearly identical parameters, including population size and generations, across various meta-heuristic algorithms facilitates fair comparisons, it's worth noting that these parameters might inadvertently affect certain algorithms due to their diverse search agent requirements. Additionally, the uniformity of stopping criteria is 50 iterations for each algorithm might not ensure fairness, given the inherent variability in convergence rates among different algorithms.

7. Conclusions and future work

Our investigation delves into the merits of diverse meta-heuristic algorithms as tools for optimizing supervised machine learning techniques. Additionally, we have conducted a comparative analysis of results between machine learning classifiers, both pre and post optimization. The findings from our study are summarized as follows:

1. The top-performing meta-heuristic algorithm is Symbiotic Organisms Search Optimization. Conversely, Krill Herd Optimization exhibited the lowest performance in the context of code smell detection.
2. In the case of Support Vector Machine, the apex metrics include an accuracy rate of 98.75%, a perfect ROC-AUC score of 100%, and an F-measure of 98.57%. The maximum improvement in accuracy and ROC-AUC observed is 32.22% and 45.11%, respectively.
3. The best k-Nearest Neighbor, outcomes are marked by a flawless accuracy rate, ROC-AUC, and F-measure value of 100%. The accuracy and ROC-AUC surged by 43.89% and 40.83%, respectively, through applying optimization algorithms.
4. SVM showcased its optimum performance when coupled with Sine Cosine Optimization, whereas k-NN exhibited superior results when joined with Flower Pollination Optimization.
5. A Rigorous statistical test underscores the profound impact of meta-heuristic algorithms in fine-tuning hyperparameters of machine learning algorithms, thereby enhancing their overall performance.
6. SVM excels in detecting God Class and k-NN masters in identifying Data Class instances, all achieved through optimizing machine learning classifiers via meta-heuristic algorithms.

Here are the prospective avenues for further research stemming from this study:

1. Future work could utilize improved versions of meta-heuristic algorithms, characterized by improved convergence speed, exploration capabilities, and diminished sensitivity to hyperparameters.
2. Exploring multi-objective, binary, hybrid, chaotic and alternative variants of meta-heuristic techniques hold the premise for achieving heightened efficiency, adaptability and flexibility in optimization processes.
3. An extensive array of over two hundred meta-heuristic algorithms exist, whereas our study has selectively implemented specific types. Future research endeavors could extend to comparative assessments with a broader spectrum of optimization algorithms.
4. The implementation and evaluation of novel optimization algorithms, including but not limited to Central Force Optimization, Vortex Search Algorithm, Thermal Exchange Optimization, and Artificial Electric Field Algorithm, offer intriguing prospects for further inquiry.
5. Expanding the scope to optimize various other machine learning classifiers such as Random Forest, Decision Tree, JRip, and Naive Bayes, among others, holds potential for diversifying the application domains of these techniques.
6. The drive to optimize machine learning classifiers for detecting various other code smells or anti-patterns presents an engaging research avenue.
7. Investigating code smells in programming languages beyond Java constitutes a compelling direction for future research, broadening the applicability of the findings.
8. The exploration of feature engineering and selection methodologies utilizing meta-heuristic algorithms emerges as an avenue with the potential to augment the performance of machine learning classifiers.

References

[1] I. Ozkaya, "The next frontier in software development: AI-augmented software development processes," *IEEE Software*, Vol. 40, No. 4, 2023, pp. 4–9.

[2] H.J. Christanto and Y.A. Singgalen, "Analysis and design of student guidance information system through software development life cycle (SDLC) and waterfall model," *Journal of Information Systems and Informatics*, Vol. 5, No. 1, 2023, pp. 259–270.

[3] M. Almashhadani, A. Mishra, A. Yazici, and M. Younas, "Challenges in agile software maintenance for local and global development: An empirical assessment," *Information*, Vol. 14, No. 5, 2023, p. 261.

[4] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta et al., "When and why your code starts to smell bad," in *International Conference on Software Engineering*, Vol. 1. IEEE, 2015, pp. 403–414.

[5] S.M. Olbrich, D.S. Cruzes, and D.I. Sjøberg, "Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems," in *International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[6] M. Fowler, *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 2018.

[7] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 214–223.

[8] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann et al., "The WEKA data mining software: An update," *ACM SIGKDD Explorations Newsletter*, Vol. 11, No. 1, 2009, pp. 10–18.

[9] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th International Conference on Software Maintenance*. IEEE, 2004, pp. 350–359.

[10] G. Travassos, F. Shull, M. Fredericks, and V.R. Basili, “Detecting defects in object-oriented designs: using reading techniques to increase software quality,” *ACM Sigplan Notices*, Vol. 34, No. 10, 1999, pp. 47–56.

[11] G. Ganea, I. Verebi, and R. Marinescu, “Continuous quality assessment with inCode,” *Science of Computer Programming*, Vol. 134, 2017, pp. 19–36.

[12] H. Li and S. Thompson, “Let’s make refactoring tools user-extensible!” in *Proceedings of the Fifth Workshop on Refactoring Tools*, 2012, pp. 32–39.

[13] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, “A review-based comparative study of bad smell detection tools,” in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, 2016, pp. 1–12.

[14] M.V. Mäntylä and C. Lassenius, “Subjective evaluation of software evolvability using code smells: An empirical study,” *Empirical Software Engineering*, Vol. 11, No. 3, 2006, pp. 395–431.

[15] E. Alpaydin, *Introduction to machine learning*. MIT Press, 2020.

[16] S. Jain and A. Saha, “Improving performance by genetically optimizing support vector machine to detect code smells,” in *Proceedings of the International Conference on Smart Data Intelligence (ICSMDI 2021)*, 2021.

[17] G.A. Pradipta, R. Wardoyo, A. Musdholifah, I.N.H. Sanjaya, and M. Ismail, “SMOTE for handling imbalanced data problem: A review,” in *Sixth International Conference on Informatics and Computing (ICIC)*. IEEE, 2021, pp. 1–8.

[18] H. Gupta, S. Misra, L. Kumar, and N. Murthy, “An empirical study to investigate data sampling techniques for improving code-smell prediction using imbalanced data,” in *International Conference on Information and Communication Technology and Applications*. Springer, 2020, pp. 220–233.

[19] S. Jain and A. Saha, “Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection,” *Science of Computer Programming*, Vol. 212, 2021, p. 102713.

[20] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, Vol. 1. New York: Springer Series in Statistics, 2001.

[21] I. Syarif, A. Prugel-Bennett, and G. Wills, “SVM parameter optimization using grid search and genetic algorithm to improve classification performance,” *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, Vol. 14, No. 4, 2016, pp. 1502–1509.

[22] D.M. Belete and M.D. Huchaiah, “Grid search in hyperparameter optimization of machine learning models for prediction of hiv/aids test results,” *International Journal of Computers and Applications*, Vol. 44, No. 9, 2022, pp. 875–886.

[23] M. Karimi-Mamaghan, M. Mohammadi, P. Meyer, A.M. Karimi-Mamaghan, and E.G. Talbi, “Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art,” *European Journal of Operational Research*, Vol. 296, No. 2, 2022, pp. 393–422.

[24] V. Chandra S.S. and H.S. Anand, “Nature inspired meta heuristic algorithms for optimization problems,” *Computing*, Vol. 104, No. 2, 2022, pp. 251–269.

[25] E. Osaba, E. Villar-Rodriguez, J. Del Ser, A.J. Nebro, D. Molina et al., “A tutorial on the design, experimentation and application of metaheuristic algorithms to real-world optimization problems,” *Swarm and Evolutionary Computation*, Vol. 64, 2021, p. 100888.

[26] J. McDermott, “When and why metaheuristics researchers can ignore “No Free Lunch” theorems,” *SN Computer Science*, Vol. 1, No. 1, 2020, p. 60.

[27] V.W. Porto, “Evolutionary programming,” in *Evolutionary Computation*. CRC Press, 2018, pp. 127–140.

[28] A. Lambora, K. Gupta, and K. Chopra, “Genetic algorithm – A literature review,” in *International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. IEEE, 2019, pp. 380–384.

[29] M. Pant, H. Zaheer, L. Garcia-Hernandez, A. Abraham et al., “Differential Evolution: A review of more than two decades of research,” *Engineering Applications of Artificial Intelligence*, Vol. 90, 2020, p. 103479.

[30] M.G.P. de Lacerda, L.F. de Araujo Pessoa, F.B. de Lima Neto, T.B. Ludermir, and H. Kuchen, “A systematic literature review on general parameter control for evolutionary and swarm-based algorithms,” *Swarm and Evolutionary Computation*, Vol. 60, 2021, p. 100777.

[31] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of ICNN’95 International Conference on Neural Networks*, Vol. 4. IEEE, 1995, pp. 1942–1948.

[32] M. Dorigo, M. Birattari, and T. Stutzle, “Ant colony optimization,” *IEEE Computational Intelligence Magazine*, Vol. 1, No. 4, 2006, pp. 28–39.

[33] S. Chattopadhyay, A. Marik, and R. Pramanik, “A brief overview of physics-inspired metaheuristic optimization techniques,” *arXiv preprint arXiv:2201.12810*, 2022.

[34] K. Karthikeyan and P. Dhal, “Multi verse optimization (MVO) technique based voltage stability analysis through continuation power flow in IEEE 57 bus,” *Energy Procedia*, Vol. 117, 2017, pp. 583–591.

[35] Z. Wei, C. Huang, X. Wang, T. Han, and Y. Li, “Nuclear reaction optimization: A novel and powerful physics-based algorithm for global optimization,” *IEEE Access*, Vol. 7, 2019, pp. 66 084–66 109.

[36] S. Cheng, Q. Qin, J. Chen, and Y. Shi, “Brain storm optimization algorithm: a review,” *Artificial Intelligence Review*, Vol. 46, 2016, pp. 445–458.

[37] T. Rahkar Farshi, “Battle royale optimization algorithm,” *Neural Computing and Applications*, Vol. 33, No. 4, 2021, pp. 1139–1157.

[38] M.D. Li, H. Zhao, X.W. Weng, and T. Han, “A novel nature-inspired algorithm for optimization: Virus colony search,” *Advances in Engineering Software*, Vol. 92, 2016, pp. 65–88.

[39] G.G. Wang, S. Deb, and L.D.S. Coelho, “Earthworm optimisation algorithm: A bio-inspired metaheuristic algorithm for global optimisation problems,” *International Journal of Bio-Inspired Computation*, Vol. 12, No. 1, 2018, pp. 1–22.

[40] S. Chinnasamy, M. Ramachandran, M. Amudha, and K. Ramu, “A review on hill climbing optimization methodology,” *Recent Trends in Management and Commerce*, Vol. 3, No. 1, 2022.

[41] A.I. Hafez, H.M. Zawbaa, E. Emary, and A.E. Hassanien, “Sine cosine optimization algorithm for feature selection,” in *International Symposium on Innovations in Intelligent Systems and Applications (INISTA)*. IEEE, 2016, pp. 1–5.

[42] S. Hassaine, F. Khomh, Y.G. Guéhéneuc, and S. Hamel, “IDS: An immune-inspired approach for the detection of software design smells,” in *Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 343–348.

[43] N. Moha, Y.G. Guéhéneuc, L. Duchien, and A.F. Le Meur, “Decor: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, Vol. 36, No. 1, 2009, pp. 20–36.

[44] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.G. Guéhéneuc et al., “Smurf: A SVM-based incremental anti-pattern detection approach,” in *19th Working conference on reverse engineering*. IEEE, 2012, pp. 466–475.

[45] F. Khomh, S. Vaucher, Y.G. Guéhéneuc, and H. Sahraoui, “BDTEX: A GQM-based Bayesian approach for the detection of antipatterns,” *Journal of Systems and Software*, Vol. 84, No. 4, 2011, pp. 559–572.

[46] F.A. Fontana, M.V. Mäntylä, M. Zanoni, and A. Marino, “Comparing and experimenting machine learning techniques for code smell detection,” *Empirical Software Engineering*, Vol. 21, No. 3, 2016, pp. 1143–1191.

[47] M. Kessentini and A. Ouni, “Detecting android smells using multi-objective genetic programming,” in *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 122–132.

[48] A. Kaur, S. Jain, and S. Goel, “SP-J48: A novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smells,” *Neural Computing and Applications*, Vol. 32, No. 11, 2020, pp. 7009–7027.

[49] S. Jain and A. Saha, “Rank-based univariate feature selection methods on machine learning classifiers for code smell detection,” *Evolutionary Intelligence*, Vol. 15, No. 1, 2022, pp. 609–638.

[50] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, "Competitive coevolutionary code-smells detection," in *Search Based Software Engineering: 5th International Symposium, SSBSE 2013, St. Petersburg, Russia, August 24–26, 2013. Proceedings 5*. Springer, 2013, pp. 50–65.
[51] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, Vol. 40, No. 9, 2014, pp. 841–861.
[52] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 24, No. 1, 2014, pp. 1–44.
[53] U. Mansoor, M. Kessentini, B.R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, Vol. 25, No. 2, 2017, pp. 529–552.
[54] G. Saranya, H.K. Nehemiah, A. Kannan, and V. Nithya, "Model level code smell detection using egapso based on similarity measures," *Alexandria Engineering Journal*, Vol. 57, No. 3, 2018, pp. 1631–1642.
[55] G. Saranya, H.K. Nehemiah, and A. Kannan, "Hybrid particle swarm optimisation with mutation for code smell detection," *International Journal of Bio-Inspired Computation*, Vol. 12, No. 3, 2018, pp. 186–195.
[56] M.M. Draz, M.S. Farhan, S.N. Abdulkader, and M. Gafar, "Code smell detection using whale optimization algorithm," *CMC-Computers Materials and Continua*, Vol. 68, No. 2, 2021, pp. 1919–1935.
[57] B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L.B. Said, "On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring," in *International Symposium on Search Based Software Engineering*. Springer, 2014, pp. 31–45.
[58] A. Ghannem, G.E. Boussaidi, and M. Kessentini, "Model refactoring using interactive genetic algorithm," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 96–110.
[59] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," in *33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1037–1039.
[60] T.J. Dea, M. Kessentini, W.I. Grosky, and K. Deb, "Software refactoring using cooperative parallel evolutionary algorithms," 2016.
[61] G. Saranya, H. Nehemiah, A. Kannan, and V. Pavithra, "Prioritizing code smell correction task using strength pareto evolutionary algorithm," *Indian Journal of Science and Technology*, Vol. 11, No. 20, 2018, pp. 1–12.
[62] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Journal*, Vol. 23, No. 2, 2015, pp. 323–361.
[63] A. Kaur, S. Jain, and S. Goel, "Sandpiper optimization algorithm: a novel approach for solving real-life engineering problems," *Applied Intelligence*, Vol. 50, No. 2, 2020, pp. 582–619.
[64] G. Lacerda, F. Petrillo, M. Pimenta, and Y.G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, Vol. 167, 2020, p. 110610.
[65] R.S. Menshawy, A.H. Yousef, and A. Salem, "Code smells and detection techniques: A survey," in *International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*. IEEE, 2021, pp. 78–83.
[66] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 23, No. 3, 2011, pp. 179–202.
[67] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li et al., "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 336–345.

[68] G. Van Rossum and F.L. Drake, Jr., *Python reference manual.* Centrum voor Wiskunde en Informatica Amsterdam, 1995.
[69] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, Vol. 12, 2011, pp. 2825–2830.
[70] S.B. Kotsiantis, D. Kanellopoulos, and P.E. Pintelas, "Data preprocessing for supervised leaning," *International Journal of Computer Science*, Vol. 1, No. 2, 2006, pp. 111–117.
[71] C.V.G. Zelaya, "Towards explaining the effects of data preprocessing on machine learning," in *35th international conference on data engineering (ICDE).* IEEE, 2019, pp. 2086–2090.
[72] M. Mehmood, N. Alshammari, S.A. Alanazi, and F. Ahmad, "Systematic framework to predict early-stage liver carcinoma using hybrid of feature selection techniques and regression techniques," *Complexity*, Vol. 2022, 2022, pp. 1–11.
[73] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise Reduction in Speech Processing.* Springer, 2009, pp. 1–4.
[74] T.T. Wong and P.Y. Yeh, "Reliable accuracy estimates from k-fold cross validation," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 32, No. 8, 2019, pp. 1586–1594.
[75] L. Abualigah, A. Diabat, S. Mirjalili, M. Abd Elaziz, and A.H. Gandomi, "The arithmetic optimization algorithm," *Computer Methods in Applied Mechanics and Engineering*, Vol. 376, 2021, p. 113609.
[76] J.S. Chou and D.N. Truong, "A novel metaheuristic optimizer inspired by behavior of jellyfish in ocean," *Applied Mathematics and Computation*, Vol. 389, 2021, p. 125535.
[77] H. Karami, M.V. Anaraki, S. Farzin, and S. Mirjalili, "Flow direction algorithm (FDA): A novel optimization approach for solving optimization problems," *Computers and Industrial Engineering*, Vol. 156, 2021, p. 107224.
[78] B. Das, V. Mukherjee, and D. Das, "Student psychology based optimization algorithm: A new population based optimization algorithm for solving optimization problems," *Advances in Engineering software*, Vol. 146, 2020, p. 102804.
[79] H. Yapici and N. Cetinkaya, "A new meta-heuristic optimizer: Pathfinder algorithm," *Applied Soft Computing*, Vol. 78, 2019, pp. 545–568.
[80] S. Mirjalili, "SCA: A sine cosine algorithm for solving optimization problems," *Knowledge-Based Systems*, Vol. 96, 2016, pp. 120–133.
[81] R. Rao, "Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems," *International Journal of Industrial Engineering Computations*, Vol. 7, No. 1, 2016, pp. 19–34.
[82] A. Askarzadeh, "A novel metaheuristic method for solving constrained engineering optimization problems: Crow search algorithm," *Computers and Structures*, Vol. 169, 2016, pp. 1–12.
[83] S. Mirjalili, "Dragonfly algorithm: A new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problems," *Neural Computing and Applications*, Vol. 27, 2016, pp. 1053–1073.
[84] A.H. Gandomi and A.H. Alavi, "Krill herd: A new bio-inspired optimization algorithm," *Communications in Nonlinear Science and Numerical Simulation*, Vol. 17, No. 12, 2012, pp. 4831–4845.
[85] S. Mirjalili, S.M. Mirjalili, and A. Hatamlou, "Multi-verse optimizer: a nature-inspired algorithm for global optimization," *Neural Computing and Applications*, Vol. 27, No. 2, 2016, pp. 495–513.
[86] M.Y. Cheng and D. Prayogo, "Symbiotic organisms search: A new metaheuristic optimization algorithm," *Computers and Structures*, Vol. 139, 2014, pp. 98–112.
[87] X.S. Yang, "Flower pollination algorithm for global optimization," in *International Conference on Unconventional Computing and Natural Computation.* Springer, 2012, pp. 240–249.
[88] R.V. Rao, V.J. Savsani, and D. Vakharia, "Teaching–learning-based optimization: a novel method for constrained mechanical design optimization problems," *Computer-Aided Design*, Vol. 43, No. 3, 2011, pp. 303–315.
[89] E. Rashedi, H. Nezamabadi-Pour, and S. Saryazdi, "GSA: A gravitational search algorithm," *Information Sciences*, Vol. 179, No. 13, 2009, pp. 2232–2248.

[90] D. Simon, "Biogeography-based optimization," *IEEE Transactions on Evolutionary Computation*, Vol. 12, No. 6, 2008, pp. 702–713.
[91] W.S. Noble, "What is a support vector machine?" *Nature Biotechnology*, Vol. 24, No. 12, 2006, pp. 1565–1567.
[92] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "KNN model-based approach in classification," in *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences.* Springer, 2003, pp. 986–996.
[93] M.N. Ab Wahab, S. Nefti-Meziani, and A. Atyabi, "A comprehensive review of swarm optimization algorithms," *PloS One*, Vol. 10, No. 5, 2015, p. e0122827.
[94] T.A. Jumani, M.W. Mustafa, A.S. Alghamdi, M.M. Rasid, A. Alamgir et al., "Swarm intelligence-based optimization techniques for dynamic response and power quality enhancement of AC microgrids: A comprehensive review," *IEEE Access*, Vol. 8, 2020, pp. 75 986–76 001.
[95] V. López, A. Fernández, and F. Herrera, "On the importance of the validation technique for classification with imbalanced datasets: Addressing covariate shift when data is skewed," *Information Sciences*, Vol. 257, 2014, pp. 1–13.
[96] R.F. Woolson, "Wilcoxon signed-rank test," *Wiley Encyclopedia of Clinical Trials*, 2007, pp. 1–3.
[97] T. Harris and J.W. Hardin, "Exact Wilcoxon signed-rank and Wilcoxon Mann–Whitney ranksum tests," *The Stata Journal*, Vol. 13, No. 2, 2013, pp. 337–343.
[98] D.H. Wolpert and W.G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, Vol. 1, No. 1, 1997, pp. 67–82.
[99] M. Meselhi, R. Sarker, D. Essam, and S. Elsayed, "A decomposition approach for large-scale non-separable optimization problems," *Applied Soft Computing*, Vol. 115, 2022, p. 108168.
[100] Z. Yang, H. Qiu, L. Gao, D. Xu, and Y. Liu, "A general framework of surrogate-assisted evolutionary algorithms for solving computationally expensive constrained optimization problems," *Information Sciences*, Vol. 619, 2023, pp. 491–508.
[101] W. Li, T. Zhang, R. Wang, S. Huang, and J. Liang, "Multimodal multi-objective optimization: Comparative study of the state-of-the-art," *Swarm and Evolutionary Computation*, 2023, p. 101253.
[102] Z. Beheshti and S.M.H. Shamsuddin, "A review of population-based meta-heuristic algorithms," *International Journal of Advances in Soft Computing and its Applic*, Vol. 5, No. 1, 2013, pp. 1–35.
[103] H. Grodzicka, A. Ziobrowski, Z. Łakomiak, M. Kawa, and L. Madeyski, "Code smell prediction employing machine learning meets emerging Java language constructs," *Data-Centric Business and Applications: Towards Software Development*, Vol. 4, 2020, pp. 137–167.
[104] S. Suthaharan, "Support vector machine," in *Machine learning models and algorithms for big data classification.* Springer, 2016, pp. 207–235.
[105] O. Kramer, "*k*-nearest neighbors," in *Dimensionality reduction with unsupervised nearest neighbors.* Springer, 2013, pp. 13–23.
[106] J. Han, J. Pei, and M. Kamber, *Data mining: Concepts and techniques.* Elsevier, 2011.
[107] J.F. O'Callaghan and D.M. Mark, "The extraction of drainage networks from digital elevation data," *Computer Vision, Graphics, and Image Processing*, Vol. 28, No. 3, 1984, pp. 323–344.

Appendix A. Machine learning algorithms used

Code smell detection exploits classification methodology as output is binary, that is, if the class/method is affected by a particular smell or not. We have implemented SVM and k-NN binary classifiers.

A.1. Support Vector Machine (SVM)

SVM is one of the most versatile machine learning algorithms for classification and regression problems. It assumes that two adjacent instances in input space must have the same output value [91]. It calculates an optimal hyperplane that separates classes in multi-dimensional space. The class of a new instance is marked by its position on which side of the hyperplane. Let's take N training samples $A = \{a_1, a_2, \ldots, a_N\}$ where a_i means i-th training sample and has f features. It is associated with one of the two classes $b_i \in \{0, 1\}$. So, complete training set is $\{(a_1, b_1), (a_2, b_2), \ldots, (a_N, b_N)\}$ where a_i training sample belongs to class b_i. The equation of hyperplane to be optimized is:

$$L_P = \frac{1}{2}||w||^2 + C\sum_{i=1}^{N}\epsilon_i - \sum_{i=1}^{N}\alpha_i\left(b_i(w^T a_i + y) - 1 + \epsilon_i\right) \tag{A.1.1}$$

where w is the weight vector, normal to the hyperplane, y is the threshold, and a is the input data point in d dimensional space. The algorithm aims to select the best values for threshold and weight such that the hyperplane is as far as possible from the closest data points. ϵ_i is the slack variable greater than equal to 0, and it is to be minimised. Each ϵ_i represents the distance between the i-th data point and the corresponding margin hyperplane. C is the regularization parameter that controls the trade-off between the slack variable penalty and the size of the margin. $\alpha_i \geq 0$ and $i = 1, 2, \ldots, N$. α_i are the Lagrange multipliers, and each α_i corresponds to one data point (a_i, b_i) and L_P becomes the primal equation that is to be optimized [104].

A.2. k-Nearest Neighbors

k-NN is one of the most popular machine learning algorithm and is known for its simplicity. It can be executed for both classification and regression problems. It analyses the parametric estimation of unknown probabilities, which are otherwise difficult to predict. The main idea behind this algorithm is that the class of a new data point is decided by its majority of k-neighbors. k-NN considers k nearest instances to determine the class of query instance and selects one with the highest frequency [105]. The distance metric is used to calculate the relative distance between instances in n-dimensional space, where n is the number of features. Minkowski distance is calculated as:

$$\sqrt[p]{\sum_{i=1}^{n}|x_i - y_i|^p}$$

It is a generalized distance metric. For Manhattan distance, p is equal to 1. For Euclidean distance, p is equal to 0, and for Chebyshev distance, $p \to \infty$ [106]. Meta-heuristic algorithms are used to retrieve the best values for k and p.

Appendix B. Meta-heuristic algorithms used

Meta-heuristic algorithms are general-purpose algorithms that do not rely on specific problem structures but search the solution space using heuristic rules inspired by natural phenomena, such as evolution, swarm intelligence, or physics. The current study uses the following meta-heuristic techniques to optimize machine learning algorithms.

B.1. Arithmetic optimization [75]

B.1.1. Behavior

Basic math operations like Addition (+), Subtraction (−), Multiplication (·) and Division (÷).

B.1.2. Phase changing variable

The algorithm strikes a balance between the exploration and exploitation phases using the MOA (Math Optimizer Accelerated) operator, which is calculated using the formula below:

$$MOA(C_t) = \text{Min} + C_t \left(\frac{\text{Max} - \text{Min}}{M_t} \right) \tag{B.1.1}$$

where C_t is t-th or current iteration and its value is between 1 and the maximum number of iterations, M_t, Max and Min are an accelerated function's maximum and minimum values, respectively. If the random number $r1$ is less than MOA, the exploration phase begins; otherwise, the algorithm proceeds with the exploitation phase.

B.1.3. Exploration equations

Exploration is done using either a division or multiplication search strategy due to their high dispersion property. The next position in exploration phase is determined using the following equations and conditions:

$$MOP(C_t) = 1 - \frac{{C_t}^{1/\alpha}}{{M_t}^{1/\alpha}} \tag{B.1.2}$$

$$x_{i,j}(C_t + 1) = \begin{cases} \text{best}(x_j)/(MOP + \text{esp})((UB_j - LB_j)\mu + LB_j) & \text{if } r2 < 0.5 \\ \text{best}(x_j) \cdot MOP((UB_j - LB_j)\mu + LB_j) & \text{otherwise} \end{cases} \tag{B.1.3}$$

where MOP means Math Optimizer Probability, $r2$ = random number that decides operator, C_t = current iteration, M_t = maximum iteration, $MOP(C_t)$ = value of the coefficient at t-th iteration, $\alpha = 5$ is sensitive parameter that defines the exploitation accuracy over the iterations, $x_i(C_t + 1)$ = i-th solution of the next iteration, $x_{i,j}(C_t)$ is i-th solution in the j-th position for the current iteration, best(x_j) is best-obtained solution so far in the j-th position, esp = small integer number, LB_j and UB_j are the lower bound and upper bound of the j-th position, respectively, $\mu = 0.5$ is a controlling parameter that adjusts the search process.

B.1.4. Exploitation equations

Subtraction and addition operators can quickly converge to a result in specific number of iterations due to their low dispersion property, thus perfect for exploitation.The random number $r3$ is employed to select operator. The next position in the exploitation phase is determined using the following equations and conditions:

$$x_{i,j}(C_t+1) = \begin{cases} \text{best}(x_j) - MOP((UB_j - LB_j)\mu + LB_j) & \text{if } r3 < 0.5 \\ \text{best}(x_j) + MOP((UB_j - LB_j)\mu + LB_j) & \text{otherwise} \end{cases} \tag{B.1.4}$$

Exploitation search operators, addition and subtraction, try to avoid plunging in local optima, and the stochastic nature of μ always allows exploration till the last iteration, introducing diversity in solutions.

B.2. Jellyfish search optimization [76]

B.2.1. Behavior

Swarming behaviour of jellyfish for foraging purposes. The solution is represented by area and its corresponding objective function depicting the quantity of food in that location.

B.2.2. Initialization

The population is initialized using a logistic map, $X_{i+1} = \rho X_i(1 - X_i)$ where $0 \leq X_0 \leq 1$. X_i is the chaotic logistic value of the i-th jellyfish's location and X_0 is a randomly generated location. $X_0, X_i \in [0,1]$, $X_0 \notin \{0.0, 0.25, 0.5, 0.75, 1.0\}$, and $\rho = 4$. If a jellyfish exceeds the boundaries of the search space, it will be relocated within the boundaries using the following equation:

$$X'_{i,d} = \begin{cases} (X_{i,d} - U_{b,d}) + L_b(d) & \text{if } X_{i,d} > U_{bd} \\ (X_{i,d} - L_{b,d}) + U_b(d) & \text{if } X_{i,d} < L_{bd} \end{cases} \tag{B.2.1}$$

where $X_{i,d}$ = location of the i-th jellyfish in d-dimensional search space, $X'_{i,d}$ = updated location after checking boundary constraints, L_{bd} and U_{bd} are the lower and upper bounds of search space. Jellyfish are attracted to places that have more nutrients or food. The new updated position of jellyfish can be evaluated using the following equations:

$$X_i(t+1) = X_i(t) + r_1 \cdot \overrightarrow{trend} \tag{B.2.2}$$

$$\overrightarrow{trend} = X^* - \beta \cdot r_2 \cdot \mu \tag{B.2.3}$$

where $(\beta > 0)$ = distribution coefficient related to the length of the $\overrightarrow{trend}$, $\overrightarrow{trend}$ = direction of ocean currents evaluated by averaging all the vectors from each jellyfish in the ocean to the jellyfish currently in the best location, X^* = jellyfish currently with the best location in the swarm, $e_c = \beta \cdot r_2$ is the attraction factor, μ = mean location of all the jellyfishes, t = current iteration, r_1, r_2 = random numbers between $[0,1]$.

B.2.3. Exploration equations

The new position in passive motion is determined using the following equation:

$$X_i(t+1) = X_i(t) + \eta \cdot \text{rand}(0,1) \cdot (U_b - L_b) \tag{B.2.4}$$

where L_b and U_b are the lower and upper bounds of the search space, η = motion coefficient, related to the length of motion around each jellyfish's location.

B.2.4. Exploitation equations

The active motion of i-th jellyfish is determined by the relative position of randomly selected j-th jellyfish. The following equations depict their movements:

$$X_i(t+1) = X_i(t) + r_3 \cdot \overrightarrow{Direction} \tag{B.2.5}$$

$$\overrightarrow{Direction} = \begin{cases} X_i(t) - X_j(t) & \text{if } f(X_i) < f(X_j) \\ X_j(t) - X_i(t) & \text{if } f(X_i) \geq f(X_j) \end{cases} \tag{B.2.6}$$

where $f(X_i)$, $f(X_j)$ = objective function values of i-th and j-th jellyfish, respectively, $\overrightarrow{Direction}$ = vector of the active motion, r_3 = random number between $[0,1]$, t = current iteration.

B.2.5. Phase changing variable

Time control function $c(t)$ is employed to decide the active and passive motion of jellyfish inside the bloom and also their movements toward ocean currents. It is calculated using the following formulae:

$$c(t) = \left| \left(1 - \frac{t}{t_{\max}}\right) \cdot (2 \cdot r_4 - 1) \right| \tag{B.2.7}$$

where t = current iteration, $t_{\max}$ = maximum iterations, $r_4 \in [0,1]$ = random number. The control function $c(t)$ fluctuates between 0 and 1 and decreases as iteration progresses. If the value of $(c(t) > C_0)$, the jellyfish follows the ocean current, otherwise, the jellyfish moves inside the jellyfish bloom. To determine the movement of jellyfish inside the swarm, the function $[1 - c(t)]$ is employed. If $(r_4 > [1 - c(t)])$, passive motion is favored otherwise, active motion is favored. That is how the algorithm converges to find an optimal solution and stops when the end criteria are met.

B.3. Flow direction optimization [77]

B.3.1. Behavior

The drainage basin system.

B.3.2. Initialization

It uses the D8 algorithm [107] to determine the flow direction of direct runoff (amount of water remaining on the ground surface after precipitation and mislaying such as interception, evapotranspiration, and infiltration). Direct runoff can be calculated using the formula:

$$r_d = \sum_{m=1}^{M} (R_m - \phi\delta t) \tag{B.3.1}$$

where ϕ = average amount of water loss during rainfall, R_m = rainfall, δt = time interval, M = total number of time steps.

Flows are population in drainage basin/search space with height as its objective function. Each flow, flows in a direction towards the lowest altitude with velocity V. The most optimal objective function is the basin's outlet point. Each flow with β neighbors has a neighborhood radius of δ and total population members are α. The initial position of flow is calculated using the following formula:

$$\mathit{Flow\_X}(i) = lb + \mathit{rand} \cdot (ub - lb) \tag{B.3.2}$$

where ub and lb are upper and lower limits of the decision variables and rand is a random number between $[0, 1]$ with uniform distribution. The position of the neighboring j-th flow can be determined using the following relation:

$$\mathit{Nghbr\_X}(j) = \mathit{Flow\_X}(i) + \mathit{rand}_n \cdot \delta \tag{B.3.3}$$

where rand_n is a random value with a normal distribution, a mean of zero, and a standard deviation of 1.

B.3.3. Phase changing variable

δ determines the phase of the algorithm. The small value of δ means exploitation, and the large value means exploration. δ starts with a significant value and is reduced over the iterations to support finding global solution and avoiding trapping in local optima. Randomness is introduced for that.

$$\delta = (\mathit{rand} \cdot X\mathit{rand} - \mathit{rand} \cdot \mathit{Flow\_X}(i)) \cdot ||\mathit{Best\_X} - \mathit{Flow\_X}(i)|| \cdot W \tag{B.3.4}$$

where rand = random number, $X\mathit{rand}$ = random position calculated using (B.3.1), W = non-linear weight with a random number from zero to infinity and is calculated using the following relation:

$$W = \left(\left(1 - \frac{\mathit{iter}}{\mathrm{Max}_{\mathit{iter}}}\right)^{2\cdot \mathit{rand}_n}\right) \cdot \left(\overline{\mathit{rand}} \cdot \frac{\mathit{iter}}{\mathrm{Max}_{\mathit{iter}}}\right) \cdot \overline{\mathit{rand}} \tag{B.3.5}$$

where $\overline{\mathit{rand}}$ is a random vector with uniform distribution.

B.3.4. Learning equations

Over the iteration, W has large variation, $Flow\_X(i)$ is closer to $Best\_X$, and the Euclidian distance between $Best\_X$ and $Flow\_X(i)$ is reduced to zero, bringing us closer to optimal solution. Each flow with V velocity moves towards the best neighbor. If the best neighbor has a better fitness value than that of a current flow, the flow velocity vector is updated using formula:

$$V = rand_n \cdot S_0 \tag{B.3.6}$$

where S_0 is the slope vector between the neighbor and the current position of the flow. Random number $rand_n$ reinforces exploration. The slope between neighbors i and j can be evaluated using the following calculation:

$$S_0(i,j,d) = \frac{Flow\_\text{fit}(i) - Nghbr\_\text{fit}(j)}{||Flow\_x(i,d) - Nghbr\_x(j,d)||} \tag{B.3.7}$$

where d is dimension of the problem. The new position can be calculated using following equation:

$$NewF_X(i) = Flow\_X(i) + V \cdot \frac{Flow\_X(i) - Nghbr\_X(j)}{||Flow\_x(i) - Nghbr\_x(j)||} \tag{B.3.8}$$

It is also possible that the fitness function of all neighbors is not less than the current flow, and then the algorithm randomly chooses another flow. The following relation shows how to simulate the flow direction under these conditions:

$$NewF\_X(i) = \begin{cases} Flow\_X(i) + rand_n \cdot (Flow\_X(r) - Flow\_X(i)) \\ \qquad\qquad \text{if } Flow\_\text{fit}(r) < Flow\_\text{fit}(i) \\ Flow\_X(i) + 2rand_n \cdot (Best\_X - Flow\_X(i)) \\ \qquad\qquad \text{otherwise} \end{cases} \tag{B.3.9}$$

where r and $rand_n$ are random integers.

B.4. Student psychology Based Optimization [78]

B.4.1. Behavior

The psychology of students making genuine efforts to improve their marks.

B.4.2. Learning equations

Students' overall marks are enhanced if the marks in each subject they are offered improves. Depending on their interest in a subject, the student may give more effort to improve overall marks. Students are categorized into four types based on their psychology:

(i) **Best Student.** The student who has the maximum overall marks is said to be the best student. They will try to maintain their position by putting in more effort than any randomly chosen student. Improvement of the best student can be evaluated using following equation:

$$X_{\text{bestnew}} = X_{\text{best}} + (-1)^k \cdot rand \cdot (X_{\text{best}} - X_j) \tag{B.4.1}$$

where X_{best} = marks obtained by best student, X_j = marks of randomly selected j-th student, $rand \in [0, 1]$ = random number, $k = 1$ or 2.

(ii) **Good Student.** The student who will try to give more effort in the subject of their interest to improve overall performance is a good student. They are subject-wise good students and are random because psychologies differ for each student.

$$X_{\text{new } i} = X_{\text{best}} + rand \cdot (X_{\text{best}} - X_i) \tag{B.4.2}$$

$$X_{\text{new } i} = X_i + rand \cdot (X_{\text{best}} - X_i) + rand \cdot (X_i - X_{\text{mean}}) \tag{B.4.3}$$

where X_i = marks of i-th student in that subject, X_{mean} = average marks of the class in that subject, If the student tries to give more or a similar effort to that of the best student, its improvement can be calculated using Eq (B.4.2). If the student gives more effort than an average student and the effort provided by the best student, their marks can be evaluated using Eq (B.4.3).

(iii) **Average Student.** While giving average effort to the subject, they will provide more effort to other exciting subjects. Their improvement can be calculated using the below formulae:

$$X_{\text{new } i} = X_i + rand \cdot (X_{\text{mean}} - X_i) \tag{B.4.4}$$

(iv) **Students trying randomly to improve.** They give random efforts to the subject irrespective of the students mentioned above. Their performance can be evaluated using the following formulae:

$$X_{\text{new } i} = X_{\min} + rand \cdot (X_{\max} - X_{\min}) \tag{B.4.5}$$

where $X_{\max}$ and $X_{\min}$ are the upper and lower bound on marks of the subject, respectively.

B.5. Pathfinder optimization [79]

B.5.1. Behavior

Swarms for foraging, breeding, and hunting purposes.

B.5.2. Learning equations

Each individual is a candidate solution in a d-dimensional space having a position vector as the fitness function. The algorithm is modeled to find prey as follows:

$$x_i^{t+1} = x_i^t + R_1 \cdot (x_j^t - x_i^t) + R_2 \cdot (x_p^t - x_i^t) + \eta, \quad i \geq 2 \tag{B.5.1}$$

where t = current iteration, x_i = position vector of the i-th search agent, x_j = position vector of the j-th search agent, x_p = position of pathfinder (leader), $R_1 = \alpha r_1$ and $R_2 = \beta r_2$, $r_1, r_2 \in [0, 1]$ = random numbers, α and β are randomly selected in the range of $[1, 2]$, α = coefficient of interaction that decides the magnitude of movement between two neighbors, β = coefficient of attraction that decides the movement of the herd with the leader, η is the vibration, which can be calculated using the following formulae:

$$\eta = \left(1 - \frac{t}{t_{\max}}\right) \cdot u_1 \cdot D_{ij}, \quad D_{ij} = ||x_i - x_j|| \tag{B.5.2}$$

where $u_1 \in [-1, 1]$ = random vector, D_{ij} = distance between two members, $t_{\max}$ = maximum iteration. The position of the pathfinder is updated according to the following equation:

$$x_p^{t+1} = x_p^t + 2r_3 \cdot (x_p^t - x_p^{t-1}) + A \tag{B.5.3}$$

where u_2 and r_3 are random vectors between $[-1, 1]$ and $[0, 1]$, respectively.

B.5.3. Phase Changing Variable

A is fluctuation factor, responsible for switching in the exploration and exploitation phases. It is calculated as follows:

$$A = u_2 \cdot e^{\frac{-2t}{t_{\max}}} \tag{B.5.4}$$

The position of the pathfinder provides the global optimum solution and converges with increasing iterations.

B.6. Sine Cosine Optimization [80]

B.6.1. Behavior

Sine and cosine functions.

B.6.2. Learning equations

The following equation decides the improved position and phase in Sine Cosine Optimization:

$$X_i^{t+1} = \begin{cases} X_i^t + r_1 \cdot \sin(r_2) \cdot |r_3 P_i^t - X_i^t|, & r_4 < 0.5 \\ X_i^t + r_1 \cdot \cos(r_2) \cdot |r_3 P_i^t - X_{ii}^t|, & r_4 \geq 0.5 \end{cases} \tag{B.6.1}$$

and

$$r_1 = a - t\frac{a}{T} \tag{B.6.2}$$

where X_i^t = position of the current solution in the i-th dimension at the t-th iteration, P_i^t = position of the destination point in the i-th dimension, r_1, r_2, r_3 and r_4 are random numbers, The r_1 is adaptive change calculated using Eq. (B.6.2), which is responsible for selecting the next search area; higher the value of r_1, the greater is the search area, t = current iteration, T = maximum number of iterations, a is a constant value, r_2 parameter decides the extent of the movement towards or away from the target and is in range $[0, 2\pi]$. r_3 is in the range $[-2, 2]$ and is a random weight score for the target that randomly asserts ($r3 > 1$) or refutes ($r3 < 1$) the influence of the target in determining the distance. r_4 is used to switch between the sine and cosine functions and lies between $[0, 1]$. The algorithm explores the search space when the sine and cosine functions range in $(1, 2]$ and $[-2, -1)$. However, exploits when the range is in the interval of $[-1, 1]$.

B.7. Jaya Optimization [81]

B.7.1. Behavior

The value of function $f(x)$, trying to get closer to $f(x)_{\text{best}}$, the best value and avoid $f(x)_{\text{worst}}$, the worst value of function to be optimized.

B.7.2. Learning equations

If n is the number of candidate solutions ($k = 1, 2, \dots, n$) and m is number of design variables ($j = 1, 2, \dots, m$), then $X_{j,k,i}$ is the value of j-th variable for the k-th candidate during i-th iteration. It is updated based on the following equation:

$$X'_{j,k,i} = X_{j,k,i} + r_{1,j,i}(X_{j,b,i} - |X_{j,k,i}|) - r_{2,j,i}(X_{j,w,i} - |X_{j,k,i}|) \tag{B.7.1}$$

where $X_{j,b,i}$ and $X_{j,w,i}$ is the best and worst value for j-th variable. $r_{1,j,i}$ and $r_{2,j,i}$ is the random number between $[0, 1]$. If updated solution is better, it is accepted and becomes input for next iteration.

B.8. Crow Search Optimization[82]

B.8.1. Behavior

Crow's mindful and intelligent behavior of stealing and hiding food.

B.8.2. Learning equations

Crows live in flocks and remember the hiding place of their food. They follow each other to steal the food and change their hiding places to avoid theft using probability. To begin with, positions of N crows are randomly initialized in d-dimensional search space. With iteration t, crow i will have memory of its hiding place, $m_{i,t}$. This is the best position that crow i has obtained so far.

Crows follow each other to search for the other food sources. Two cases arises, that is, if crow knows it is being followed or not. If crow j doesn't know that it is being followed by crow i, crow i will change its position according to first case. If crow j knows it is being followed by crow i, then random position is assigned. Following are the modeled equations:

$$x_{i,t+1} = \begin{cases} x_{i,t} + r_i \cdot fl_{i,t} \cdot (m_{j,t} - x_{i,t}) & r_i \geq AP_{j,t} \\ \text{random position} & \text{otherwise} \end{cases} \tag{B.8.1}$$

where $r_i \in [0, 1]$ is a random number and $fl_{i,t}$ is the flight length of i-th crow at t-th iteration. If value of fl is small, local search is favored otherwise global search is supported. $AP_{j,t}$ and $m_{j,t}$ is the awareness probability and memory of j-th crow at iteration t, respectively. AP helps in switching phases as high value of AP helps in exploration, while small value of AP guides toward exploitation. Memory function is updated as follows:

$$m_{i,t+1} = \begin{cases} F(x_{i,t+1}) & F(x_{i,t+1}) < F(m_{i,t}) \\ m_{i,k} & \text{otherwise} \end{cases} \tag{B.8.2}$$

where $F(\cdot)$ represents the objective function.

B.9. Dragonfly optimization [83]

B.9.1. Behavior

Swarming behavior of dragonflies – hunting (static swarm) and migration (dynamic swarm).

B.9.2. Learning equations

Position of each search agent is updated with two vectors, step (ΔX) and position (X). Step vector represents direction of the movement of dragonflies and is calculated by adding all the properties:

$$\Delta X_{t+1} = sS_i + aA_i + cC_i + fF_i + eE_i + w\Delta X_t \tag{B.9.1}$$

where s, a, c, f, and e are the weights associated with Separation $\left(S_i = -\sum_{j=1}^{N}(X - X_j)\right)$, Alignment $\left(A_i = \sum_{j=1}^{N} V_j \Big/ N\right)$, Cohesion $\left(C_i = \sum_{j=1}^{N} X_j \Big/ N - X\right)$, Attraction $(F_i = X^+ - X)$, and Distraction $(E_i = X^- + X)$ of i-th search agent, N = total number of neighboring agents, X = current agent, X_j = position of the j-th neighbor, V_j = velocity of the j-th neighbor, X^+ = position of the food, X^- = position of the enemy, w = inertia weight, t = iteration count. Position vector, $X_{t+1} = X_t + \Delta X_{t+1}$ is calculated next.

Swarming weights $(s, a, c, f, e$, and $w)$ are tuned adaptively and the radii of neighborhoods are increased proportional to the number of iterations to strike the balance between exploration and exploitation. To add to the randomness and exploration of the dragonflies movement, Lévy flight is being introduced in the new position as follows: $X_{t+1} = X_t + LF \cdot X_t$.

B.10. Krill herd optimization [84]

B.10.1. Behavior

Swarming and foraging behaviour of krills.

B.10.2. Learning equations

The position of a krill is dependent on three crucial factors – movement induced by other krills (N_i), foraging motion (F_i), and random diffusion (D_i). This is modeled by the following equation:

$$\frac{dx_i}{dt} = N_i + F_i + D_i \tag{B.10.1}$$

(i) **Motion induced by other krills.** The direction of motion induced for the i-th krill is:

$$N_i^{\text{new}} = N_{\max}\alpha_i + \omega_n N_i^{\text{old}} \tag{B.10.2}$$

where $N_{\max}$ = maximum speed induced, ω_n = inertia weight of motion induced in the range $[0,1]$, N_i^{old} = last motion induced, $\alpha_i = \alpha_i^{\text{local}} + \alpha_i^{\text{target}}$, α_i^{local} = local effect provided by the neighbors, α_i^{target} = target direction effect provided by the best krill. The impact of the neighboring krills in a krill movement is evaluated as follows:

$$\alpha_i^{\text{local}} = \sum_{j=1}^{n} \hat{K}_{i,j}\hat{x}_{i,j} \tag{B.10.3}$$

where $\hat{K}_{i,j}$ = normalized value of the similarity vector of the i-th krill, $\hat{x}_{i,j}$ = normalized value of related positions for the i-th krill, n = total number of neighboring krills. $\hat{K}_{i,j}$ is evaluated as:

$$\hat{K}_{i,j} = \frac{K_i - K_j}{K^{\text{worst}} - K^{\text{best}}} \tag{B.10.4}$$

where K_i and K_j are the fitness value of i-th and j-th neighboring krill, K^{worst} and K^{best} are the worst and best value of fitness for a krill so far, respectively, $\hat{x}_{i,j}$ is evaluated as:

$$\hat{x}_{i,j} = \frac{x_j - x_i}{||x_j - x_i|| + \epsilon} \tag{B.10.5}$$

where x_i and x_j are the positions of i-th krill and neighboring j-th krill, $||x_j - x_i||$ is the distance between j-th and i-th krill, ϵ is a small positive number added to avoid singularities. A sensing distance (d_s) is evaluated for each krill using formulae:

$$d_{si} = \frac{1}{5N}\sum_{j=1}^{n} ||x_i - x_j|| \tag{B.10.6}$$

wher d_{si} = sensing distance for the i-th krill, N = number of krills. Factor 5 is empirically obtained. If the distance of two krills is less than the defined sensing distance, they are neighbors. α_i^{target} is the effect of a krill with the best fitness on the i-th krill and leads to global optima. It is evaluated as follows:

$$\alpha_i^{\text{target}} = C^{\text{best}}\hat{K}_{i,\text{best}}\hat{x}_{i,\text{best}} \tag{B.10.7}$$

where $\hat{K}_{i,\text{best}}$ = best objective function value of the i-th krill, $\hat{x}_{i,\text{best}}$ = best position value of the i-th krill, C^{best} is the effective coefficient of the krill with the best fitness to the i-th krill and is evaluated as:

$$C^{\text{best}} = 2\left(rand + \frac{I}{I_{\max}}\right) \tag{B.10.8}$$

where $rand \in [0,1]$ = random number, I = current iteration, $I_{\max}$ = maximum number of iterations.

(ii) **Foraging motion.** The foraging motion of the i-th krill is a factor of two parameters, first is the food location, and the second is the previous experience with the food location and is calculated as:

$$F_i = V_f \beta_i + w_f F_i^{\text{old}} \tag{B.10.9}$$

where where V_f = parameter for tuning the foraging speed, β_i = centroid location of the i-th krill, w_f = inertia weight of the foraging speed in the range $[0, 1]$, F_i^{old} = last foraging motion value for the i-th krill. The centroid location of the i-th krill is evaluated as follows:

$$\beta_i = \beta_i^{\text{food}} + \beta_i^{\text{best}} \tag{B.10.10}$$

where β_i^{best} = best objective function value for the i-th individual, β_i^{food} is centroid attractive of the i-th krill and is determined as follows:

$$\beta_i^{\text{food}} = C^{\text{food}} \hat{K}_{i,\text{food}} \hat{x}_{i,\text{food}} \tag{B.10.11}$$

$$C^{\text{food}} = 2\left(1 - \frac{I}{I_{\max}}\right) \tag{B.10.12}$$

where $\hat{K}_{i,\text{food}}$ and $\hat{x}_{i,\text{food}}$ is the normalized value of the objective function and the normalized value of the i-th centroid. The center of the individual's food for each iteration is calculated as:

$$x^{\text{food}} = \frac{\displaystyle\sum_{i=1}^{n} \frac{1}{K_i} x_i}{\displaystyle\sum_{i=1}^{n} \frac{1}{K_i}} \tag{B.10.13}$$

The effect of the best objective function value of the i-th krill is evaluated as:

$$\beta_i^{\text{best}} = \hat{K}_{i,\text{ibest}} \hat{x}_{i,\text{ibest}} \tag{B.10.14}$$

where $\hat{K}_{i,\text{ibest}}$ and $\hat{x}_{i,\text{ibest}}$ is the best previous objective function value, and the best previously visited centroid of the i-th krill. The movement induced by other krills and forging movement decrease with increasing iterations.

(iii) **Physical diffusion.** Physical diffusion is the net movement of each krill from high-density to low-density regions. Physical diffusion for the i-th krill is determined as:

$$D_i = D_{\max}\left(1 - \frac{I}{I_{\max}}\right)\rho \tag{B.10.15}$$

where $D_{\max}$ = parameter for tuning the diffusion speed, ρ refers to an array containing random values between $[1, 1]$.

(iv) **Updating the krills.** The motion is induced by other krills, foraging motion, and physical diffusion change each krill's position toward the best objective function using the following equation:

$$x_i(I+1) = x_i(I) + \Delta t \frac{dx_i}{dt} \tag{B.10.16}$$

$$\Delta t = C_t \sum_{i=1}^{n} (UB_i - LB_i) \tag{B.10.17}$$

where Δt = sensitive constant, n = total krills, LB_i and UB_i are the lower and upper bounds of the i-th individual, C_t = constant value between $[0, 2]$ used as a scale factor of the speed vector.

B.11. Multi-verse optimization [85]

B.11.1. Behavior

The multi-verse theory, which implies that multiple universes have their own physical laws and an inflation rate that causes their expansion in space.

B.11.2. Learning equations

Universes interact through white holes, black holes, and wormholes. White holes play a significant role in the birth of the universe and have a high inflation rate. Black holes have a colossal gravitational force that gulps everything inside, even light. Wormholes are tunnels through which objects can travel among universes or galaxies. The fitness function of each universe/solution is proportional to the inflation rate. Initially, universes are sorted according to their inflation rates and one is selected randomly through the roulette wheel mechanism to be a white hole. This is done by the following equation:

$$x_i^j = \begin{cases} x_k^j & r1 < NI(U_i) \\ x_i^j & r1 \geq NI(U_i) \end{cases} \tag{B.11.1}$$

where $U[n, d]$ = matrix represents the complete universe with all elements, d = total number of parameters, n = number of universes, x_i^j = j-th parameter in the i-th universe, NI is normalized inflation rate of the i-th universe, $r1 \in [0, 1]$ = random number, x_k^j = j-th parameter of the k-th universe chosen by roulette wheel. This allows universes to exchange objects and improve inflation rates. Things are also randomly exchanged between universes through wormholes, and it is assumed that wormholes are established between others and the best universe formed yet. This mechanism can be formulated as follows:

$$x_i^j = \begin{cases} \begin{cases} X_j + TDR \cdot ((ub_j - lb_j) \cdot r4 + lb_j) & r3 < 0.5 \\ X_j - TDR \cdot ((ub_j - lb_j) \cdot r4 + lb_j) & r3 \geq 0.5 \end{cases} & r2 < WEP \\ x_i^j & r2 \geq WEP \end{cases} \tag{B.11.2}$$

where X_j is the j-th parameter of the best universe obtained so far, ub_j and lb_j are the upper and lower bound values of the j-th variable, x_i^j is the j-th parameter of the i-th universe, and $r2, r3, r4$ are random numbers between $[0, 1]$, WEP and TDR are coefficients used in the equation and can be calculated using the formula:

$$WEP = \min + l \cdot \left(\frac{\max - \min}{L}\right) \tag{B.11.3}$$

$$TDR = 1 - \frac{l^{1/p}}{L^{1/p}} \tag{B.11.4}$$

where *WEP* (Wormhole Existence Probability) increases linearly over iterations to support exploitation, L = maximum number of iteration, l = current iteration, *max* and *min* are 1 and 0.2 by default, *TDR* (Traveling Distance Rate) defines the distance rate by which an object can be transferred to the best universe obtained yet, $p = 6$ is exploitation precision, the higher its value, the faster the exploitation.

B.12. Symbiotic organisms search [86]

B.12.1. Behavior

The symbiotic relationships that exist between paired organisms to survive in the ecosystem.

B.12.2. Learning equations

Each organism in an algorithm is a solution in the d-dimensional search space, and they are refined through three phases applied serially. The three phases are explained as follows:

(i) **Mutualism.** Mutualism relationship benefits both the interacting organisms. Let X_i is an organism in search space, and X_j is a random candidate interacting with i-th organism to increase mutual survival advantage. Their positions are updated according to the following equations:

$$X_i^* = X_i + \text{rand}(0,1) \cdot (X_{\text{best}} - \mathit{Mutual\ Vector} \cdot BF_1) \tag{B.12.1}$$

$$X_j^* = X_j + \text{rand}(0,1) \cdot (X_{\text{best}} - \mathit{Mutual\ Vector} \cdot BF_2) \tag{B.12.2}$$

$$\mathit{Mutual\ Vector} = \frac{X_i + X_j}{2} \tag{B.12.3}$$

where rand is a random function that produces a number between 0 and 1, X_{best} is the best position searched by all organisms yet or best fitness value, BF_1 and BF_2 are the benefit factor of i-th and j-th organism, respectively. Their values are either 1 or 2.

(ii) **Commensalism.** Commensalism only benefits one organism, and the other one remains unaffected. The interaction between X_i and X_j is updated as follows:

$$X_i^* = X_i + \text{rand}(-1,1) \cdot (X_{\text{best}} - X_j) \tag{B.12.4}$$

where only X_i is benefited from the interaction while X_j neither benefits nor gets harmed from it. $(X_{\text{best}} - X_j)$ represents that benefit.

(iii) **Parasitism.** In this phase, only one candidate benefits and the relationship harms the other candidate. Parasite Vector is created in search space by copying X_i to interact with host X_j. Parasite Vector replaces X_j if it has better fitness value; otherwise, X_j survives.

$$X_j = \begin{cases} PV & \text{if } f(PV) > f(X_j) \\ X_j & \text{if } f(PV) \leq f(X_j) \end{cases} \tag{B.12.5}$$

where PV is a parasite vector and $f(\cdot)$ represents fitness function.

B.13. Flower pollination optimization [87]

B.13.1. Behavior

The reproduction mechanism of flowers through pollination in nature.

B.13.2. Exploration equations

When pollinators such as bees that can fly far and may portray Lévy flight behavior, contributes to cross-pollination and are considered global pollination. For single objective problems, it can be assumed that each plant has only one flower, and each flower has only one pollen, referred to as a solution x_i. Global pollination ensures the reproduction of the most fittest flowers and can be represented as g_*. Consistency of a flower is its reproduction probability and can be calculated using the following formula:

$$x_i^{t+1} = x_i^t + L(x_i^t - g_*) \tag{B.13.1}$$

where x_i^t is the solution vector at iteration t for pollen i and g_* is the best solution found yet. L is the step size drawn from Lévy flight distribution, it can be evaluated as:

$$L \sim \frac{\lambda\Gamma(\lambda)\sin(\pi\lambda/2)}{\pi}\frac{1}{s^{1+\lambda}}, (s \gg s_0 > 0) \tag{B.13.2}$$

where $\Gamma(\lambda)$ is the standard gamma function with an index λ. This distribution works for large step sizes, $s > 0$.

B.13.3. Exploitation equations

Local pollination is when self-pollination happens through abiotic means. For local pollination, flower consistency is calculated as follows:

$$x_i^{t+1} = x_i^t + \epsilon(x_j^t - x_k^t) \tag{B.13.3}$$

where x_j^t and x_k^t are pollens from the different flowers of the same plant species. If x_j^t and x_k^t are selected from the same population and ϵ is from a uniform distribution in $[0, 1]$, this equation will represent the local random walk. The switch between local and global pollination is controlled by parameter p which ranges between $[0, 1]$.

B.14. Teaching learning based optimization [88]

B.14.1. Behavior

The traditional teaching-learning phenomenon of a classroom.

B.14.2. Learning equations

The teacher tries to train learners in the best way possible to increase their level of knowledge. All the learners are the population and the teacher is the best solution. Design variables are different subjects taught in the class, and the result is analogous to the fitness value. Teaching is done in two phases:

(i) **Teacher Phase.** Learners learn from best learner, i.e., teacher. Teacher tries to elevate the mean of class, best to his abilities. Let, M_i be the mean of class result and T_i be the teacher at any iteration i. T_i will try to improve M_i to M_{new}. So, new solution is updated according to the following equation:

$$X_{\text{new},i} = X_{\text{old},i} + r_i(M_{\text{new}} - T_F M_i) \tag{B.14.1}$$

where $r_i \in [0,1]$ = random number, T_F is a teaching factor that can have value either 1 or 2. It is a heuristic step and decided randomly with equal probability as $T_F = \text{round}(1 + \textit{rand}(0,1)\{2-1\})$.

(ii) **Learner Phase.** Learners learn from teachers or among themselves through presentations, discussions or formal communication. A learner will gain knowledge if another learner has more knowledge than him. For leaner X_i in the class, the updating mechanism is as follows:

$$newX_i = \begin{cases} X_i + rand \cdot (X_i - X_k) & f(X_i) < f(X_k) \\ X_i + rand \cdot (X_k - X_i) & \text{otherwise} \end{cases} \tag{B.14.2}$$

where $newX_i$ = new positions of the i-th learner X_i, X_k = random learner from the class, $f(X_i)$ = fitness values of the learner X_i, $f(X_k)$ = fitness values of the learner X_k, $rand \in [0,1]$ = random number.

B.15. Gravitational search optimization [89]

B.15.1. Behavior

Newton's law of gravity and the second law of motion.

B.15.2. Learning equations

Each agent has its Position X, Active Gravitational Mass (M_a), Passive Gravitational Mass (M_p), and Inertial Mass (M_i). The position is the solution of the problem and masses are evaluated using the fitness function. Gravitational force applies to all agents; thus global movement of all agents is forced towards heavier masses supporting exploitation and an optimum solution in the search space. The system of N agents with their initial positions is defined as follows:

$$X_i = (x_i^1, x_i^2, \ldots, x_i^d, \ldots, x_i^n) \text{ for } i = 1, 2, \ldots, N. \tag{B.15.1}$$

where x_i^d is the position of the i-th search agent in the d-th dimension. At any given time t, the gravitational force acting between agent i and agent j is:

$$F_{ij}^d(t) = G(t)\frac{M_{pi}(t) \cdot M_{aj}(t)}{R_{ij}(t) + \epsilon}(x_j^d(t) - x_i^d(t)) \tag{B.15.2}$$

where M_{pi} = passive gravitational mass of the i-th object, M_{aj} = active gravitational mass of the j-th agent, ϵ is a constant, $G(t)$ = gravitational constant at time t, $R_{ij}(t)$ is the Euclidean distance between two agents i and j, can be calculated as:

$$R_{ij}(t) = \sqrt{\sum_{k=1}^{n}(X_{ik}(t) - X_{jk}(t))^2} \tag{B.15.3}$$

To support exploration, a random factor is added to the total force acting on an agent. It can be represented as:

$$F_i^d(t) = \sum_{j=1, j\neq i}^{N} rand_j \; F_{ij}^d(t) \tag{B.15.4}$$

where $rand_j$ is a random number between $[0, 1]$. According to the law of motion, acceleration of the i-th agent in the d-th dimension at time t is:

$$a_i^d(t) = \frac{F_i^d(t)}{M_{ii}(t)} \tag{B.15.5}$$

where M_{ii} is the inertial mass of the i-th agent. The next velocity and position of i-th agent can be updated using following formulas:

$$v_i^d(t+1) = rand_i \cdot v_i^d(t) + a_i^d(t) \tag{B.15.6}$$

$$x_i^d(t+1) = x_i^d(t) + v_i^d(t+1) \tag{B.15.7}$$

where $rand_i$ is a uniform random number in between $[0, 1]$. The gravitational constant will be reduced over time to control the search accuracy. If it is assumed that gravitational and inertial masses are equal, they can be updated as follows using the map of fitness:

$$M_{ai} = M_{pi} = M_{ii} = M_i, i = 1, 2, \ldots, N. \tag{B.15.8}$$

$$M_i(t) = \frac{m_i(t)}{\sum_{j=1}^{N} m_j(t)} \tag{B.15.9}$$

$$m_i(t) = \frac{\text{fit}_i(t) - \text{worst}(t)}{\text{best}(t) - \text{worst}(t)} \tag{B.15.10}$$

$$\text{best}(t) = \min_{j\in\{1,2,\ldots,N\}} \text{fit}_j(t) \tag{B.15.11}$$

$$\text{worst}(t) = \max_{j\in\{1,2,\ldots,N\}} \text{fit}_j(t) \tag{B.15.12}$$

where $\text{fit}_i(t)$ is the fitness value of the i-th agent at time t. The number of agents over iteration reduces to maintain a robust balance between exploration and exploitation, so only heavy mass agents apply their force on other agents stored as $Kbest$. It is the function of time, and has an initial value of K_0 which decreases with time. Initially, all search agents apply the force, and with each iteration, $Kbest$ is linearly reduced and in the end, there will be just one agent applying force to the others. Therefore, force is updated as:

$$F_i^d(t) = \sum_{j \in Kbest, j \neq i} rand_j \, F_{ij}^d(t) \tag{B.15.13}$$

where $Kbest$ is the set of first K agents with the best fitness value and biggest mass.

B.16. Biogeography-based optimization [90]

B.16.1. Behavior

The migration of species between islands.

B.16.2. Learning equations

Habitats with good and favorable living conditions have high Habitat Suitability Index (HSI), represents good solutions and have high emigration rate and low immigration rate. Islands with low HSI represent poor solutions but have a high immigration rate due to their sparse species count and low emigration rate. Immigration and emigration rates are fitness functions of a habitat. The factors influencing HSI are called Suitability Index Variables (SIVs) and are considered to be the independent variables of the habitat. The algorithm has two main steps, migration and mutation.

(i) **Migration.** It is a probabilistic operator that improves a habitat H_i. Each habitat's migration rate is used to share features between habitats. For each habitat H_i, its immigration rate (λ_i) is used to decide whether or not to immigrate. If immigration is selected, the emigrating habitat H_j is selected probabilistically based on the emigration rate (μ_i). Rates and Migration are defined as follows:

$$\mu_i = \frac{Ei}{N} \tag{B.16.1}$$

$$\lambda_i = I\left(1 - \frac{i}{N}\right) \tag{B.16.2}$$

$$H_i(SIV) \leftarrow H_j(SIV) \tag{B.16.3}$$

where N is the total population size.

(ii) **Mutation.** It is a probabilistic operator that randomly modifies a habitat's SIV based on the habitat's a priori species count probability. The purpose of mutation tends to increase diversity among the population. For low HSI solutions, mutation gives them a chance to enhance the quality of solutions, and for high HSI solutions, the mutation can improve them even more than they already have.